Vérification formelle d'une méthodologie pour la conception et la production de systèmes numériques critiques

Formal verification of a methodology for the design and production of safety-critical digital systems

Présentée par Vincent IAMPIETRO
Le 16 décembre 2021

**Sous la direction de David Delahaye
et David Andreu**

Devant le jury composé de

Christian Retoré, Professeur, Université de Montpellier	Président du jury
David Déharbe, Docteur, Ingénieur, Clearsy	Examinateur
Marc Pouzet, Professeur, ENS, Paris	Examinateur
David Andreu, MCF, Université de Montpellier/Neurinnov	Co-Directeur
Sandrine Blazy, Professeure, Université de Rennes	Rapporteure
Frédéric Boniol, Maître de recherche, ONERA, Toulouse	Rapporteur
David Delahaye, Professeur, Université de Montpellier	Directeur

**UNIVERSITÉ
DE MONTPELLIER**

Remerciements

Je remercie mes deux directeurs de thèse David Delahaye et David Andreu pour avoir initié ce projet de thèse et m'avoir permis d'y participer. Durant ces trois ans, nous avons appris à nous connaître et à travailler ensemble (même si en bonne partie à distance, à cause de ce virus dont il est interdit de prononcer le nom...). Grâce à vous, j'ai pu non seulement acquérir des compétences en informatique, en mathématiques, en automatisme et électronique, mais aussi des compétences en termes d'organisation et de conduite de projets. Je vous suis reconnaissant pour votre patience et votre pédagogie à mon égard.

Je remercie aussi toute l'équipe MAREL du LIRMM pour m'avoir accueilli, et m'avoir apporté des conseils et des critiques constructives vis-à-vis de mon travail ; notamment lors des fameux séminaires de l'équipe. Au sein de l'équipe MAREL, je remercie tout particulièrement mon co-bureau Nicolas Hlad et Pascal Zaragoza (qui était dans un bureau un peu plus loin) qui ont été des soutiens tout au long de ma thèse, et sont maintenant des amis.

Je remercie également les autres doctorants du LIRMM, notamment ceux de l'équipe MAORE, à savoir Gabriel Volte, Tom Davot, Samuel Masseport (pour ne citer que les plus permanents), qui ont aussi été des soutiens au quotidien.

Finalement, je remercie mes amis et ma famille pour leur soutien, leur amour et l'intérêt qu'ils ont pu porter à mon projet de thèse.

Pour conclure, merci à toutes les personnes qui ont contribué de près ou de loin à ce projet, et à l'écriture de ce mémoire.

Contents

Résumé

La production de circuits numériques complexes est devenue impossible sans l'aide des ordinateurs. La méthodologie HILECOP (HIgh LEvel hardware COmponent Programming) assiste les ingénieurs dans la conception et la production de circuits numériques dans le contexte des systèmes critiques, i.e. systèmes dont le malfonctionnement peut résulter en la perte de vies humaines, des catastrophes naturelles, des désastres économiques, etc. À titre d'exemple, la société Neurinnov[1] applique la méthodologie HILECOP pour la production de neuroprothèses, considérées comme des dispositifs médicaux hautement critiques par la loi de régulation de l'UE[2]. Dans HILECOP, les ingénieurs produisent un modèle de circuit numérique. Ils utilisent un formalisme graphique qui regroupe les diagrammes à composant et un type particulier de réseaux de Petri (RdP). Ensuite, le modèle est transformé en une représentation textuelle intermédiaire décrite en langage VHDL (Very high speed integrated circuit Hardware Description Language). Finalement, un compilateur/synthétiseur industriel génère un circuit numérique physique, i.e. un ASIC ou sur carte FPGA, depuis la représentation VHDL. Ici, l'utilisation des RdPs est liée au contexte des systèmes critiques. Les RdPs permettent la vérification de propriétés sur les modèles de circuits numériques grâce à l'application de techniques de *model-checking*. Cependant, une des transformations décrite dans la méthodologie HILECOP pourrait altérer le *comportement* (ou *sémantique*) des modèles initiaux, invalidant ainsi les précédentes étapes de vérification. Le but de cette thèse est de prouver que la transformation modèle-vers-texte de HILECOP, qui génère une description VHDL depuis un modèle de circuit numérique, préserve le comportement des modèles d'entrée, i.e.: pour tout modèle passé en entrée de la transformation, la description VHDL résultante se comporte de la même manière. Pour prouver cette propriété, nous nous inspirons des travaux menés sur la vérification formelle de compilateurs (notamment sur le compilateur C certifié CompCert [72]). Notre approche est celle de la vérification déductive interactive avec assistants de preuve. Dans ce contexte, les étapes pour établir la propriété de préservation de comportement de la transformation sont : (1) formaliser la sémantique d'exécution de la représentation source, (2) de la représentation cible, (3) décrire la transformation, et (4) prouver un théorème de préservation sémantique. Même en suivant ce processus clairement détaillé, les spécificités de la transformation modèle-vers-texte de HILECOP (comparaît notamment aux compilateurs) apportent de nouvelles questions recherches et des challenges à chaque étape. Dans cette thèse, nous utilisons l'assistant à la preuve Coq pour nous accompagner tout au long du processus. Finalement, nous avons prouvé que la transformation de HILECOP préserve le comportement de tous modèles initiaux. La mécanisation complète de la preuve avec Coq est un travail en cours.

---

[1] https://neurinnov.com/

[2] https://eur-lex.europa.eu/eli/reg/2017/745/2020-04-24

Abstract

The complexity of digital hardware circuits makes it difficult to produce them without the help of computers. The HILECOP (HIgh LEvel hardware COmponent Programming) methodology aims at the assistance of engineers in the design and production of such digital circuits. The context of production is the one of *safety-critical* digital systems, i.e. systems which failure could result in direct human losses, natural catastrophes, economic disasters, etc. To give an example, the Neurinnov[3] company leverages the HILECOP methodology to produce highly critical medical devices known as neuroprostheses. In HILECOP, engineers rely on a graphical formalism, based on component diagrams and a particular kind of Petri nets, to produce a model of a digital circuit. Then, a computer program turns the model into an intermediary description written in VHDL (Very high speed integrated circuit Hardware Description Language). Finally, an industrial compiler/synthesizer transforms the VHDL description into a concrete physical circuit on an FPGA, or as an ASIC. The use of Petri nets permits the engineers to describe a *formal* model of a digital circuit. The mathematical foundations of Petri nets enable the use of model-checking techniques. Thus, proofs can be brought that the produced models verify certain *soundness* properties. However, even with a *sound* model of a circuit, one transformation step could alter the behavior of the initial model. The goal of this thesis is to bring the formal proof that the model-to-text transformation from a HILECOP high-level model to a VHDL description is *semantic preserving* (or *behavior preserving*); i.e. for all high-level model given as an input to the transformation, the resulting VHDL description behaves similarly. To perform this task, we draw our inspiration from the works pertaining to the formal verification of compilers for programming languages (especially from the certified C compiler CompCert [72]). Specifically, we are interested in proving the property of semantic preservation in the context of *deductive* verification with proof assistants. In this context, the steps to verify that a transformation is semantic preserving include: (1) the formalization of the execution semantics of the source representation, (2) of the target representation, (3) the formal description of the transformation, and (4) the proof of a corresponding semantic preservation theorem. In this thesis, these steps have been carried within the framework of the Coq proof assistant. Even though these steps are clearly set, the specificities of the HILECOP model-to-text transformation, compared to compilers for programming languages, bring some interesting research challenges. Finally, we have brought the *paper* proof that the HILECOP transformation is semantic preserving by demonstrating a related behavior preservation theorem. The full mechanization of the proof using the Coq proof assistant is an ongoing task.

---

[3] https://neurinnov.com/

Résumé étendu

0.1   Introduction

Pour répondre aux contraintes liées à la conception de circuits numériques critiques, et à l'augmentation constante de la complexité des systèmes, le domaine de l'Ingénierie Système à Base de Modèles (ISBM) a été développé. L'intérêt est de travailler sur des modèles de haut niveau avec un pouvoir d'expression et des qualités de compréhension et de lisibilité qui facilitent les interactions entre les acteurs de la conception du circuit (i.e, les ingénieurs). Plusieurs formalismes existent : le langage SysML [49], des variantes du langage C [116], ou encore les réseaux de Petri (RdPs) [113], pour citer les plus répandus. Une fois la conception terminée, les modèles sont physiquement synthétisés en suivant un procédé manuel ou automatique. Il reste alors à prouver que la phase de transformation préserve le comportement du modèle de conception. La présente thèse s'intéresse à la vérification d'un processus d'aide à la modélisation et à la production de circuits numériques critiques : la méthodologie HILECOP (HIgh LEvel hardware COmponent Programming). Cette méthodologie est mise en œuvre dans le cadre de la création de micro-contrôleurs intégrés à des dispositifs médicaux de type neuroprothèses. La Figure 1 en décrit les principales étapes.
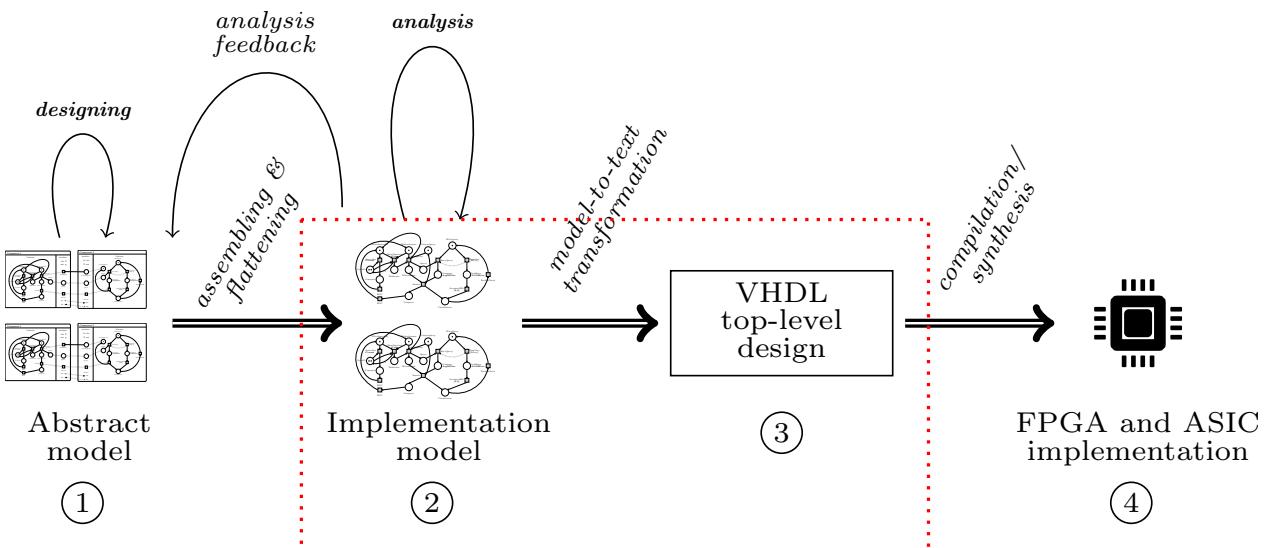
FIGURE 1: Principe de la méthodologie HILECOP; les double flèches horizontales représentent des phases de transformation; les simple flèches indiquent les autres types d'opérations ayant cours à une étape précise, ou entre étapes.

Le concepteur de systèmes électroniques esquisse premièrement un modèle graphique de haut niveau de son circuit (①). Ce modèle s'appuie sur le formalisme des diagrammes à composants, avec l'addition des RdPs pour décrire le comportement interne des parties du circuit. Dans un deuxième temps, les parties du modèle sont assemblées et la structure des composants est effacée. Le résultat obtenu est un réseau de Petri global décrivant le système modélisé (②). Des outils d'analyse exploités par la méthodologie permettent alors de vérifier certaines propriétés du modèle (caractère borné, vivacité...) et présentent un compte rendu au concepteur. Après plusieurs itérations du cycle analyse-correction, du code VHDL est généré à partir du modèle d'implémentation (③). Dès lors, la dernière étape de la méthodologie, qui opère la synthèse du circuit électronique depuis le code source VHDL, est prise en charge par un compilateur/synthétiseur industriel propriétaire (④).

L'objectif de la thèse est de prouver que la transformation du modèle d'implémentation en code VHDL (i.e, de ② vers ③ dans la Figure 1) n'introduit pas de divergences de comportement. Dans cette optique, il sera nécessaire de formaliser la sémantique des modèles de haut niveau (RdP), du langage cible (VHDL), et de décrire la transformation. Ensuite, la preuve de similarité comportementale devra être établie. L'intégralité de la démarche sera mécanisée avec l'assistant à la preuve Coq [106]. Même si cette démarche a été éprouvée pour la vérification de compilateurs, son application à la conception de circuits numériques est bien moins fréquente. L'intérêt scientifique provient de la distance qui existe entre le modèle d'exécution du formalisme source (SITPN) et celui du langage cible (VHDL). Cette distance devra être prise en compte lors de la preuve de préservation de comportement.

0.2 Un formalisme de haut-niveau : les réseaux de Petri

Du fait de leur statut de modèles formels et des possibilités d'analyse qui en résultent, les RdPs ont été retenus comme modèles de haut niveau de la méthodologie HILECOP. Le but de la méthodologie étant la conception et la production de circuits numériques *critiques*, les modèles se doivent d'être validés par analyse formelle. Afin d'augmenter l'expressivité des modèles, les RdPs HILECOP combinent plusieurs classes connues de RdPs (présentées ci-après), mais leur particularité réside dans leur exécution synchrone. Les RdPs HILECOP sont nommés SITPNs pour Synchronously executed Interpreted Time Petri Nets with priorities.

Les SITPNs sont des RdP interprétés; des actions peuvent être associées aux places d'un réseau et des fonctions/conditions peuvent être associées aux transitions. Actions et fonctions définissent des opérations sur une ensemble de variables, ici, des *signaux* VHDL. Les conditions associées aux transitions sont des expressions Booléennes sur la valeur des signaux. Dans un RdP interprété, une transition est franchissable si elle est sensibilisée et que toutes les conditions qui lui sont associées sont *vraies*. La Figure 2 donne un exemple de RdP interprété.
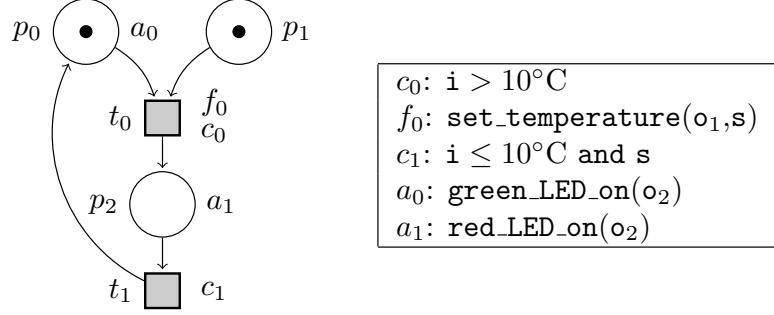
FIGURE 2: Un exemple de réseau de Petri interprété; sur le côté gauche, le RdP; sur le côté droit, les expressions Booléennes associées aux conditions et les opérations associées aux actions et fonctions.

Les RdP utilisés dans HILECOP sont temporels ; une fenêtre de tir, i.e un intervalle de temps, peut être associée à une transition. Un compteur de temps est lancé lorsqu'une transition devient sensibilisée; celle-ci devient franchissable lorsque son compteur de temps a atteint l'intervalle de tir. La Figure 3 donne un exemple de RdP temporel. La valeur courante des compteurs de temps est représentée entre chevrons en dessous des intervalles temporels associés. En résumé, une transition d'un SITPN est franchissable si elle est sensibilisée, si toutes les conditions qui lui sont associées sont vraies et si son compteur de temps est dans l'intervalle défini.
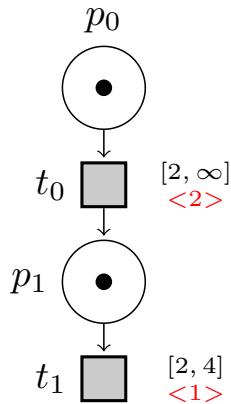
FIGURE 3: Un exemple de RdP temporel. La valeur des compteurs de temps apparaît en rouge.

Contrairement au cas général, les SITPNs ont une politique de tir (i.e, une sémantique) *synchrone*. Fondamentalement, le tir des transitions d'un RdP est un phénomène indéterministe (si deux transitions sont franchissables au même instant, tous les ordres de tirs sont possibles), et asynchrone (dès qu'une transition est franchissable, elle peut être tirée sans attente). A contrario, l'évolution d'un SITPN est rythmée par le front montant et le front descendant d'un signal d'horloge, comme montré dans la Figure 4. Sur le front descendant (① de la Figure 4), toutes les transitions devant être tirées sont déterminées, ce après mise à jour des conditions et intervalles de temps ; sur le front montant (② de la Figure 4), les précédentes transitions

sont tirées, entraînant la mis à jour du marquage du réseau et l'exécution de fonctions. La sémantique d'évolution d'un tel réseau est synchrone et déterministe.
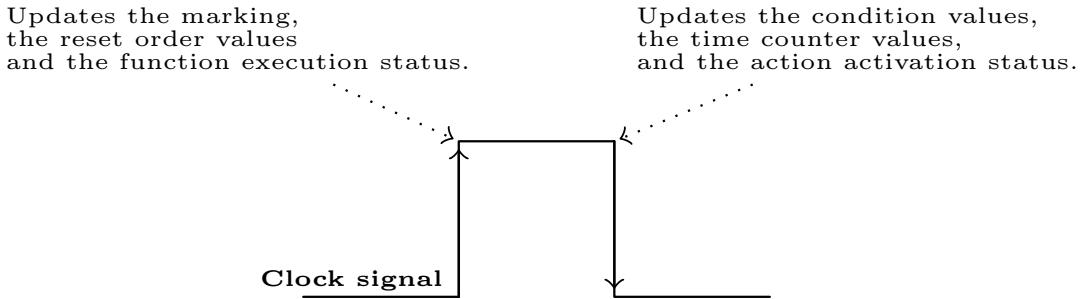
FIGURE 4: Evolution d'un SITPN synchronisée avec un signal d'horloge.

La structure et la sémantique des SITPNs ont été formalisées dans [71, 78]. La sémantique est exprimée comme un système états-transitions où les transitions sont étiquetées par les évènements d'un signal d'horloge. Il y a deux évènements possibles : le front montant et le front descendant du signal. L'état d'un SITPN décrit, entre autres, le marquage courant du SITPN, la valeur des compteurs de temps et des conditions associés aux transitions, la liste des transitions à tirer... La sémantique des SITPNs fixe les règles de changement d'état en fonction des évènements d'horloge. Par exemple, sur le front descendant d'horloge, la liste des transitions à tirer au prochain front montant est calculée; une règle stipule qu'une aucune transition non franchissable au front descendant n'appartient à l'ensemble des transitions à tirer.

La première contribution de la thèse est l'implantation en Coq de la structure et de la sémantique des SITPNs. La sémantique a été implantée comme une relation inductive paramétrée par un SITPN, deux états (i.e, avant et après transition), et un évènement d'horloge. La relation présente deux cas de construction, un pour chaque évènement d'horloge considéré. Afin de tester notre implantation de la sémantique des SITPNs, un interprète a été conçu, i.e un programme qui simule les changements d'état d'un SITPN pour n cycles d'horloge, en partant de l'état initial du réseau. Cet interprète est prouvé correct et complet vis-à-vis de la sémantique des SITPNs pour une évolution sur un cycle d'horloge. L'intégralité de la formalisation et de la mécanisation est mise à disposition du lecteur[4]. Cependant, nous avons utilisé une autre version de l'implantation des SITPNs en Coq pour effectuer la preuve de préservation sémantique. La dernière version est plus élégante et utilisent les types dépendants[5].

0.3 Un langage cible : VHDL

Il existe plusieurs techniques permettant la synthèse physique d'un RdP. Cependant, la technique la plus étudiée est la transformation vers la langage VHDL. Cette technique a donc

---

[4] https://github.com/viampietro/sitpns
[5] https://github.com/viampietro/ver-hilecop/tree/master/sitpn/dp

été retenue par la méthodologie HILECOP. Le langage VHDL permet les descriptions structurelle et comportementale de circuits électroniques, à des fins de simulation ou de synthèse physique. En VHDL, un *design* décrit un composant électronique en termes d'interface entrée-sortie (l'*entité*) et de comportement interne (l'*architecture*). Le comportement d'un design s'exprime de deux manières : via l'interconnexion d'instances d'autres designs (des sous-composants), ou à l'aide de *processus*. La spécificité du langage VHDL tient à l'exécution concurrente des processus et des sous-composants décrivant une architecture de design. Un processus définit un bloc d'instructions séquentielles; il observe un certain nombre de signaux qui composent sa liste de sensibilité. Le changement d'état d'un signal de cette liste entraîne l'exécution du bloc d'instructions du processus. Conceptuellement, un signal VHDL représente une connexion physique sur un circuit électronique. Les signaux sont les principaux véhicules des valeurs dans les programmes VHDL.

La sémantique de VHDL est décrite dans une prose informelle dans le manuel de référence du langage (MRL). De fait, interpréter un programme VHDL, qui décrit un *design* de circuit, revient à simuler le design décrit. Dans le MRL, la sémantique de VHDL est donc définie sous la forme d'une boucle de simulation. La boucle de simulation spécifie la dynamique d'exécution des blocs concurrents qui composent une architecture de design, ainsi que la propagation des valeurs au travers des signaux.

La littérature propose de nombreuses formalisations de la sémantique de VHDL [69]. Certaines formalisations expriment la boucle de simulation telle qu'exhibée dans le MRL; d'autres choisissent de s'abstraire de cette boucle, et optent pour une formalisation alternative basée sur des modèles permettant la gestion de la concurrence et du temps (automates temporels, réseaux de Petri, logique d'intervalles temporels...).

La méthodologie HILECOP opère la génération d'un design VHDL dans l'optique de sa synthèse physique. Dès lors, nous ne considérons qu'une partie *synthétisable* du langage que nous définissons et nommons $\mathcal{H}$-VHDL. De plus, les designs VHDL générés par la méthodologie HILECOP décrivent des circuits synchrones, i.e, dont l'exécution est rythmée par un signal d'horloge. La prise en compte d'une sous-partie synthétisable et du synchronisme nous a permis d'exprimer la sémantique des programmes $\mathcal{H}$-VHDL en termes d'une boucle de simulation bien plus simple en comparaison de celle exprimée dans le MRL. L'Algorithme 1 décrit notre boucle spécifique de simulation pour un design $\mathcal{H}$-VHDL.

Algorithm 1: Simulation(Δ, σ_e, cs, E_p, T_c)

    // Initialization phase.
1 $\sigma'_e \leftarrow$ RunAllOnce(Δ,σ_e,cs)
2 $\sigma \leftarrow$ Stabilize(Δ,σ'_e,cs)

    // Main loop.
3 $\theta \leftarrow [\sigma]$
4 **while** $T_c > 0$ **do**
5 $\sigma_i \leftarrow$ Inject(Δ,σ,E_p,T_c)
6 $\sigma_\uparrow \leftarrow$ RisingEdge(Δ,σ_i,cs)
7 $\sigma' \leftarrow$ Stabilize(Δ,$\sigma_\uparrow$,cs)
8 $\sigma_\downarrow \leftarrow$ FallingEdge(Δ,σ',cs)
9 $\sigma \leftarrow$ Stabilize(Δ,$\sigma_\downarrow$,cs)

10 $\theta \leftarrow \theta \mathbin{+\!\!+} [\sigma',\sigma]$
11 $T_c \leftarrow T_c - 1$

12 **return** θ

L'Algorithme 1 est paramétré par un design élaboré Δ et un état de design σ_e. Ces deux paramètres sont le résultat de l'élaboration du design qui va être simulé. Le paramètre cs correspond au *comportement*, ou, pour être précis, à la partie comportementale de l'architecture du design. C'est ce comportement qui sera exécuté au cours de la simulation. Le paramètre E_p est l'environnement de simulation. Il permet l'injection de nouvelles valeurs sur les ports d'entrée du design à chaque nouveau cycle d'horloge. Le paramètre T_c correspond au nombre de cycles de simulation à effectuer, c'est à dire, le *front* de simulation.

La première partie de l'Algorithme 1 correspond à la phase d'initialisation. Chaque processus et sous-composants appartenant à cs sont exécutés exactement une fois lors de cette phase (RunAllOnce(Δ,σ_e,cs)). S'ensuit une phase de stabilisation des signaux (Stabilize(Δ,σ_e,cs)) où seules les parties combinatoires du design sont exécutées. Ensuite, vient l'exécution de la boucle principale de simulation. La boucle principale exécute T_c fois les phases d'un cycle d'horloge. Dans l'ordre, ces phases sont : (1) injection de nouvelles valeurs dans les ports d'entrée du design simulé, (2) exécution des processus séquentiels qui réagissent au front montant de l'horloge, (3) stabilisation des signaux, (4) exécution des processus séquentiels qui réagissent au front descendant de l'horloge, (5) stabilisation des signaux. Pour un cycle d'horloge, l'état stable obtenu au milieu du cycle et à la fin du cycle sont ajoutés à la trace de simulation θ. Cette trace de simulation est retournée à la fin de l'Algorithme 1.

Une formalisation de la sémantique de $\mathcal{H}$-VHDL a été effectuée sous la forme d'une sémantique opérationnelle à petit pas pour la partie simulation, c.-à-d., chaque état intermédiaire est considéré dans la trace de simulation. Le corps des processus est lui interprété avec une sémantique à grands pas. La mécanisation en Coq de la syntaxe et la sémantique de $\mathcal{H}$-VHDL a été réalisée. Cette sémantique s'inspire des travaux de formalisation esquissés dans [109, 17]. La sémantique formalisée prend également en compte la phase d'élaboration du design, préliminaire à la simulation. L'élaboration génère l'environnement de simulation, i.e un couple Δ, σ_{init} qui se trouve en paramètre de la boucle de simulation (voir Algorithme 1). Durant la phase d'élaboration, une vérification de type est effectué sur le code VHDL. La vérification de type

s'assure que la partie déclarative et la partie comportementale du design VHDL respectent certaines règles de typage définies par le MRL. Par exemple, pour une instruction d'affectation de valeur à un signal, l'expression affectée doit être du même type que le signal cible.

0.4 La transformation modèle-vers-texte de HILECOP

Comparée à la compilation de programmes (qui est un type de transformation), l'originalité de la transformation modèle-vers-texte de HILECOP provient de plusieurs critères. Premièrement, la représentation source de la transformation HILECOP n'est pas un programme écrit dans un langage de programmation. C'est un formalisme graphique, c.-à-d., celui des RdPs. La structure des modèles d'entrée est alors bien différente de celle de l'arbre syntaxique d'un programme. Par conséquent, l'expression de la transformation ne peut pas suivre la définition récursive opérant une descente dans l'arbre syntaxique du programme d'entrée, définition qui est usuelle pour les compilateurs de langage de programmation. Deuxièmement, le langage cible de la transformation HILECOP est un langage de description d'architecture de circuits, c.-à-d., VHDL. Même spécifique, ce langage reste un langage de programmation.

Nous allons illustrer la transformation modèle-vers-texte HILECOP en prenant le SITPN présenté en Figure 5 comme modèle d'entrée.
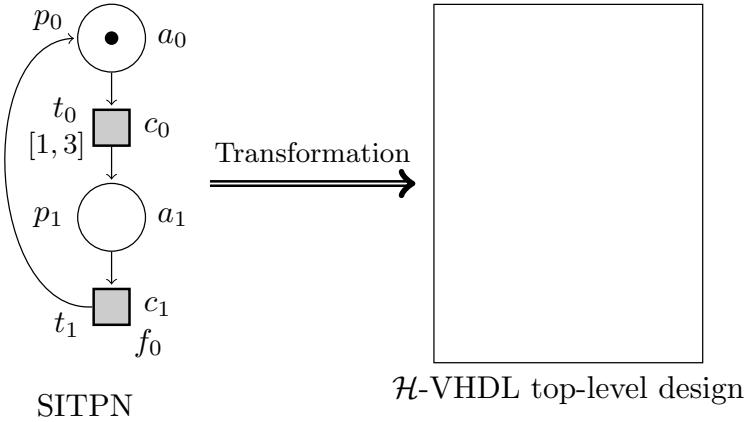
FIGURE 5: Transformation d'un modèle d'entrée SITPN en un design de top-niveau $\mathcal{H}$-VHDL. Le modèle d'entrée est composé de deux places p_0 et p_1, deux transitions t_0 et t_1. La transition t_0 est associée à l'intervalle de temps $[1, 3]$ et à la condition c_0. La transition t_1 est associée à la condition c_1, et son tir déclenche l'exécution de la fonction f_0. L'action a_0 est activé lorsque la place p_0 est marqué, et de même pour l'action a_1 et la place p_1.

La transformation modèle-vers-texte HILECOP génère un design $\mathcal{H}$-VHDL dit de *top-niveau* (c.-à-d., un circuit qui n'est pas lui-même embarqué dans un autre circuit) depuis un modèle d'entrée SITPN. La Figure 6 présente la forme finale du design $\mathcal{H}$-VHDL de top-niveau résultant de la transformation.
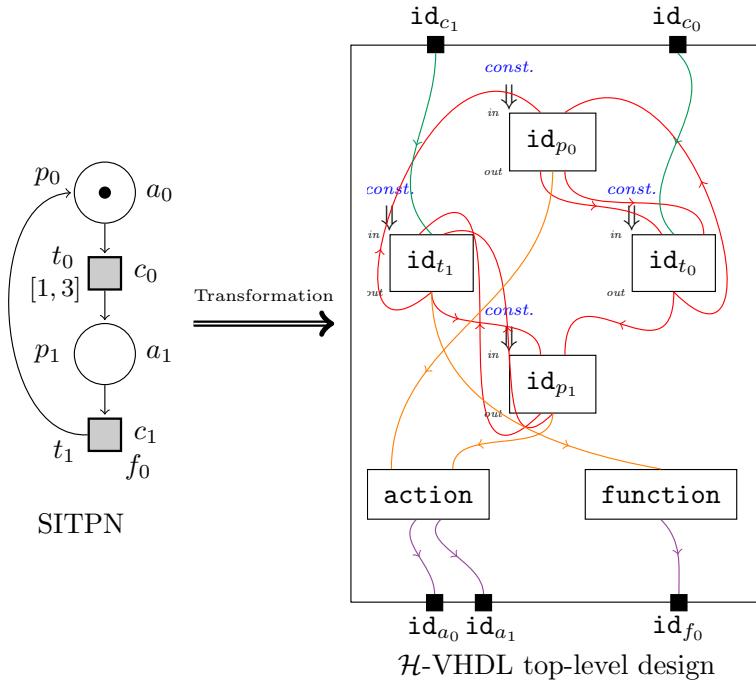
FIGURE 6: Design $\mathcal{H}$-VHDL de top-niveau résultant de la transformation HILECOP.

La première partie de la transformation HILECOP génère les composants qui vont constituer l'architecture interne du design de top-niveau. Pour chaque place du modèle d'entrée, un composant de *type* place, qui correspond à une instance du design place défini au préalable, est généré. Il en va de même pour chaque transition du modèle d'entrée. Dans la Figure 6, la place p_0 donne lieu au composant place d'identifiant id_{p_0}, la transition t_0 au composant transition d'identifiant id_{t_0}, etc. Lors de cette première phase, les parties *constantes* des composants sont générées (en bleue sur la Figure 6). Les parties constantes comprennent les constantes génériques, qui donne les dimensions aux interfaces des composants, et les informations liées aux arcs du SITPN d'entrée (c.-à-d., poids et types) qui sont encodées dans l'interface des composants de type place.

Lors de la deuxième phase de la transformation, les interconnexions entre composants de type place et composants de type transition sont générées. Les interconnexions apparaissent en rouge dans la Figure 6. C'est grâce à ces interconnexions et aux comportements internes de chaque composant que la même sémantique d'exécution du SITPN d'entrée sera obtenue dans sa version VHDL.

La dernière phase de la transformation concerne les éléments d'interprétation contenus dans le modèle d'entrée. Pour chaque condition du modèle d'entrée, un port d'entrée *primaire* (c.-à-d., un port d'entrée d'un design de top-niveau) est généré. Ce port d'entrée est connecté à l'interface de certains composants de type transition (fils verts dans la Figure 6). Cela représente l'association de la condition à certaines transitions du SITPN. Pour chaque action et fonction du modèle d'entrée, un port de sortie correspondant est généré dans l'interface de sortie du design de top-niveau. Ces ports de sortie représentent l'état d'activation/exécution des actions/fonctions associées. Pour qu'une action soit activée, il faut qu'au moins une des

places à laquelle l'action est associée soit marquée d'un jeton. Pour représenter ce mécanisme en VHDL, les composants de type place possèdent un port de sortie marked qui indique leur état de marquage. Lors de la transformation, tous les ports de sortie marked des composants de type place sont branchés au processus action, qui est aussi généré par la transformation. Le processus action est alors chargé d'activer les ports de sortie représentant les actions du modèle d'entrée. Le même mécanisme est mis en place pour les fonctions. Chaque composant de type transition est armé d'un port de sortie fired qui indique leur état de tir. Rappelons que dans la sémantique des SITPNs, une fonction est exécutée lorsqu'une des transitions qui lui est associée est tirée. Lors de la transformation, chaque port fired est branché au processus function, qui est aussi généré par la transformation. Le processus function va se charger d'activer les ports de sortie représentant les fonctions du modèle d'entrée. Cette activation se fait selon la valeur des ports fired des composants de type transition. L'interconnexion entre les ports marked et le processus action, et les ports fired et le processus function est représentée par les fils orange dans la Figure 6.

Un algorithme complet de la transformation a été exprimé en pseudo-langage impératif. Ensuite, l'algorithme a été implanté par une fonction écrite en langage Coq.

0.5 Preuve de préservation sémantique

Le but de cette thèse a été de prouver que la transformation modèle-vers-texte de HILECOP préserve le comportement de ses modèles d'entrée. Plus précisément, pour un modèle d'entrée de la transformation, nous voulons prouver que le design de top-niveau $\mathcal{H}$-VHDL résultant se comporte de la même manière. Il est donc d'abord important de définir la relation nous permettant de comparer un état d'un SITPN avec un état d'un design de top-niveau $\mathcal{H}$-VHDL. Nous avons défini une relation de similarité entre l'état d'un SITPN et l'état d'un design $\mathcal{H}$-VHDL. C'est à travers cette relation de similarité que notre théorème de préservation de comportement pourra être exprimé. La relation de similarité relie les valeurs présentes dans l'état d'un SITPN aux valeurs de certains éléments, principalement les valeurs de signaux, présents dans l'état d'un design $\mathcal{H}$-VHDL. Pour un état s de SITPN et un état σ de design $\mathcal{H}$-VHDL, s et σ sont similaires si :

- Pour toute place p, le marquage de p est égal à la valeur du signal interne s_marking d'un composant de type place d'identifiant id_p (où p et id_p sont liés par la transformation).

- Pour toute transition t, la valeur du compteur de temps associé à t est égale à la valeur du signal interne s_time_counter d'un composant de type transition d'identifiant id_t (où t et id_t sont liés par la transformation).

- Pour toute transition t, la valeur de l'ordre de reset associé à une transition t est égale à la valeur du signal interne s_reinit_time_counter d'un composant de type transition d'identifiant id_t (où t et id_t sont liés par la transformation).

- Pour toute condition c, la valeur d'une condition c est égale à la valeur du port d'entrée id_c représentant la condition dans le design de top-niveau $\mathcal{H}$-VHDL.

– Pour toute action a, la valeur d'une action a est égale à la valeur du port de sortie id_a représentant l'action dans le design de top-niveau $\mathcal{H}$-VHDL.

– Pour toute fonction f, la valeur d'une fonction f est égale à la valeur du port de sortie id_f représentant la fonction dans le design de top-niveau $\mathcal{H}$-VHDL.

Notre théorème de préservation de comportement prend donc la forme suivante. Pour un modèle SITPN d'entrée et le design de top-niveau $\mathcal{H}$-VHDL résultant de la transformation, si le SITPN renvoie la trace d'exécution θ, et le design renvoie la trace de simulation θ' en s'exécutant pendant τ cycle d'horloges, alors chaque couple d'états, considéré dans les traces à un même instant temporel, vérifie la relation de similarité.

Pour prouver ce théorème, nous avons raisonné par induction sur la structure des traces d'exécution. Le lemme fondamental pour la preuve déclare qu'à états de départ similaires, un SITPN et un design $\mathcal{H}$-VHDL liés par la transformation, et qui s'exécutent pendant un cycle d'horloge, arrivent en fin de cycle à deux états similaires. La Figure 7 exprime graphiquement ce lemme.
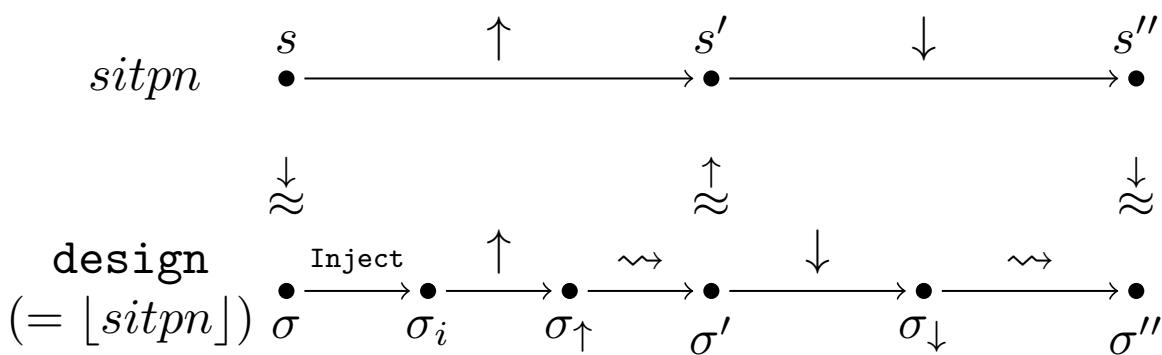
FIGURE 7: Représentation graphique du lemme déclarant que la transformation HILECOP préserve la sémantique des modèles initiaux pour une exécution sur un cycle d'horloge.

La partie supérieure de la Figure 7 représente l'exécution d'un cycle d'horloge pour un modèle SITPN ; la partie inférieure représente l'exécution d'un cycle d'horloge pour le design de top-niveau $\mathcal{H}$-VHDL résultant de la transformation HILECOP (i.e. `design` $= \lfloor sitpn \rfloor$ où le symbole de plancher représente la transformation). Il y a deux types de relation de similarité entre états, représentés par deux symboles différents. Le premier symbole $\overset{\downarrow}{\approx}$ représente la relation de similarité après front descendant ; le deuxième symbole $\overset{\uparrow}{\approx}$ représente la relation de similarité après front montant. Selon la phase du cycle d'horloge considérée, la relation de similarité varie quelque peu.

Nous avons prouvé que la transformation modèle-vers-texte HILECOP vérifie bien la propriété de préservation sémantique. La preuve a été effectuée informellement sur papier. Elle s'étale sur une centaine de pages. La mécanisation de la preuve avec l'assistant de preuves Coq est en cours de réalisation.

0.6   Conclusion

Le but de cette thèse a été de vérifier formellement une partie de la méthodologie HILECOP, usitée dans le cadre de la conception de circuits numériques critiques. Spécifiquement, le travail de vérification porte sur la phase transformant un modèle de conception, à base de RdPs, en *design* VHDL. Au final, nous avons fait la démonstration d'un théorème de préservation de comportement pour cette phase de transformation. La preuve a été effectuée informellement, mais représente tout de même un volume d'une centaine de pages. La mécanisation complète de cette preuve avec l'assistant de preuves Coq est en cours de réalisation.

Comme autres perspectives de travail, les modèles d'entrée de la transformation doivent inclure de nouveaux éléments. Deux éléments déjà existant dans ce formalisme restent à prendre en compte : les macroplaces, qui permettent d'exprimer la gestion d'exceptions dans les SITPNs, ainsi que la possibilité de spécifier des domaines d'horloge différents au sein d'un même modèle; c'est le cas des systèmes Globalement Asynchrones Localement Synchrones (GALS).

Le code Coq de la thèse, comprenant la formalisation des sémantiques des SITPNs et du langage $\mathcal{H}$-VHDL, l'implémentation de la transformation, l'expression du théorème de préservation sémantique ainsi qu'une partie de la preuve formelle, est accessible à l'adresse:

https://github.com/viampietro/ver-hilecop

Chapter 1

Introduction

With the use of every human-bred technology is associated a risk. Regarding the nature of the technology, and the broader system in which it is involved, the consequences of a failure can be dramatic. Thus arises the notion of the safety of systems; [20] gives the following definition of safety:

"Safety can [. . .] be defined as the freedom from exposure to danger, or the exemption from hurt, injury or loss."

A *safety-critical* system can be understood as a system for which the safety aspect is the main concern, being that important consequences, such as direct human losses, natural catastrophes, or economic disasters, could result of the failure of the system. In this thesis, conducted in the field of computer science, we are particularly interested in safety-critical *computer* systems. The concept of computer system encompasses both the low-level hardware-related and the more abstract software-related aspects involved in computer technologies. These days, computers pervade a considerable number of objects and technologies that pave our every-day life, including safety-critical systems. Thus, the risk associated with the use of computers in certain critical applications is real. Failures of safety-critical computer systems have happened and continue to happen; the list of critical incidents maintained by the ACM Committee on Computers and Public Policy and Peter G. Neumann ever since the mid 80s [85] is always growing[1].

To ensure the safety of computer systems involved in critical domains such as avionics, railway, power plants or medicine, there exists a number of standards and norms developed by international organizations. These standards set of a number of rules and techniques to be followed for the design, the production and the validation of safety-critical computer systems. To cite some well-known standards, the EUROCAE and RTCA organisms has devised the ED-12C/DO-178C and ED-80/DO-254 industry standards for the development cycle of software and hardware computer systems involved in avionics; the CENELEC has defined the EN-50128 standard for the development of software programs for railway control and protection systems; the IEC is at the origin of the IEC-60880 standard for the development of software programs involved in the control of power plants; the USA, Canada and the EU have defined the Common Criteria (CC) referential for the evaluation of the safety of systems and software programs. In this thesis, we are interested in verifying a *computer-related* methodology involved in the

---

[1]The *risks digest* website continues to register the computer-related incidents that resulted or could result in important damages: https://catless.ncl.ac.uk/Risks/

production of safety-critical medical devices. This domain is regulated by the EU 2017/145 standard[2] that sets the rules for the development of medical devices, including how to validate the technologies involved in the production line.

The rules imposed by the standards vary with respect to the criticality of the considered systems; for instance, in the medical field, the regulation text 2017/145 of the EU, pertaining to the marketing of medical devices, sets a different requirement level whether we are considering the production of dressings (class I), or of neuroprostheses (class III). The IEC defines a SIL (Safety Integrity Level) measure that qualifies the criticality level of a system. The CC defines a level of Evaluation Assurance Level (EAL), from 1 to 7, that must be met by the evaluated systems regarding their functional requirements.

Among the mandatory procedures, prescribed by the standards, which must be followed to validate a computer system involved in a safety-critical system, there are tests (unit, functional or integration tests) or simulation (especially applied to hardware systems). However, in the case of the development of safety-critical computer systems, a particular kind of methods, called *formal methods* (FM), are also applied. In formal methods, a computer system is considered as a mathematical object [9]. As pointed out in [20], "formal methods address *correctness* issues", that is whether a system delivers the required service. The perks of formal methods are to set a formal mathematical framework around a computer system. This framework allows us to reason about the system and prove that the system meets some required properties. Thus, a "formal methods" framework for computer systems requires at least a formal requirement language, i.e. with a formal semantics, to express the properties that a given system must verify. The expression of these required properties is called the *specification* of the system. We can cite some specification formalism such as CCS [79], CSP [61], Petri nets [93] or TLA+ [70] to describe reactive systems (i.e. systems that continuously interact with an environment); hardware description languages such as VHDL [74] and Verilog [65] can also be considered as formal specification languages for hardware designs if provided with a formal semantics and embedded in a formal proof system (see for instance [16]). We can also cite specification languages such as VDM [10], the Z notation [2] on which is based the B language [1] (included in the broader B-method); but also, all the theorem provers and proof assistants that come with their own specification languages such as Isabelle/HOL [86], Coq [106], PVS [38], etc. A FM framework must also provide a formal proof system to reason about the formal specification of the system. Some FM frameworks come with means to implement the computer system or simplified version of the system (i.e. a model) in a formal setting. This latter kind of framework enables to check if the implementation of a system always complies with its specification, i.e. the *soundness* of the system, and if all the aspects of the specification are met by the implementation, i.e. the *completeness* of the system.

Even though the purpose is always to check the correctness of systems, there exist multiple kinds of formal verification techniques. These techniques can be separated in three groups. The first group refers to the *deductive verification* methods; the techniques aims at establishing some proofs over a computer system in a formal proof setting; the deduction process can either be *interactive* (i.e. conducted by a human) or automated. The second group refers to model-checking techniques. The third group refers to the abstract interpretation of programs.

---

[2] http://data.europa.eu/eli/reg/2017/745/2020-04-24

The techniques applied to the formal verification of hardware computer systems or software computer systems are quite similar. Thus, most of the techniques presented here apply to both hardware and software developments. Note that the following presentation of formal verification techniques is not exhaustive.

Abstract interpretation

Abstract interpretation [37] aims at performing automatic static analysis of programs by approximating the possible execution states of the program. To do so, the concrete domains of the program variables are related to more abstract domains notably through the use of a lattice structure. Afterwards, invariant properties that the program verifies can be automatically checked against the lattice structure. For instance, abstract interpretation can help to determine that a given program terminates. To give examples of *abstract-interpretation-based* tools, we can cite the TERMINATOR static analyzer [35] that proves termination and other properties over C programs, or the Astrée [12] program analyzer for real-time embedded systems.

Similarly to abstract interpretation, symbolic execution [21] is another method for the static analysis of programs based on the partial execution of the considered programs. The method consists in generating all the execution traces, also called *symbolic* traces, of a given program; the result takes the form of a symbolic execution tree. In these traces, some of the program inputs, i.e. the *variables*, will be associated with *symbolic* expressions denoting the fact that the values of these inputs are yet unknown. Thus, by reasoning on the definition domains of these inputs, some properties of the program can be checked at execution points, that is, at the nodes of the symbolic execution tree. The property checking process is most of the time performed by a constraint solver [3], where constraints are expressed over the *symbolic* variables of the program. Pertaining to the construction of the symbolic execution tree, it is obtained most of the time by applying the rules of a structural formal semantics [94] associated with the language of the considered program. Typically, a branching in the execution tree is the result of the evaluation of a conditional statement. Each path of the execution tree is associated with a satisfiability condition, i.e. a Boolean formula, that determines if a given execution point is reachable or not. Symbolic execution methods are often used to generate test suites with an important coverage of the execution paths of programs. To give examples of *symbolic-execution-based* tools, we can cite the DART [53] or CUTE [101] test generators.

Model checking

Model-checking techniques [96, 32] build a model reflecting the execution of a computer system by enumerating all the possible execution states of the system. Then, the *execution* model can be automatically checked against some properties that the computer system must verify. The execution model must be a finite-state model, i.e. the enumeration of the execution states of the model must not be infinite. Most of the time, the properties that the computer system must verify are expressed through formulas of a modal logic. Model-checking techniques are broadly used for the formal verification of reactive systems, especially hardware systems. In that case, the properties that must be met by the considered system are expressed within formulas of one of the many temporal logics (Interval Temporal Logic [83], Linear Temporal Logic

[95], Computation Tree Logic [47], etc.), which are handy to express time-related properties. To give examples of well-known model-checkers, we can cite the BLAST [59], CADP [52] or Up-paal [6] model-checkers.

Deductive software verification

"Deductive software verification aims at formally verifying that all possible behaviors of a given program satisfy formally defined, possibly complex properties, where the verification process is based on some form of logical inference, i.e., 'deduction' " [57]. Deductive software verification methods are divided into two categories: interactive theorem proving methods and automated theorem proving methods.

In the philosophy of Interactive Theorem Proving (ITP), the programmer is responsible for the specification and the implementation of a computer program, but he also expresses theorems and conducts the corresponding proofs in a formal proof system. Interactive theorem proving methods are closely tied to proof assistants (cf. Isabelle/HOL, Coq, PVS, etc.), which offer the possibility to specify, implement, perform proofs over a given program in the same framework. The programmer builds the proof for a given theorem in an interactive manner, for instance assisted by a *tactic* language in the case of the Coq proof assistant (see Chapter 2 for an example). Each proof assistant comes with its own specification language and underlying proof system. In between the world of interactive theorem proving and automatic theorem proving, we can also cite the *contract-based* verification methods based on the Floyd-Hoare logic [48] and Dijkstra's weakest precondition calculus [44] such as SPARK [27], ESC [99], the B-method [1], Frama-C [68], or the Escher Verifier [26].

The Automated Theorem Proving (ATP) methods aim at automating the deduction steps involve in the proof search. Multiple automated theorem provers exist based on different proof search techniques such as natural deduction (e.g. Isabelle [86]), the tableaux method (e.g. the FaCT++ reasoner [108], Zenon [14]), or resolution algorithms (e.g. the E theorem prover [100], Vampire [98]).

In this thesis, we address the problem of the formal verification of a particular program. This program transforms an input model, which is an instance of a particular kind of Petri nets (PNs), into a program written in a Hardware Description Language (HDL). The program, the context in which it is involved, the specificities of the input model, and the target HDL, will all thoroughly be presented in this thesis. Here, we want to zoom in on the nature of the considered program that we aim to formally verified. The transformation from an instance of one formalism to another instance of another formalism is analog to the case of a *compiler* program. The only difference is that here an input to the transformation is not a program of a source *language*, but rather a model of an abstract source formalism, namely a PN model. Thus, our formal verification task amounts to the formal *verification* of a compiler program. The problem of compiler verification has greatly stimulated the use of formal methods in the field of software verification. Because a complete computer system is made out of complete chain of hardware, firmware and software components, the ultimate goal of the verification of such a system is to be able to prove the safety of all the layer composing it. In this system of layers, the place of compiler programs are mandatory as they are placed at the layer interfaces.

Indeed, one can prove that a given program and a given hardware is safe, but what if the compilation phase from the given program to low-level version introduces errors and behavior divergences. With these considerations in mind, compiler verification is an important aspect for one that needs to certify an entire computer system.

Thus, certifying a compiler program amounts to proving that the compiler verifies certain properties; [90] presents three of them. First, one can verify that a compiler is *type-preserving*, also called the *subject reduction* property. A type-preserving compiler yields a well-typed target program given a well-typed source program. Second, one can verify that a compiler is *semantic-preserving*. A semantic-preserving compiler yields a target program that behaves similarly to the source program. Thirdly, one can verify that the compiler is *equivalence-preserving*. Given two source programs that verify a certain *source-level* equivalence relation, an equivalence-preserving compiler yields two corresponding target programs that verify a certain *target-level* equivalence relation; this target-level equivalence relation is of course somehow related to the source-level equivalence relation. In this thesis, we are interested in proving that a *compiler-like* transformation program is *semantic-preserving*.

[72] lists several techniques that exist to establish that a compiler is semantic-preserving.

The first method is simply called *compiler verification*. Compiler verification aims at establishing the semantic-preserving property of a compiler program by proving a so-called semantic preservation theorem of the form:

For all source program S, and compiler C from the language of S to a target language, S has the behavior B (written $S \Downarrow B$) iff $C(S)$ (i.e. the compiled version of S) has the behavior B:
$$\forall S, C, B, S \Downarrow B \Leftrightarrow C(S) \Downarrow B.$$

Now the above form of the theorem is the strongest one, i.e. it can be proved only for a very particular kind of source and target languages. Other refined versions of this semantic preservation theorem exist depending on the nature of the source and target languages. Proving such a theorem is often performed with the help of a proof assistant as can be witnessed in the pioneering work on the CompCert compiler [73]; thus, compiler verification falls under the hood of deductive verification methods.

The second method is called compiler *validation*. Compiler validation does not aim at proving a theorem stating that a given compiler program is semantic preserving for all input programs. The strategy of compiler validation is to equip the compiler program with a *validator* program. Each time the compiler produces a target program from a source program, the validator tries afterwards to prove that the two programs have the same behavior. To establish such a proof, the validator program often relies on model-checking or abstract interpretation techniques.

The third method is called *proof-carrying* compilation. In this setting, the compiler program generates alongside the target program a proof that this program conforms to some property. The generated proof must be is such a format that can be verified by a *proof-checker*, built in a proof assistant for instance.

Even though it is not considered as a formal method, compiler testing is also a way to validate the semantic preservation property for a given compiler program. Considering the essential part played by compiler programs regarding the production of software products, a lot of efforts has been dedicated for the generation of test suites [29].

In the thesis, we will follow the *compiler verification* technique. Consequently, our aim is to prove a semantic preservation theorem over a transformation program and mechanize the process within the framework of the Coq proof assistant.

1.1   The HILECOP methodology

In this section, we present in more details the context of our work, and more specifically, the subject of our verification task, i.e. HILECOP, a methodology for the design and the production of safety-critical digital systems.

1.1.1   Designing safety-critical digital systems

According to Moore's law [80], the complexity of digital integrated circuits is always increasing. To give an example, the cut-of-the-edge *AMD Epyc Rome* microprocessor (2019) is made out of 50 billion of transistors. Composing billions of transistors on a wired circuit is no more a task for humans but is very suited to computers. However, engineers need to think about the design of digital circuits in a way that is understandable for humans. Therefore, they need high-level views of the circuits they are designing in order to work together and to communicate about the designs. The domain of Model-Based Systems Engineering (MBSE) [75] proposes a framework to help engineers to design and produce digital circuits, in a well-documented, safe and reliable way. Comparable to what Model Driven Engineering (MDE) does in the world of software engineering, models are first order concepts in MBSE. A model represents a simplified view of real object. As illustrated in Figure 1.1, a MBSE process describes a way to design a digital circuit starting from a high-level view of the system. This high-level view can follow a graphical formalism such as SysML [49] or Petri nets [93], or a textual one such as SystemC [11] or VHDL [5]. Then, the MBSE process describes many refinement phases (the downward-going green arrows in Figure 1.1) during which the input model will be transformed; at each refinement phase, the model goes down in abstraction towards its final implementation as a hardware circuit. A refinement phase, which is also a transformation phase, can be performed automatically, manually, or both. Depending on the refinement phase, the full automation of the transformation can sometimes never be achieved. In that case, a manual intervention is necessary.
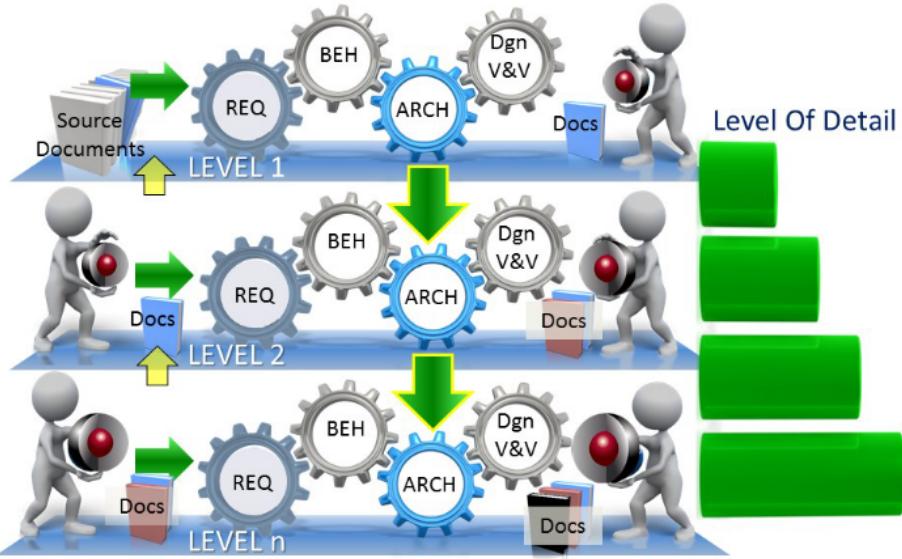
FIGURE 1.1: A Model-Based Systems Engineering process. Level 1 represents the highest abstraction level while Level n represents the concrete implementation of the system. REQ stands for requirements, BEH for behavior, ARCH for architecture, Dgn V&V for design verification and validation. This figure is an excerpt from [75].

In the case where the digital circuit being designed is a safety-critical system, an MBSE process will often employ formal models, i.e. models with a formal mathematical definition, as the design formalism. Thus, these models enable a certain extent of mathematical reasoning to prove that safety properties are met during the design V&V phase (cf. Figure 1.1).

The refinement process of the MBSE is really close to the one of the B-method [1] for the development of safety-critical software programs. The B-method allows the developers to specify, implement and verify a software program at an abstract level, using the B language; then, several refinement phases are performed until a concrete program is generated in the Ada or C language.

1.1.2 Introducing the HILECOP methodology

The INRIA CAMIN team has developed a new technology of neuroprostheses [55]. Neuroprostheses are medical devices which purpose is to electro-stimulate the nerves of patients suffering from moving disabilities. The nerves are responding to the stimulation, i.e. an electric influx, in order to activate the muscles and so that the patient can recover some movements. Thus, controlling stimulation applied to the patient's nerve is a critical point of the device overall functioning. This stimulation is generated and controlled by an implanted mixed circuit (resp. analogous and digital parts), embedded in the neuroprosthesis. Therefore, the design of such digital systems becomes utterly critical as a faulty circuit could result in the injury of patients. To assist the engineers in the design and the implementation of these safety-critical digital systems, the CAMIN team came up with a process called the "HILECOP methodology"

[4]. This methodology follows the principles of a MBSE process and relies on several trans-formations going from abstract models to concrete FPGA (Field-Programmable Gate Array) or ASIC (Application-Specific Integrated Circuit) implementations through the production of VHDL code. Figure 1.2 details the global workflow of HILECOP.
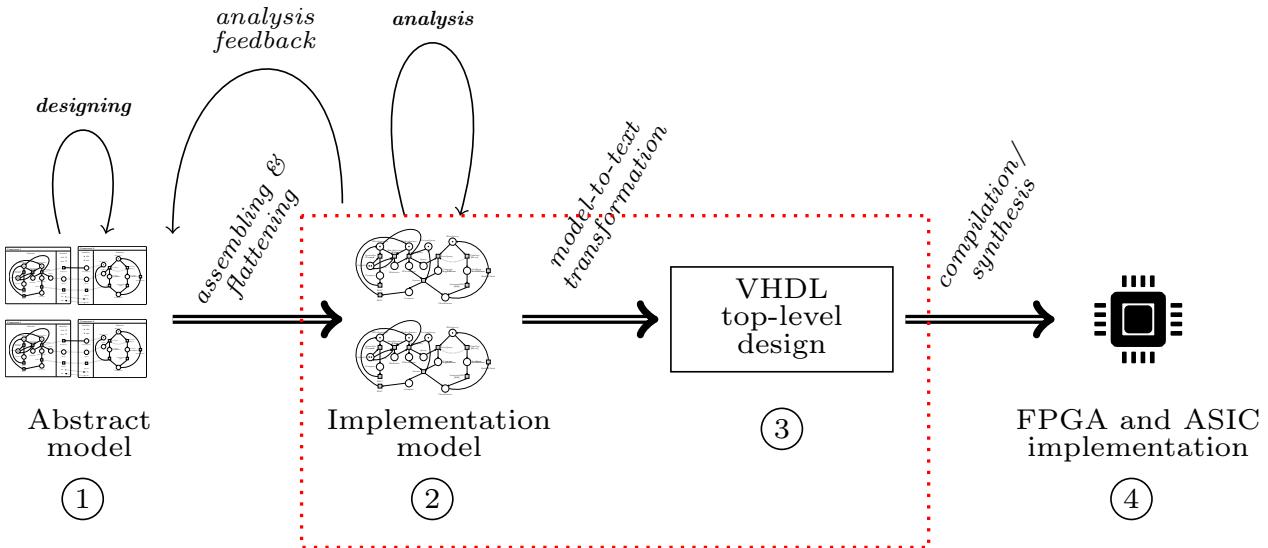
FIGURE 1.2: Workflow of the HILECOP methodology; horizontal double arrows in-dicate the transformation phases, i.e. the refinement phases in MBSE terms; simple arrows indicate different kinds of operations performed at a given step.

In Figure 1.2, Step 1 corresponds to the design phase of a digital system. At this step, the engineers produce a model of the required system; the leveraged model formalism is a graph-ical formalism specially designed for the methodology and based on component diagrams. Figure 1.3 provides an example of such a model.
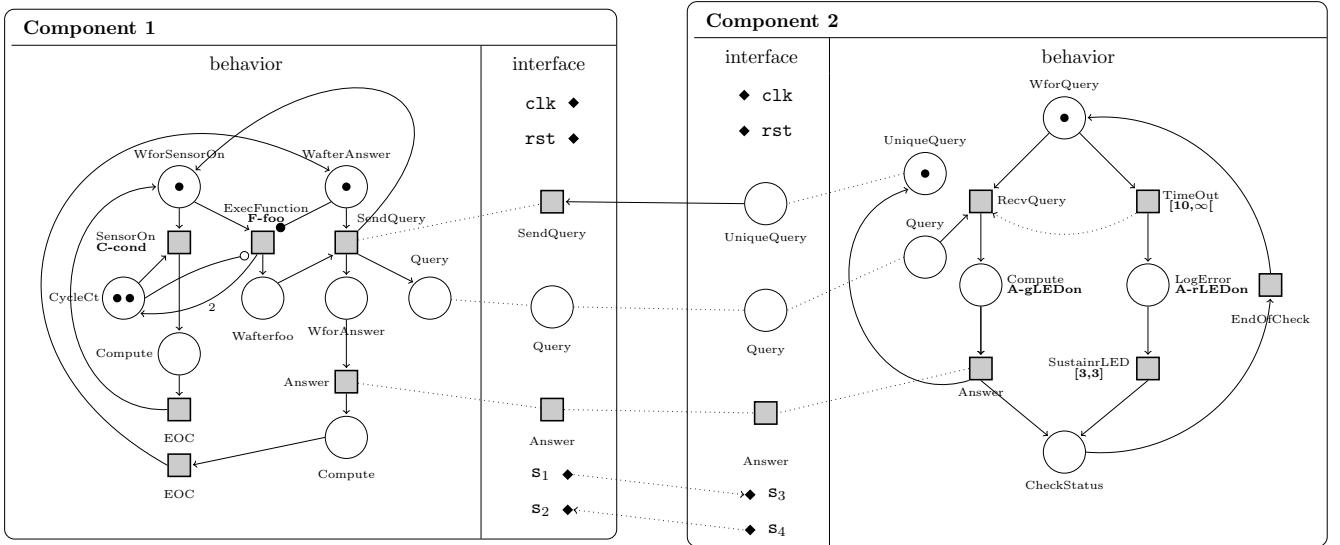
FIGURE 1.3: An Example of HILECOP high-level model. Black diamonds represent VHDL signals.

As shown in Figure 1.3, a component of the HILECOP high-level model formalism is represented by a box having an internal behavior and an interface that allows the connection to other components. The internal behavior of a component is defined with a specific kind of Petri Net (PN) model. These PNs and their distinguishing features will be thoroughly presented in Chapter 3. The component interface exposes references to the places, transitions, and signals of the internal behavior to the outside so that multiple components can be assembled. Each component has a clock and a reset input port (clk and rst) in its interface. The presence of the clk port shows that the HILECOP methodology has been built for the design of synchronous digital systems. To a certain extent, VHDL signals can be integrated to the high-level components to represent a direct wiring between components. A component behavior can also be defined through the composition of other components. In that case, we talk about a composite hardware structure.

Next, in Figure 1.2, the transformation from Step 1 to Step 2 flattens the model. The internal behaviors are connected according to the interface compositions, and embedding component structures are removed. Figure 1.4 gives the result of the flattening phase for the model of Figure 1.3. In Figure 1.4, we do not show the VHDL signals that were present in Figure 1.3. As these signals already constitute plain VHDL code, they will simply be copied as they stand during the model-to-text transformation happening from Step 2 to Step 3.
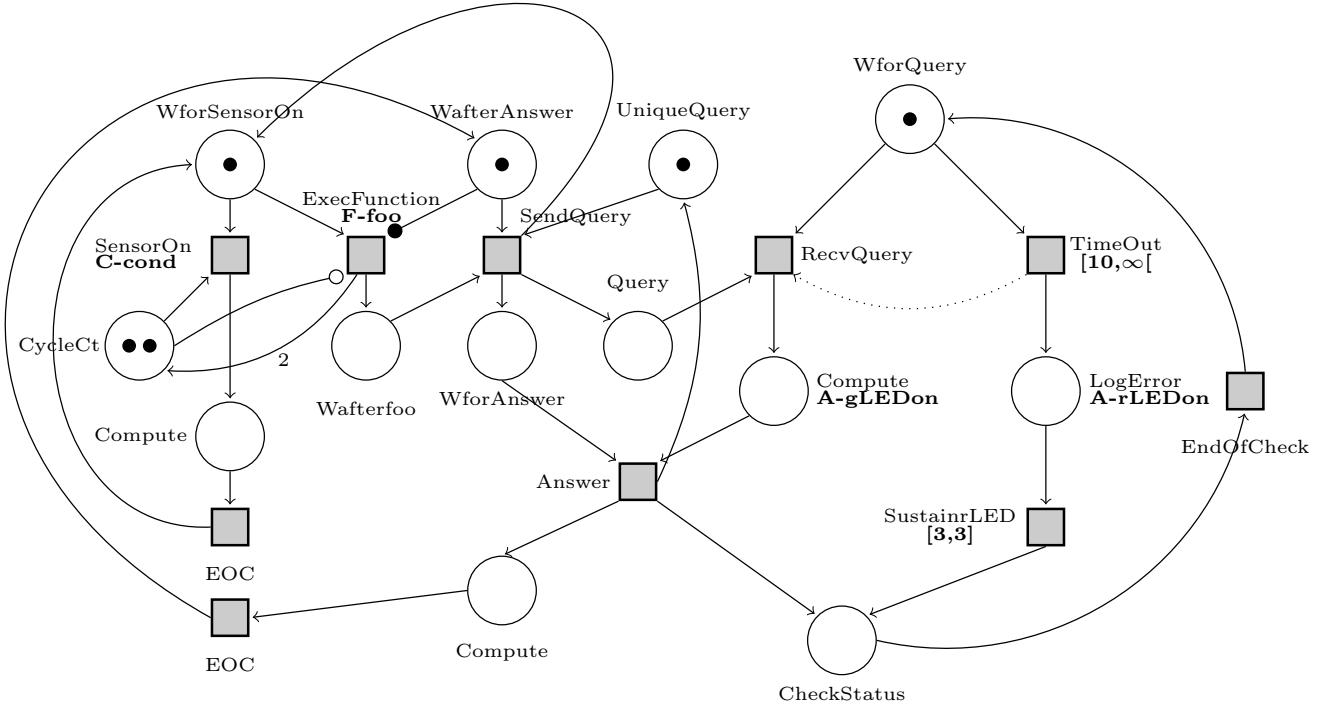
FIGURE 1.4: A global Petri net model obtained after the flattening of a HILECOP high level model.

The PN models used in HILECOP have been specifically devised for the design of safety-critical digital systems; a first thesis has formalized the execution semantics of these PN models [71]. What makes them a very particular kind of models is their *synchronous* execution semantics. This semantics denotes from the standard *asynchronous* execution of PNs. The PN formalism is a formal model and therefore allows us to apply mathematical reasoning on its instances. Particularly, a PN model can be analyzed, and a proof that a given model meets some properties can be automatically produced through the direct analysis of the structure or through the use of model-checking techniques. This feature of PNs has been one of the reason of the adoption of this formalism as HILECOP's base formalism. A thesis has been dedicated to the development of new methods to analyze the HILECOP PN models [78]. In fact, the transformation of the abstract model is a bit different in preparation of the model analysis. The transformation adds new information to the flattened model to help the analysis. Figure 1.4 only gives the flattened version of the model produced in preparation of the next transformation into a VHDL design. The analysis phase is here to convince the engineers that they are indeed designing a safe system. The analysis process is a round trip between Step 1 and Step 2. It aims at producing a model that is conflict-free (see Section 3.2.6 for more details about the definition of a conflict), bounded, and deadlock-free, using model-checking techniques. After several iterations, the model should reach soundness and is then said to be *implementation-ready*.

In Figure 1.2, from Step 2 to Step 3, VHDL source code is then generated by means of an automatic model-to-text transformation. The generated code describes a VHDL design, i.e. a textual description of a hardware system, which has an interface defining input and output ports and an internal behavior called an architecture. Details about the syntax and the semantics of the

VHDL language will be given in Chapter 4. Figure 1.5 succinctly illustrates the transformation between Step 2 and Step 3. Each place (resp. transition) of the input PN model (on the left) is transformed into a *place* (resp. *transition*) component instance, which is an instance of the `place` (resp. `transition`) design. The transformation from Step 2 to Step 3 will be thoroughly presented in Chapter 5.
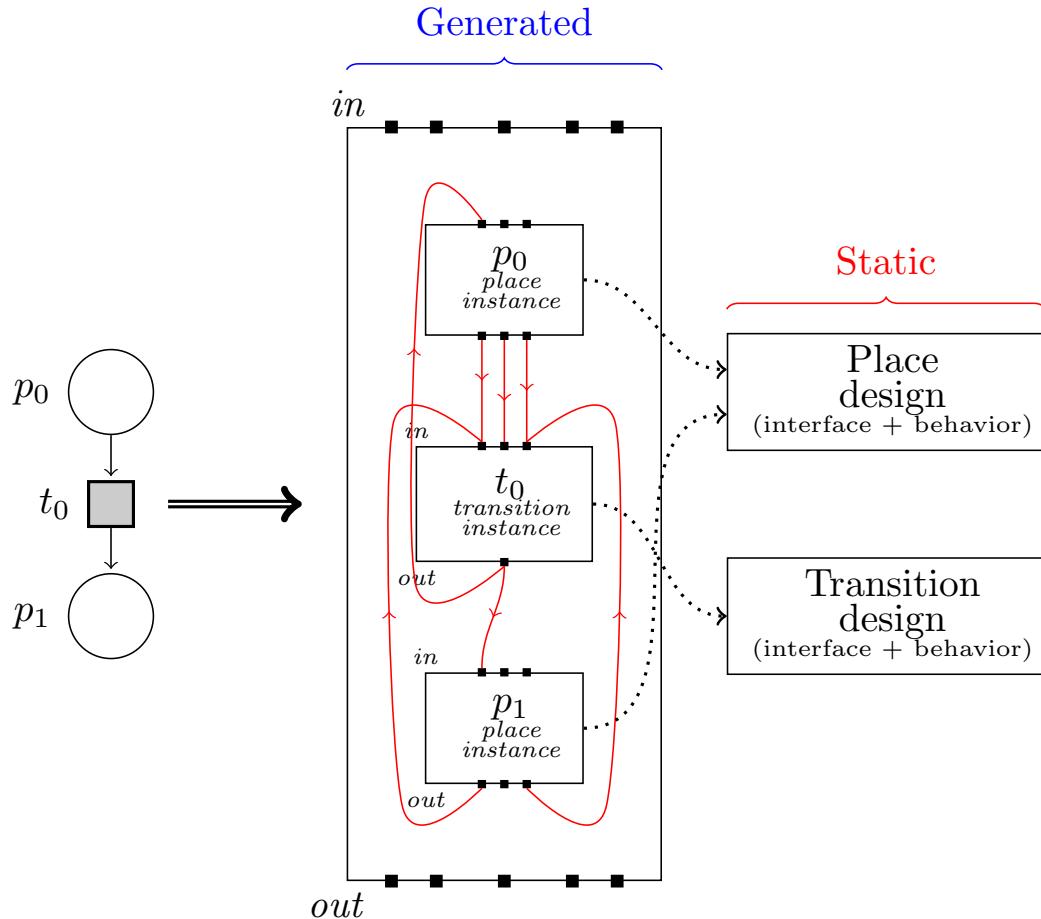
FIGURE 1.5: Generation of a top-level VHDL design from a Petri net. On the left, the input PN and on the right, the generated VHDL top-level design. Dotted arrows show the relation between the component instances and their source design.

For the purpose of the HILECOP methodology, two VHDL designs have been defined: the `place` design, which is a hardware description of a PN place (circle nodes in a PN) and the `transition` design, which is a hardware description of a PN transition (square nodes in a PN). Like all VHDL designs, the `place` and the `transition` designs have an input and output port interface, and their own internal behavior. A VHDL design describes a kind of "class" of hardware component. Thus, a design can be instantiated in the behavior of other designs in order to obtain more complex behaviors. As illustrated in Figure 1.5, the transformation from Step 2 to Step 3 creates a place component (or design) instance (PCI) and a transition component (or design) instance (TCI) for each place and transition of the input Petri net. Then,

the PCIs and TCIs are connected together through their input and output port interfaces. These connections reflect the arc connections, and thus the interactions, between the places and the transitions of the input PN model.

From Step 3 to Step 4, the VHDL compilation/synthesis and the FPGA programming, or ASIC realization, are finally performed using industrial tools. At the end of Step 4, the designed circuit is physically built on an FPGA device or an ASIC. What happens between Step 3 and Step 4 appears as a black box in the whole HILECOP methodology. Therefore, we will not consider this transformation phase, which will not be verified.

1.1.3   Verifying the HILECOP methodology

The use of Petri nets as a base model is one of the major advantage of the HILECOP methodology. All the analysis tools that accompany the Petri net formalism, and allow us to prove that the models meet some required properties, qualify the HILECOP methodology as a formal method for the design and implementation of safety-critical digital systems. However, even with input models that are proved to be *sound*, the advantages provided by the use Petri nets would be lost if one of the transformations performed during the process changes the input definition of the circuit in a way that would alter its behavior. Thus, the engineers would have specified a perfectly correct digital system but would never obtain the expected circuit on a physical device. Therefore, in order to reinforce the confidence in the HILECOP methodology, the goal of this thesis is to verify, by establishing a formal proof, that the model-to-text transformation from Step 2 to Step 3 (i.e. the framed part with red dotted lines in Figure 1.2) preserves the behavior of the input models into the generated VHDL designs. We choose to carry out this task as a deductive verification task. We aim at proving a theorem stating that the HILECOP model-to-text transformation is *semantic-preserving*. This theorem will be of the following form: for all PN model, input to the HILECOP transformation, the generated output VHDL design behaves similarly at execution time. Chapter 6 formally presents our behavior preservation theorem, and thus, what we mean about the similarity of execution between a PN model and a VHDL design.

One could argue that to qualify the entire HILECOP methodology, one has to verify all the transformations used in the methodology, i.e. consider also the transformation from Step 1 to Step 2, and the transformation from Step 3 to Step 4. However, we shall say that:

– The transformation from Step 1 to Step 2 changes the structure of the component-based input model. Even if the removal of the component structures induces some structural rearrangements, the behavior of the flattened model is almost similar to the one of the component-based model. Therefore, we argue that verifying that this transformation is semantic-preserving is an easy enough task.

– The transformation from Step 3 to Step 4 is performed by industrial tools. We rely on these tools because they are widely used in the industry for the development of safety-critical systems (e.g. cadence tools in aerospace and defense domains). Moreover, the compiler/synthesizer used at this stage of the methodology is a proprietary product. Thus, we don't have any access to the code of this program. Moreover, the compiler/synthesizer performs a lot of

optimizations over the input VHDL code. Even with a provided access to the code, verifying such an optimizing compiler would not possible within the time-span of this thesis.

Now that we have clarified the nature of the verification task we want to achieve, we can state our research question as follows:

CAN WE PROVE THAT THE MODEL-TO-TEXT TRANSFORMATION DESCRIBED IN THE HILECOP METHODOLOGY IS SEMANTIC PRESERVING?

This task is really close to the formal verification of compilers for programming languages. Compiler verification has been widely explored, and many works are accessible in the literature [39]. The major source of inspiration of this thesis has been the work done on the CompCert certified C compiler [72]. Thus, we argue here that the scientific interest of our research comes from the comparison between the methods used to perform our verification task and the methods used to perform similar verification task in other domains such as compiler verification. Thus, we can complement our research question with the following ones:

– What are the similarities and the differences between the HILECOP transformation and other transformation situations (compilers, model transformations...)?

– Is there a strategy to perform the verification of the HILECOP transformation?

– How far the correspondence holds between this strategy and the strategy used in other transformation situations such as compiler verification?

To achieve the formal verification of HILECOP, our approach is similar to what has been done for the CompCert compiler. The idea is to formalize the semantics of the source and target languages, and verify that the transformation preserves the semantics of any input model. In the thesis, we propose both to perform the formalization work on "paper" and mechanize it within the Coq proof assistant [8].

In the case of HILECOP, some specificities of the source and target languages introduce additional technical difficulties in the process of formal verification. A first difference pertains to HILECOP's high-level formalism (the input language), which is quite abstract. This formalism depends on PNs, and thus is not a common programming language.

A second difference is about the VHDL language (the output language). Similarly to the PN models used in HILECOP, the VHDL language is not a common programming language as its purpose is both the structural and behavioral description of hardware circuits.

To further motivate the necessity of the verification task, the development of neuroprostheses by the INRIA CAMIN team is at the base of the creation of the Neurrinov company[3]. The Neurrinov company is now looking towards the industrial development of such neuroprostheses. We hope that once the verification performed on the HILECOP methodology, it will help to obtain the CE certification, related to the EU 2017/745 regulation text, necessary to qualify the neuroprostheses as eligible for the medical market.

Moreover, the HILECOP methodology comes with a working implementation based on the Eclipse framework. This software is currently used by the engineers of the Neurinnov company to design the digital systems having a part in the neuroprostheses. Figure 1.6 gives a view of the existing HILECOP software.

---

[3] http://neurinnov.com/
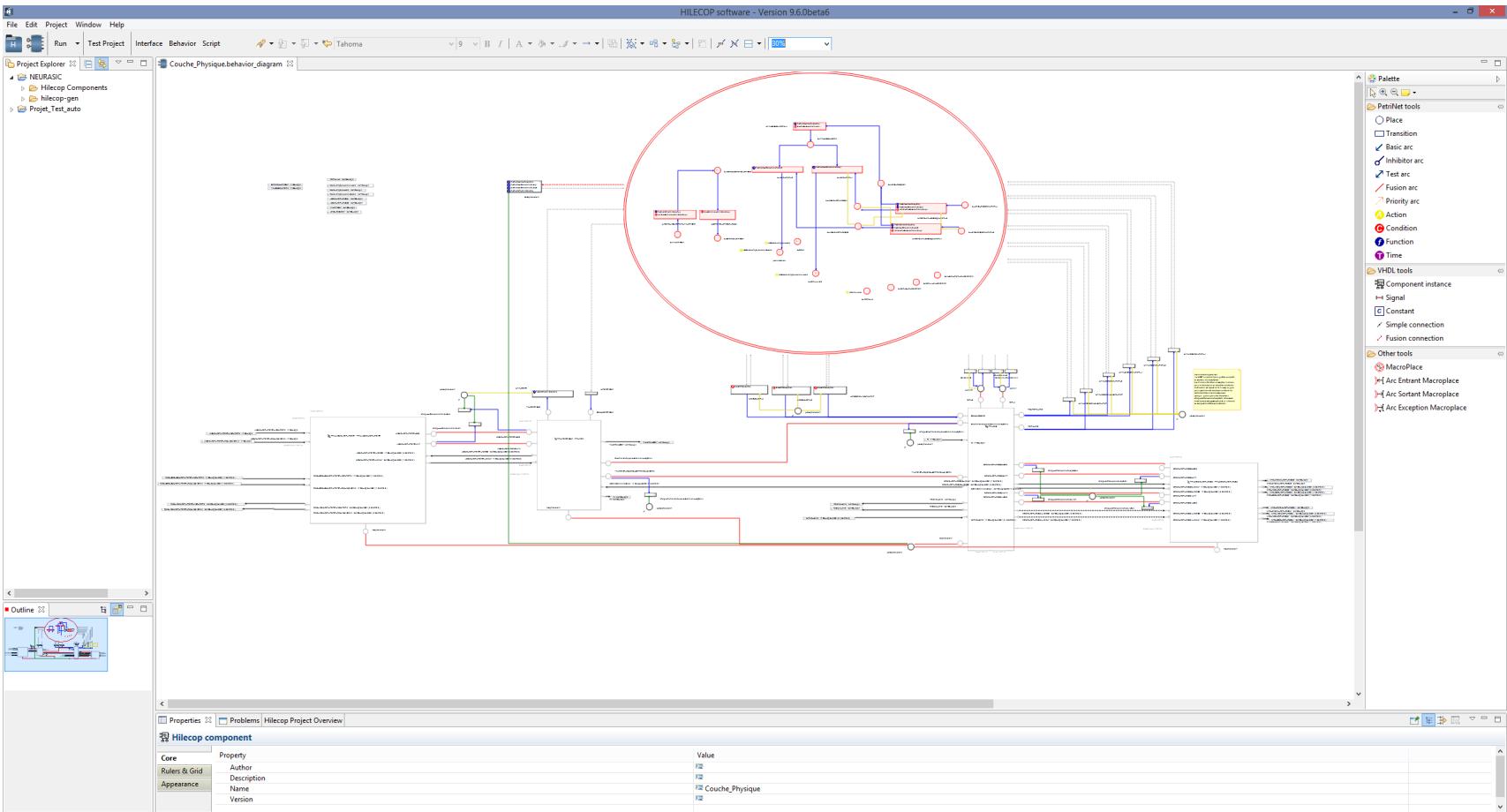
FIGURE 1.6: A view of the HILECOP software implemented on top of the Eclipse framework. The middle frame shows high-level model of digital system such as can be designed in the HILECOP methodology. On the right side, the frames correspond to the palette of tools available to the user to build a model of a digital system.

To the purpose of formal verification, we will implement the HILECOP model-to-text transformation leveraging the functional language of the Coq proof assistant. However, after the mechanization of the proof of semantic preservation, we could use the extraction feature of the Coq proof assistant to produce the implemented transformation as an OCaml program. Then, we will be able to connect this program to the existing HILECOP software in order to use the verified version of the transformation.

This thesis memoir is structured as follows.

Chapter 2 introduces all the necessary mathematical notions to understand the remainder of the memoir. Chapter 3 presents in an informal and formal way a specific kind of Petri net models; these models are the input to the HILECOP transformation program. Chapter 4 gives an informal presentation of the VHDL language. The VHDL language is the target language in which the programs generated by the HILECOP transformation are written. We also give in this chapter a formal definition of the syntax and semantics of a subset of the VHDL language that we call $\mathcal{H}$-VHDL. Chapter 5 presents the algorithm of the HILECOP transformation and its implementation with the Coq proof assistant. Chapter 6 details the semantic preservation theorem expressing that the HILECOP transformation is semantic-preserving. It also gives the high-level theorems and lemmas involved in the proof of the semantic preservation theorem. Finally, Chapter 7 ends the memoir, and outlines the perspectives regarding the full completion of the task of proving that the HILECOP transformation is semantic-preserving.

The results of a literature review pertaining to the formal semantics VHDL is presented at the beginning of Chapter 4. Similarly, the results of a literature review pertaining to the task of compiler verification in the world of deductive verification methods are presented at the beginning of Chapters 5 and 6.

Chapter 2

Preliminary notions

In this chapter, we introduce the mathematical formalisms and notations used throughout this thesis to express and formalize our ideas. Section 2.1 introduces both classical first-order logic and set theory which constitute our mathematical frameworks. Section 2.2 is a reminder on induction principles. In Section 2.3, we provide the basics to understand the Coq proof assistant, which is the framework we adopt to write our programs and mechanize our proofs. This chapter is inspired by the book *The Formal Semantics of Programming Languages: an Introduction* [112] by Glenn Winskel, the courses of the University of Cambridge on the semantics of programming languages[1] by Neel Krishnaswami, the documentation of the Coq proof assistant[2], and the book *Certified Programming with Dependent Types* [31] by Adam Chlipala.

2.1 Mathematical formalisms

In this section, we introduce classical first-order logic and the Zermelo-Fraenkel (ZF) set theory which combination constitutes the base formalism for all our mathematical definitions, and our framework for the expression and the interpretation of logical formulas.

2.1.1 Classical first-order logic

In this section, we define the syntax of the first-order logic. Here, we already use concepts and notations that belong to set theory; set theory will be presented in the following section. For now, the reader has only to consider the *intuitive* definition of a set as a collection of elements, and a function as an entity that relates each element of a set to a unique element of another set.

To define the syntax of the first-order logic, let us first define:

– The set $\mathcal{V}$ of variables x, y, etc.

– The set $\mathcal{S}_{\mathcal{F}}$ of function symbols f, g, etc.

– The set $\mathcal{S}_{\mathcal{P}}$ of predicate symbols P, Q, etc.

---

[1] https://www.cl.cam.ac.uk/teaching/2021/Semantics/
[2] https://coq.inria.fr/distrib/current/refman/index.html

Let us also consider a function $a \in S_{\mathcal{F}} \cup S_{\mathcal{P}} \rightarrow \mathbb{N}$ that associates a given function or predicate symbol to an arity, i.e. the number of parameters of the function or the predicate. E.g, if $f(x,y)$ with $f \in S_{\mathcal{F}}$ then $a(f) = 2$; if $P(x,y,z)$ with $P \in S_{\mathcal{P}}$ then $a(P) = 3$. Constants are functions of arity 0, i.e. with no parameter.

The syntax of classical first-order logic [81] is divided between *terms* and *formulas*. We define a term of the classical first-order logic with the following BNF entry:

$$t ::= v \mid f(t_1, \ldots, t_n)$$

This entry states that a term is either a variable $v \in \mathcal{V}$, or a function symbol $f \in S_{\mathcal{F}}$ of arity n with terms $t_1, \ldots, t_n$ as inputs.

We define a formula of the classical first-order logic with the following BNF entry:

$$\mathcal{F} ::= \bot \mid \top \mid P(t_1, \ldots, t_n) \mid \mathcal{F} \wedge \mathcal{F} \mid \mathcal{F} \vee \mathcal{F} \mid \mathcal{F} \Rightarrow \mathcal{F} \mid \mathcal{F} \Leftrightarrow \mathcal{F} \mid \neg \mathcal{F} \mid \forall x, \mathcal{F} \mid \exists x, \mathcal{F}$$

This entry states that a formula is either:

– $\bot$ (bottom), the always *false* formula, or $\top$ (top), the always *true* formula.

– A predicate $P(t_1, \ldots, t_n)$ (i.e. an atomic formula) of arity n, where $P \in S_{\mathcal{P}}$ with terms $t_1, \ldots, t_n$ as inputs.

– The composition of two subformulas with one of the following binary operators: the conjunction $\wedge$, the disjunction $\vee$, the implication $\Rightarrow$, the double implication $\Leftrightarrow$.

– The composition of one subformula with the negation operator $\neg$.

– A subformula prefixed by the universal quantifier $\forall$ or the existential quantifier $\exists$. For instance, the formula $(\forall x, P(x))$ denotes the atomic formula $P(x)$ where the parameter x is a universally quantified variable of the formula. As a shorthand notation, we write $\forall x, y, z. \ldots$ to denote $\forall x, \forall y, \forall z. \ldots$. The same stands for the existential quantifier $\exists$. Variables that are introduced in a logical formula by one of the previous quantifiers are called the *bound* variables of the formula. Variables that appear in a logical formula without being introduced by a quantifier are called *free* variables. For instance, in the logical formula $(\forall x, P(x) \wedge Q(y))$, x is a bound variable and y is a free variable of the formula.

In this thesis, our formulas are interpreted as formulas of the *classical* logic [102]. Thus, during a proof, we can appeal to the *law of excluded middle* to reason on the truth value of a given formula.

2.1.2   ZF Set theory

In this thesis, we use the Zermelo-Fraenkel (ZF) set theory as the base formalism for all our mathematical definitions and proofs. In this section, we present the axioms of the ZF set theory, and the associated definitions and notations that will be used throughout this memoir. The reader will find further information on the ZF set theory in [82].

In the ZF set theory, a set represents a group of elements called the members of the set. For every set A, we write $a \in A$ to denote that the element a is a member of set A. The *membership* property is a basic set-theoretic property. Given a set A, we sometimes have to express a property P of the elements of A in the following form: $\forall x, x \in A \Rightarrow P(x)$. When there is no ambiguity, we equivalently write $\forall x \in A, P(x)$.

Now, let us define the axioms of the ZF set theory.

Axiom 1 (Existence). *There exists a set which has no elements, i.e. $\exists A, \forall a, a \notin X$.*

Definition 1 (Empty set). *We call the empty set the unique set with no element, written $\varnothing$.*

Axiom 2 (Extensionality). *If every element of A is an element of B, and vice-versa, then $A = B$, i.e. $\forall A, B, x, (x \in A \Leftrightarrow x \in B) \Rightarrow A = B$.*

Axiom 3 (Schema of comprehension). *Let $P(x)$ be a property of x. For all set A, there exists a set B such that $x \in B$ if and only if $x \in A$ and $P(x)$ holds. I.e. $\forall A, B, x, x \in B \Leftrightarrow x \in A \wedge P(x)$.*

Axiom 4 (Pair). *For all set A and B, there exist a set C such that $x \in C$ if and only if $x = A$ or $x = B$. I.e. $\forall A, B, \exists C, \forall x, (x \in C \Leftrightarrow x = A \vee x = B)$.*

More intuitively, Axiom 4 states that if A and B are sets then their corresponding pair C is also a set.

Axiom 5 (Union). *For all set A having sets as elements, there exists a set C corresponding to the union of the elements of A. I.e. $\forall A, \exists U, x \in U \Leftrightarrow \exists B, B \in A \wedge x \in B$.*

Definition 2 (Subset). *A set A is a subset of set B, written $A \subseteq B$, if for all x if $x \in A$ then $x \in B$, i.e. $\forall x, x \in A \Rightarrow x \in B$.*

Definition 3 (Union). *Given a set A and B, the set $A \cup B$ is the union of the members of A and the members of B, i.e. $\forall x, x \in A \cup B \Leftrightarrow x \in A \vee x \in B$.*

Axiom 6 (Infinity). *There exists a set A such that the empty set belongs to A and for all x such that $x \in A$ then $x \cup \{x\} \in A$. I.e. $\exists A, \varnothing \in A \wedge (\forall x, x \in A \Rightarrow x \cup \{x\} \in A)$.*

Axiom 7 (Power set). *For all set A, there exists a set B such that for all set X, $X \in B$ if and only if $X \subseteq A$. I.e. $\forall A, \exists B, \forall X, (X \in B \Leftrightarrow X \subseteq A)$.*

Axiom 8 (Schema of replacement). *Let $P(x,y)$ be a property such that for all x there exists a unique y such that $P(x,y)$ holds. For all set A, there exists a set B such that for all $x \in A$, there exists $y \in B$ for which $P(x,y)$ holds. I.e. $(\forall x, \exists y, P(x,y)) \land \forall A, \exists B, \forall x, (x \in A \Leftrightarrow \exists y, y \in B \land P(x,y))$.*

Definition 4 (Intersection). *Given a set A and B, the set $A \cap B$ denotes the set formed by the intersection of set A and B, i.e. $\forall x, x \in A \cap B \Leftrightarrow x \in A \land x \in B$.*

Axiom 9 (Foundation). *For all non-empty set A, there exists a set B such that $B \in A$ and B has no common element with A. I.e. $\forall A, A \neq \varnothing \Rightarrow \exists B, B \in A \land (A \cap B = \varnothing)$.*

Based on the previous axioms, we can complement the theory with following definitions and notation.

Notation 1 (Extension). *A set is defined by* extension *with the enumeration of all its members. For instance, $\{1, 0, -1\}$, $\{a, b, c\}$ or $\{p_0, \ldots, p_n\}$ are all sets defined by extension.*

Notation 2 (Intension). *Let $P(x)$ be a property of x, we write $\{x \mid P(x)\}$ the set of x for which $P(x)$ holds. The set $\{x \mid P(x)\}$ is defined by intension. For instance, here is the intensional definition of the set of even numbers: $\{n \in \mathbb{N} \mid \exists k \in \mathbb{N}, n = 2k\}$.*

Given two sets A and B, the following sets are formed:

Definition 5 (Difference). *$A \setminus B$ denotes the set formed by the elements of set A that are not elements of set B (the difference between set A and B), i.e. $A \setminus B = \{x \mid x \in A \land x \notin B\}$.*

Definition 6 (Cartesian product). *$A \times B$ denotes the Cartesian product between the elements of set A and set B, i.e. the set of all ordered pairs defined by $\{(x,y) \mid x \in A \land y \in B\}$. We generalize the definition to build the set of n-tuples $A_0 \times A_1 \times \cdots \times A_n$ defined by $\{(x_0, (x_1, \ldots (\ldots, x_n))) \mid x_0 \in A_0, x_1 \in A_1, \ldots, x_n \in A_n\}$.*

It is sometimes useful to give a name to the elements of a tuple without referring to their index. In such a case, a tuple is called a record where each element, called a field, has been given an explicit name. This formalism is useful to represent rather complex data structures. For instance, say that we want to represent the set of humans by a triplet composed of the size, weight, and eye color of a given human. We can define this set as the set of triplet $\mathbb{R} \times \mathbb{R} \times \{green, blue, brown\}$. If we want to give a concrete name to the elements of the triplet, we can equivalently define such a triplet as a record, written $<size, weight, eye>$, where $size \in \mathbb{R}$, $weight \in \mathbb{R}$ and $eye \in \{green, blue, brown\}$.

> **Definition 7** (Disjoint union). *$A \sqcup B$ denotes the set formed by the disjoint union of set A and set B. The disjoint union is obtained by adjoining an index i to the elements of A and an index j to the elements of B such that $i \neq j$. Then, the two sets of couples are joined together to build the disjoint union of A and B. For instance, consider that $A = \{a, b, c\}$ and $B = \{a, b\}$. To obtain the disjoint union $A \sqcup B$, we create the two sets $A_i = \{(i, a), (i, b), (i, c)\}$ and $B_i = \{(j, a), (j, b)\}$, and then join the sets together s.t. $A \sqcup B = \{(i, a), (i, b), (i, c), (j, a), (j, b)\}$.*

When the two sets A and B are disjoint, i.e. $A \cap B = \varnothing$, then $A \sqcup B$ is isomorphic to $A \cup B$. To stress the fact that we are building a set from the union of two disjoint sets, we prefer to use the disjoint union operator. For instance, we write $\mathbb{N} \sqcup \{\infty\}$, instead of $\mathbb{N} \cup \{\infty\}$, to denote the set of values ranging from the set of natural numbers with the addition of the infinite value ∞.

> **Definition 8** (Powerset). *$\mathcal{P}(A)$ denotes the powerset of A defined by all the possible subsets formed with the elements of set A, i.e. $\mathcal{P}(A) = \{X \mid X \subseteq Y\}$.*

Relations and functions

> **Definition 9** (Relation). *A binary relation R between two sets X and Y is a subset of the set of pairs $X \times Y$, i.e. $R \subseteq X \times Y$, or an element of the powerset $\mathcal{P}(X \times Y)$, i.e. $R \in \mathcal{P}(X \times Y)$. We write $R(x, y)$ to denote $(x, y) \in R$. We generalize the definition to n-ary relations. An n-ary relation between sets $X_0, \ldots, X_n$ is a subset of the set of n-tuples $X_0 \times \cdots \times X_n$, i.e. $R \subseteq X_0 \times \cdots \times X_n$, or an element of the powerset $\mathcal{P}(X_0 \times \cdots \times X_n)$, i.e. $R \in \mathcal{P}(X_0 \times \cdots \times X_n)$. We write $R(x_0, \ldots, x_n)$ to denote $(x_0, \ldots, x_n) \in R$.*

> **Definition 10** (Partial function). *A partial function f from set X to set Y is a binary relation from X to Y verifying that $\forall x \in X, y, y' \in Y, (x, y) \in f \wedge (x, y') \in f \Rightarrow y = y'$, i.e. x appears at most once as the first element of a pair in f. We note $f \in X \nrightarrow Y$ to denote a partial function from X to Y. The set of the first elements of the pairs defined in f is called the domain of f. We write it $\mathrm{dom}(f) = \{x \mid \exists y \; s.t. (x, y) \in f\}$. When there is no ambiguity, and given an $x \in X$ and $f \in X \nrightarrow Y$, we write $x \in f$ as a shorthand to $x \in \mathrm{dom}(f)$.*

> **Definition 11** (Application). *A total function a, or application, from X to Y is a partial function verifying that all the elements of X appear as the first element of a pair in a, i.e. for all $x \in X$, there exists $y \in Y$ such that $(x, y) \in a$. In other words, the domain of an application from X to Y is equal to the set X. We note $a \in X \rightarrow Y$ to denote an application from X to Y.*

2.1.3 Rule-based definition of sets

All along this memoir, we define sets (especially relations) with rule instances, also called inference rules or judgments. A rule instance can take the following forms:

$$\frac{}{C} \ \text{or} \ \frac{P_1, \ldots, P_n}{C}$$

The left form of rule instance is called an axiom. In the right form of rule instance, $P_1, \ldots, P_n$ are the premises of the rule and C is the conclusion of the rule.

> **Definition 12** (Rule instances). *We define a set R of rule instances as a set of pairs of the form (P/C) where P is a finite (possibly empty) set of premises and C is an element called the conclusion. A pair (P/C) is a rule instance.*

Rule instances define a way to build *derivation trees*. A derivation of C takes either the form of an axiom, i.e. $\overline{C}$, or of a tree with C as a root and with branches composed of the derivation trees of the premises, i.e.

$$\frac{\begin{matrix} \vdots & & \vdots \\ \overline{P_1} & \ldots & \overline{P_n} \end{matrix}}{C}$$

Given a set R of rule instances, we define the set A such that if $(\varnothing/C) \in R$ then $C \in A$, and if $(\{P_1, \ldots, P_n\}/C) \in R$, then if there exists a derivation for all premises $P_1, \ldots, P_n$ then $C \in A$. In fact, the set R of rule instances define the properties that must be verified by the elements of set A. There exists an infinity of sets A that verify the properties outlined by the rule instances R. Thus, we define the set of elements defined by the rule instances R as the least set verifying the properties outlined by the rules. For instance, the two following rules, named rules $\mathrm{Ev}0$ and $\mathrm{Ev}2$, define the set of even natural numbers:

$$\mathrm{Ev}0 \frac{}{IsEven(0)} \qquad\qquad \mathrm{Ev}2 \frac{IsEven(n-2)}{IsEven(n)}$$

The rule $\mathrm{Ev}0$ states as an axiom that 0 is an even number; the rule $\mathrm{Ev}2$ states that for all natural number n, n is an even number if one can derive that fact that $n-2$ is an even number. Thus, we can derive from the previous rules that 4 is an even number by building the following derivation tree:

$$\frac{\dfrac{\dfrac{}{IsEven(0)} \ \mathrm{Ev}0}{IsEven(2)} \ \mathrm{Ev}2}{IsEven(4)} \ \mathrm{Ev}2$$

Starting from $IsEven(4)$, we can apply the $\mathrm{Ev}2$ rule to derive $IsEven(2)$. Then, another application of the $\mathrm{Ev}2$ rule leads to $IsEven(0)$, and we can close the derivation branch by applying the $\mathrm{Ev}0$ rule. Here, the only branch of the tree has reached an axiom, and thus the derivation tree is finite. To further illustrate the use of rule instances in the definition of a

set, let us consider the following minimal language of arithmetic expressions expressed in the Backus-Naur form:

$$e \quad ::= \quad n \mid id \mid e_0 + e_1$$

Here, n ranges over the set of natural numbers $\mathbb{N}$; id ranges over the set $\texttt{string}$ of non-empty strings (i.e. it is the set of identifiers). To evaluate the arithmetic expressions, we need a state $s \in \texttt{string} \nrightarrow \mathbb{N}$ that maps each variable identifier to a natural number value. We assume that only a certain set of declared identifiers can appear in an arithmetic expression. Thus, the state is a partial function from the set of non-empty strings to the set of natural numbers. We define the evaluation relation for the arithmetic expressions with the three following rules:

$$\text{NAT} \quad \frac{}{s \vdash n \to n} \qquad \text{VAR} \quad \frac{}{s \vdash id \to s(id)} \; id \in \texttt{dom}(s) \qquad \text{ADD} \quad \frac{s \vdash e_0 \to n \qquad s \vdash e_1 \to m}{s \vdash e_0 + e_1 \to n + m}$$

Here, the evaluation relation is a subset of the set of triplets $(\texttt{string} \nrightarrow \mathbb{N}) \times e \times \mathbb{N}$. In the rule instances defining the evaluation relation, the $\vdash$ symbol (pronounced *thesis*) means that the left part implies the right part, or it is involved in the evaluation of the right part. For instance, the second rule can be read: in the context of state s, id evaluates to $s(id)$ if $id \in \texttt{dom}(s)$. When the *context* is not involved in the evaluation of the syntactic constructs on the right side of the $\vdash$ symbol, we remove the context and the $\vdash$ from the rule instances. For example, we can define to the NAT rule by:

$$\text{NAT} \quad \frac{}{n \to n}$$

as the state s in not involved in the evaluation of expressions that are natural numbers.

Note that in the VAR rule, there appears an extra statement, at the right of the judgment line, called a *side condition*. This is an extra condition that must hold with all the premises of the rule instance, but which does not generate a derivation tree of its own.

Finally, here is an example of a derivation tree for the evaluation of the expression $x + (y + 1)$ in the context of state $\{(x, 1), (y, 2)\}$:

$$\text{VAR} \quad \frac{}{\{(x,1),(y,2)\} \vdash x \to 1} \; x \in \{x,y\} \qquad \frac{\text{VAR} \; \dfrac{}{\{(x,1),(y,2)\} \vdash y \to 2} \; y \in \{x,y\} \qquad \dfrac{}{1 \to 1} \; \text{NAT}}{\{(x,1),(y,2)\} \vdash y + 1 \to 3} \; \text{ADD}$$
$$\frac{}{\{(x,1),(y,2)\} \vdash x + (y+1) \to 4} \; \text{ADD}$$

Here again, all the branches of the derivation tree have reached axioms, and thus $\{(x,1),(y,2)\} \vdash x + (y+1) \to 4$ is a member of the evaluation relation of arithmetic expressions. It states that the expression $x + (y + 1)$ evaluates to 4 in the state $\{(x,1),(y,2)\}$.

The evaluation relation can also include rule instances defining error cases. For instance, we can add an extra rule to the definition of the evaluation relation for arithmetic expressions;

the following rule states that an arithmetic expression that is an unreferenced variable in state σ results in an error:

$$\frac{\text{UNREFVAR}}{s \vdash id \to \texttt{err}} \quad id \notin \texttt{dom}(s)$$

The special value $\texttt{err}$ is defined to represent error cases. Thus, the evaluation relation defines a subset of triplets $(\texttt{string} \nrightarrow \mathbb{N}) \times e \times (\mathbb{N} \sqcup \{\texttt{err}\})$.

2.2 Induction principles

In the proofs presented in this thesis, we often rely on *induction*. Here are some reminders on induction principles to help the reader understand the proofs of Chapter 6 and Appendix D.

2.2.1 Well-founded induction

The most general principle of induction is called *well-founded* induction. From well-founded induction derives all induction principles presented afterwards.

To introduce well-founded induction, let us define a well-founded relation.

> **Definition 13** (Well-founded relation). *A binary relation $\prec$ over a set A is well-founded if there exist no infinite descending chain, i.e. $\cdots \prec a_i \prec \cdots \prec a_1 \prec a_0$.*

For instance, the *strictly less than* relation $<$ over the set of natural numbers is a well-founded relation.

Let $\prec$ be a well-founded binary relation on a set A. The principle of well-founded induction on the relation $\prec$ says that in order to prove that a property P holds for all elements of A, it suffices to prove that P holds of any $a \in A$ whenever P holds for all $b \in A$ such that $b \prec a$, formally:

$$\big(\forall a \in A, ([\forall b \in A, b \prec a \Rightarrow P(b)] \Rightarrow P(a))\big) \Rightarrow \forall a \in A, P(a)$$

2.2.2 Structural induction

Sometimes, reasoning by induction requires to follow the structure of a given set, i.e. the formation rules of a given set. This kind of reasoning is called structural induction.

Let us consider the formation rules of the set of natural numbers:

$$\frac{\text{ZERO}}{0 \in \mathbb{N}} \qquad \frac{\text{SUCC} \quad n \in \mathbb{N}}{n + 1 \in \mathbb{N}}$$

These rules state that zero is a natural number and that for every natural number, its direct successor is also a natural number. Structural induction describes a way to deduce that a

property holds for the set of natural numbers, first by stating that the property holds for zero, i.e. the minimal element of the set, then by stating that if the property holds for a given number then it holds for its successor. Thus, knowing that $P(0)$ holds, we can deduce that $P(1)$ holds, $P(2)$ holds, $P(3)$ holds, etc. Following the structural induction scheme, given a property P, to prove that P holds for all natural numbers, it is sufficient to prove that:

– P holds for 0

– if P holds for a given n then it holds at $n + 1$

To take another example, if we want to prove that a given property P holds for the set of arithmetic expressions described in Section 2.1.3, we must prove that:

– P holds for all natural number n

– P holds for all identifiers *id*

– if P holds for all sub-expressions e_0 and e_1, then P holds for $e_0 + e_1$

A proof that leverages structural induction follows the structure of the elements we are reasoning upon. In this thesis, we are using structural induction to prove that a sum expression verifies a certain property. Thus, the structural induction follows the recursive definition of the sum term, which is, for any set A, function $f \in A \to \mathbb{N}$ and $X \subseteq A$ and :

$$\sum_{x \in X} f(x) = \begin{cases} 0 \text{ if } X = \varnothing \\ f(x) + \sum_{x' \in X'} f(x') \text{ if } X = \{x\} \cup X' \end{cases}$$

In the second computation branch, it is left implicit that set X' is strict subset of X such that $x \notin X'$ or $X' = X \setminus \{x\}$. Given a set A and a function $f \in A \to \mathbb{N}$, to prove that for all $X \subseteq A$, the property $P(X, \sum_{x \in X} f(x))$ holds, we must show that:

– $\forall X \subseteq A, \; X = \varnothing \Rightarrow P(\varnothing, 0)$

– $\forall X \subseteq A, x \in X, X' \subset X,$
  $X = \{x\} \cup X' \Rightarrow P(X', \sum_{x' \in X'} f(x')) \Rightarrow P(\{x\} \cup X', f(x) + \sum_{x' \in X'} f(x'))$

The induction follows the structure of the function. In this specific case, structural induction is often referred to as *functional* induction. Let us prove Proposition 1 to illustrate the use of structural induction over a sum term:

Proposition 1. *For all $X \subset \mathbb{N}$ a finite set of natural numbers, $\sum_{x \in X} 2x$ is even, i.e.*

$$\exists k \in \mathbb{N} \text{ s.t. } \sum_{x \in X} 2x = 2k$$

Proof.

Let us define the property P as follows:

$$P(X, \sum_{x \in X} 2x) \equiv \exists k \in \mathbb{N} \text{ s.t. } \sum_{x \in X} 2x = 2k$$

Then, let us use structural induction to prove $P(X, \sum_{x \in X} 2x)$.

First, let us show $P(\emptyset, 0)$, i.e. $\exists k \in \mathbb{N}$ s.t. $0 = 2k$. Let us take $k = 0$ to build a tautology.

Then, given an $X' \subset X$ and an $x \in X$ s.t. $X = \{x\} \cup X'$, and assuming that $P(X', \sum_{x' \in X'} 2x')$ holds (i.e. the induction *hypothesis*), let us show $P(\{x\} \cup X', 2x + \sum_{x' \in X'} 2x')$. Appealing to the induction hypothesis, let us take a j such that $\sum_{x' \in X'} 2x' = 2j$. Rewriting $\sum_{x' \in X'} 2x'$ as $2j$:

$\Rightarrow \exists k \in \mathbb{N}$ s.t. $2x + 2j = 2k$

$\Rightarrow \exists k \in \mathbb{N}$ s.t. $2(x + j) = 2k$

$\Rightarrow$Then, let us take $k = x + j$ to obtain a tautology.

$\square$

2.2.3   Rule induction

A specific kind of structural induction, called *rule* induction, is applied to prove properties over sets that are defined by rule instances. Let us take the evaluation relation for arithmetic expressions used in Section 2.1.3 to illustrate the principle of rule induction. To prove that a property P holds for the evaluation relation of arithmetic expressions, which is a subset of triplets $(\texttt{string} \to \mathbb{N}) \times e \times \mathbb{N}$, we must prove that:

– For all $s \in \texttt{string} \nrightarrow \mathbb{N}, n \in \mathbb{N}, P(s, n, n)$

– For all $s \in \texttt{string} \nrightarrow \mathbb{N}, id \in \texttt{string}$, if $id \in \text{dom}(s)$ then $P(s, id, s(id))$

– For all $s \in \texttt{string} \nrightarrow \mathbb{N}, e_0, e_1 \in e, n, m \in \mathbb{N}$,
  if $s \vdash e_0 \to n$ and $P(s, e_0, n)$, and $s \vdash e_1 \to m$ and $P(s, e_1, m)$
  then $P(s, e_0 + e_1, n + m)$

Rule induction states that in order to prove a property over a set defined by rule instances, the property must hold in any construction case of the considered set. The idea is that if the property is preserved from the premises of rules to the conclusions then the property holds for all the elements of the set.

Let us give an application of rule induction to prove a property over the evaluation relation of arithmetic expressions. First, we define, through the three following rules, the relation $\in_r$ stating that a given identifier id is referenced in an arithmetic expression e, written $id \in_r e$:

$$\frac{}{id \in_r id} \text{INRID} \qquad \frac{id \in_r e_0}{id \in_r e_0 + e_1} \text{INRADDL} \qquad \frac{id \in_r e_1}{id \in_r e_0 + e_1} \text{INRADDR}$$

Then, the property of Proposition 2 states that an arithmetic expression that contains references to identifiers that are not part of the current state's domain can not be evaluated.

Proposition 2. *Let* $id \in$ `string`. *For all state s, arithmetic expression e, and natural number n,*

$$id \notin \texttt{dom}(s) \wedge id \in_r e \Rightarrow \neg s \vdash e \rightarrow n$$

Proof.

Let us define the property P as follows:

$$P(s, e, n) \equiv id \notin \texttt{dom}(s) \wedge id \in_r e \Rightarrow \neg s \vdash e \rightarrow n$$

Then, let us use rule induction to prove $P(s, e, n)$.

First, we must prove $P(s, n, n)$. Assuming $id \in_r n$, there is a contradiction as no rule instance defining the relation $\in_r$ includes the case where the considered expression is a natural number.

Then, we must prove $P(s, id', s(id'))$, assuming that $id' \in \texttt{dom}(s)$. We know that $id \in_r id'$, and thus $id = id'$. Then, there is a contradiction between $id \in \texttt{dom}(s)$ and $id \notin \texttt{dom}(s)$.

Finally, we must prove $P(s, e_0 + e_1, n + m)$, assuming that $s \vdash e_0 \rightarrow n$ and $P(s, e_0, n)$, and $s \vdash e_1 \rightarrow m$ and $P(s, e_1, m)$. We know that $id \in_r e_0 + e_1$; this hypothesis has either be constructed by applying Rule INRADDL or Rule INRADDR. If Rule INRADDL has been applied, then we know $id \in_r e_0$; thus, from $P(s, e_0, n)$, we can deduce $\neg s \vdash e_0 \rightarrow n$, which contradicts $s \vdash e_0 \rightarrow n$. We can perform the proof similarly if Rule INRADDR has been applied. $\square$

2.3 The Coq proof assistant

In this section, we present the Coq proof assistant [106]. The Coq proof assistant constitutes our framework to encode the different semantics, programs and proofs involved in the verification of the HILECOP model-to-text transformation. Here, we give an overview of the different concepts underlying the Coq proof assistant. The aim is to give to the reader the tools to understand the different listings presenting Coq code in the following chapters. For a thorough presentation of the Coq proof assistant, the reader can refer to [31, 91, 8].

2.3.1   The Calculus of Inductive Constructions (CIC)

The kernel of the Coq proof assistant implements the Calculus of Inductive Constructions (CIC) [36]. The CIC is a typed lambda-calculus that includes polymorphism, dependent and inductive types. Thus, the CIC permits us to define programs and types similarly; both are *terms* of the language. A program is a term with a certain type, and a type is also a term with a certain type. The type of a type is called a *sort*. We can mention three basic sorts built in the Coq proof assistant: the `Prop` sort which is the type of logical formulas, the `Set` sort which is the type of *small* sets, and the `Type` sort which encompasses the `Prop` and `Set` sorts.

The Coq proof assistant allows us to express logic formulas and to interactively build proofs of these formulas by using a high-level tactic language. The sequence of tactics that builds a proof for a given formula is called a *proof script*. The execution of a proof script builds a proof term. In the CIC, a logic formula can be seen as a *type* and a proof of this formula is an *inhabitant* of the type denoted by the logic formula. Thus, when building a proof term by executing a proof script, the Coq kernel checks that the proof term is of the type of the logic formula by applying typing rules[3]. For instance, let us take two logical propositions A and B. In Coq, we can declare these propositions as elements of the `Prop` type in the Coq top-level loop[4]:

```
Coq < Variables A B : Prop.
```

The `Variables` keyword adds the propositions A and B to the global environment accessed by the Coq kernel. Now, say that we want to prove the *modus ponens* theorem expressed with the propositions A and B, namely that $A \Rightarrow (A \Rightarrow B) \Rightarrow B$. In Coq, we can express it as follows:

```
Coq < Theorem modus_ponens : A → (A → B) → A.
```

Here, we declare the modus ponens theorem as an element of type $A \rightarrow (A \rightarrow B) \rightarrow A$. The arrows represent functional arrows; in fact, $A \rightarrow B$ is a notation for the product type $\Pi x : A.B$ where x is not referenced in B. According to the Curry-Howard correspondence [62], there is an equivalence between a proof term and a program. This correspondence is a consequence of the Brouwer-Heyting-Kolmogorov (BHK) interpretation of the *intuitionistic* logic [81]. Intuitionistic logic is the underlying logic built-in the Coq proof assistant. Intuitionistic logic denies the use of the law of excluded middle to perform proofs. Thus, intuitionistic logic has a *constructivist* approach of proofs. In the intuitionistic setting, one has to provide an explicitly built proof to demonstrate a theorem; one can not rely on pure proof by contradiction by appealing to the law of excluded middle. Thus, a proof term of the logical implication $A \Rightarrow B$ is equivalent to an explicitly built program, or a function, of type $A \rightarrow B$, i.e. a program that takes an element of type A and yields an element of type B. Thus, the type $A \rightarrow (A \rightarrow B) \rightarrow A$ is a valid encoding of the formula $A \Rightarrow (A \Rightarrow B) \Rightarrow B$.

The `Theorem` keyword triggers the interactive proof mode through which the user will build a proof term for the corresponding formula. A simple proof term for the modus ponens theorem is a function that takes an element x of type A and a function f of type $A \rightarrow B$ as inputs, and yields an element of type B by applying the function f to parameter x, i.e. $(f\ x)$. The function takes the form of the following term of the typed lambda-calculus:

---

[3]https://coq.inria.fr/distrib/current/refman/language/cic.html
[4]Coq scripts can be either interpreted or compiled.

$$\lambda(x : \text{A}).\lambda(f : \text{A} \rightarrow \text{B}).(f \; x)$$

While passing this *lambda-term* as a proof term of the modus ponens theorem, the Coq kernel checks the well-typedness of the term by building the following derivation tree, which is a simplified version of the full derivation tree according to the typing rules of the CIC:

$$\frac{\dfrac{}{\text{A B} : \text{Prop}[x : \text{A}, f : \text{A} \rightarrow \text{B}] \vdash f \; : \; \text{A} \rightarrow \text{B}} \text{VAR} \quad \dfrac{\dfrac{}{\text{A B} : \text{Prop}[x : \text{A}, f : \text{A} \rightarrow \text{B}] \vdash x \; : \; \text{A}} \text{VAR}}{} \text{APP}}{\dfrac{\dfrac{\text{A B} : \text{Prop}[x : \text{A}, f : \text{A} \rightarrow \text{B}] \vdash (f \; x) \; : \; \text{B}}{\text{A B} : \text{Prop}[x : \text{A}] \vdash \lambda(f : \text{A} \rightarrow \text{B}).(f \; x) \; : \; (\text{A} \rightarrow \text{B}) \rightarrow \text{B}} \text{LAM}}{\text{A B} : \text{Prop}[] \vdash \lambda(x : \text{A}).\lambda(f : \text{A} \rightarrow \text{B}).(f \; x) \; : \; \text{A} \rightarrow (\text{A} \rightarrow \text{B}) \rightarrow \text{B}} \text{LAM}}$$

In the above derivation tree, the global and the local environment are represented at the left of the thesis symbol $\vdash$. The local environment is represented by square brackets. The global environment is represented at the left of the local environment. At the root of the derivation tree, the global environment contains our two previously declared logical propositions A and B, whereas the local environment is empty. The application of the LAM rule adds new entry to the local environment; the APP triggers the type-checking of the left and the right part of an application; the VAR rule checks that a term is well-typed if it is referenced as an element of the given type in the global or the local environment.

As said before, the `Theorem` keyword triggers the interactive proof mode. The interactive proof mode will accompany the user to an incremental building of a proof term for the current goal, i.e. the current logic formula we want to prove. Then, to prove the modus ponens theorem, the following interface is first presented to the user:

```
Coq <  Theorem modus_ponens : A → (A → B) → B.
1 subgoal
```

```
============================
A → (A → B) → B
```

The term under the horizontal bar represents the current goal to prove, i.e. the current formula for which we are building a proof term. Above the horizontal bar are referenced the variables constituting the local environment. At the beginning of the proof, the local environment is empty – and so is the local environemnt at the root of the derivation tree presented above. To build a proof term in interactive mode, the user will then invoke commands called *tactics*. Each tactic invocation corresponds to the invocation of a typing rule of the CIC performed by the Coq kernel. To build a proof term for the modus ponens theorem, the first thing to do is to invoke the LAM rule; this is done by appealing to the `intros` tactic.

```
Coq <  intros x.
1 subgoal
```

```
x : A
============================
(A → B) → B
```

Here, the user passes to the system the name of the variable that will be introduced in the local environment by the LAM rule, i.e. the variable x. Then, we repeat the operation, applying the LAM rule a second time to introduce an element of type A → B in the environment.

```
Coq < intros f.
1 subgoal

x : A
f : A → B
============================
B
```

Then, based on the local environment, we can build an object of type B by applying f to the input x. We can do it by appealing to the `apply` tactic. The `apply` tactic invokes the APP rule.

```
Coq < apply (f x).
No more subgoals.

Coq < Qed.
```

After the invocation of the `apply` tactic, the proof term for the modus ponens theorem is completely built, thus, the Coq top-level loop displays the message that all goals are completed. Then, we can close the interactive proof mode and store the proof of the modus ponens theorem in the global environment by using the `Qed` keyword.

Another way to prove the modus ponens theorem is to directly pass the proof term with the `exact` tactic.

```
Coq < exact (fun (x : A) ⇒ fun (f : A → B) ⇒ f x).
No more subgoals.

Coq < Qed.
```

Here the term `fun (x : A) ⇒ fun (f : A → B) ⇒ f x` represents the lambda-term $\lambda(x : A).\lambda(f : A \to B).(f\ x)$ (i.e. the proof term) in the Coq syntax.

2.3.2   Inductive types

One of the major strength of the CIC, and therefore of the Coq proof assistant, is the possibility to enrich the global type system with the definition of inductive types. For instance, here is the definition of the type of natural numbers, named `nat`:

```
Inductive nat : Set :=
| O : nat
| S : nat → nat.
```

The `nat` type is of the `Set` sort (remember that the type of a type is called a sort). The `nat` type is defined through two constructors, represented by the pipe-separated entries. The `0` constructor states that zero is a natural number. The `S` constructor takes a natural number as input and yields the successor to this natural number. This corresponds to the structural definition of natural numbers in Peano's arithmetic. Thus, in this setting, the number 2 is represented by (`S (S 0)`), the number 3 by (`S (S (S 0))`), etc. The result of the evaluation of an inductive type, declared through the `Inductive` keyword, is the addition of this type and each of its constructors to the global environment accessible by the Coq kernel. Also, a corresponding structural induction principle is generated at the evaluation of an inductive type definition. For instance, the `nat_ind` induction principle is generated at the evaluation of the `nat` type. It is a proof term of the logical formula denoting the structural induction principle over the `nat` type, i.e.:

```
forall P : nat → Prop, P 0 → (forall n : nat, P n → P (S n)) → forall n : nat, P n
```

Then, the induction principle `nat_ind` can be used to perform structural induction in a proof involving natural numbers. For instance, say that we want to prove the following theorem stating that a natural number elevated at the power 2 is always greater than or equal to itself. We can write as follows:

```
Coq < Theorem ge_pow2 : forall n : nat, n <= n * n.
1 subgoal

============================
forall n : nat, n <= n * n
```

Then, we can use the `nat_ind` induction principle to prove such a theorem. Most conveniently, the built-in `induction` tactic chooses the appropriate induction principle based on the type of its argument. Thus, the following command invokes the `nat_ind` induction principle over the universally quantified variable `n`:

```
Coq < induction n.
2 subgoals

============================
0 <= 0 * 0
subgoal 2 is:
S n <= S n * S n
```

The result of the invocation of the `induction` tactic is a branching in the proof tree. Thus, the system indicates that two subgoals must be proved to complete the proof of the `gt_pow2` theorem. These two subgoals correspond to the proof of $P(0)$ and the proof that assuming $P(n)$ we can show $P(n+1)$, as agreed with structural induction. Here, the property P is defined by $P(n) \equiv n \leq n \times n$. We can use the built-in `lia` tactic, defined in the `Lia` module of the Coq standard library, to solve the two remaining subgoals. The `lia` tactic implements a whole decision procedure to prove theorems involving systems of equalities and inequalities over the set of natural numbers. We can combine the `induction` tactic with the `lia` tactic using the semi-colon operator. Then, the `lia` tactic is applied to all the subgoals generated by the `induction` tactic.

```
Coq < induction n; lia.
No more subgoals.
Coq < Qed.
```

The Coq proof assistant permits us to define proof tactics, or procedures, in order to autom-atize some proof tasks. At the top-level of the Coq proof assistant, the `Ltac` and `Ltac2` languages are the supports for the definition of this kind of tactics. These languages allow us to compose sequences of tactics, to perform pattern matching over the local environment and the current goal in interactive proof mode, to define loops or recursive tactics, etc. Even though the `Ltac` and `Ltac2` languages offer a lot of possibilities, the user willing to implement complex proof tactics must turn to the OCaml language which the implementation, and thus meta-language, of the Coq proof assistant. For instance, the `lia` tactic is implemented partly with the `Ltac` lan-guage and as a OCaml program.

Leveraging the definition of inductive types, the syntactic constructs of programming lan-guages are also easily implemented. Here is the implementation of the syntax of arithmetic expressions presented in the previous section:

```
Inductive e : Set :=
| enat : nat → e
| eid : string → e
| eadd e → e → e.
```

Each constructor corresponds to a construction case in the definition of arithmetic expres-sions in the Backus-Naur form. The Coq system also generates the induction principle fol-lowing the structure of arithmetic expressions (thus, a structural induction principle). The induction principle is a proof term of the following logical formula:

```
forall P : e → Prop,
    (forall n : nat, P (enat n)) →
    (forall id : string, P (eid id)) →
    (forall e0 : e, P e0 → forall e1 : e, P e1 → P (eadd e0 e1)) →
    forall e : e, P e
```

The evaluation relation for arithmetic expressions is defined similarly:

```
Inductive evale (s : string → option nat) : e → nat → Prop :=
| evalnat : forall n : nat, evale s (enat n) n
| evalid : forall (id : string) (n : nat),
            s id = Some n →
            evale s (eid id) n
| evaladd : forall (e0 e1 : e) (n m : nat),
            evale s e0 n →
            evale s e1 m →
            evale s (eadd e0 e1) (n + m).
```

In the above listing, the state that yields the value of identifiers present in an arithmetic expression is a named parameter of the `evale` relation, i.e. the `s` parameter. Parameters which are not varying from one construction case to another can be passed as named parameters

while defining an inductive type. The state `s` takes a string identifier as input and yields an `option` to a natural number. As so, the `option` type permits the definition of partial functions. The identifiers that belong to the domain of state `s` will be associated with `Some` natural number, whereas the unreferenced identifiers will be associated with the `None` value of the `option` type. The `Some` and the `None` constructors are the two constructors of the `option` type which is defined in Coq as follows:

```
Inductive option (A : Type) : Type :=
| Some : A → option A
| None : option A.
```

The `option` type is parameterized by a type `A` that will set the type of elements passed to the `Some` constructor. As so, the `option` type is an example of generic type.

2.3.3 Functional programming

As told in the presentation of the CIC, the Coq proof assistant permits to write functional programs, including the definition of recursive functions. The definition of a recursive function is performed with `Fixpoint` keyword. Here is an example of recursive function defined in Coq. The `pow` function takes two natural numbers `a` and `n` as inputs and yields `a` to the power `n`.

```
Fixpoint pow (a n : nat) {struct n} : nat :=
  match n with
  | 0 ⇒ 1
  | S m ⇒ a * pow a m
  end.
```

In the body of the `pow` function, the `match` construct performs pattern-matching over the structure of the input `n`. The input `n` is an element of the `nat` type, and thus it could either have been built with the `0` constructor or as the successor of another element of the `nat` type, i.e. with the `S` constructor. The `match` construct enumerates all the possible construction cases for the given input. Each construction case leads to a pipe-separated entry; for each entry, the structure of the input appears at the left of the arrow, and the result returned appears at the right the arrow. In the above example, 1 is returned if `n` equals `0`, and the result of the multiplication of `a` with the recursive call `pow a m` is returned if `n` is the successor of a certain `m`. In that case, we have $m = n - 1$, and then the recursive call `pow a m` can be read as `pow a (n − 1)`.

When declaring a recursive function, the user must specify which parameter is structurally decrementing through the recursive call. This is performed through {`struct` *id*} annotation, where *id* denotes one parameter of the declared function. This information permits to the Coq kernel to generate the fixpoint equation for the function, thus proving that the function is always terminating. For consistency reasons, all Coq functions must terminate and must be total. A user willing to implement a non-terminating function equivalently define the function as an inductive type where termination limitations do not apply. For instance, let us say that we want to implement this following *ill-formed* version of the `pow` function:

```
Fixpoint pow (a n : nat) {struct n} : nat :=
  match n with
  | 0 ⇒ 1
```

```
  | _ ⇒ a * pow a n
  end.
```

Clearly, this function diverges for all `n` strictly superior to 0. The Coq kernel will not allow such a definition; thus, we can implement the `pow` function as the following relation $\text{Pow} \subseteq (\mathbb{N} \times \mathbb{N} \times \mathbb{N})$:

```
Inductive Pow (a : nat) : nat → nat → Prop :=
| Pow0 : Pow a 0 1
| Pown : forall n res, Pow a n res → Pow a n (a * res).
```

The `Pow` relation takes three parameters of the `nat` type and projects a value in the `Prop` type (meaning that the `Pow` relation is a predicate). The third `nat` parameter corresponds to the result of the computation of a^n given that the two first parameters are *a* and *n*. Determining the result of the computation of a^n is equivalent to finding a natural number *m* that verifies that `Pow a n m` holds. In intuitionistic logic, finding a proof of the existence of a *m* such that `Pow a n m` holds amounts to explicitly building such a *m*. Here, one can notice that when the second parameter passed to the `Pow` relation is greater than zero, the formation rules of the `Pow` relation will not permit us to find a result for the computation. Thus, a tactic implementing a proof search for a *m* such that `Pow a n m` holds when $n > 0$ will never terminate.

2.3.4   Dependent types

In the listings that the reader will find in the following chapters, and also in the code repository associated with this thesis, some data structures are *dependently-typed* structures. Thus, we introduce here the notion of dependent type and how it is expressed with the Coq proof assistant.

A type is said to be dependent when its expression depends on one or more elements of other types. To give an example of dependent type, let us take the definition of polymorphic lists that carry their own length. In Coq, these lists are defined as follows:

```
Inductive listn (A : Type) : nat → Set :=
| niln : list A 0
| consn : forall n : nat, A → listn A n → listn A (S n).
```

The `listn` takes the type `A` of its elements as its first parameter, then its second parameter is an element of the `nat` type which represent the actual length of the list. Note that the first parameter, i.e. the `A` parameter, of the `listn` type alone is not sufficient to qualify `listn` as a dependent type. The `A` parameter is the expression of the polymorphism of the elements of the list involved in generic programming. Polymorphism relates to the fact that the `A` type is general enough to accept multiple types as the type of the list's elements. The `niln` constructor of the `listn`, i.e. the constructor of the empty list, has the type of lists of length 0. The `consn` constructor permits to add a new element at the head of an existing *tail* list to build a new list. Thus, the type of the resulting list is the type of lists of length $n + 1$, where *n* is the length of the tail list.

To further illustrate the use of dependent types, let us say that we want to write a function that takes two natural numbers *n* and *m* as inputs, and yields $n - m$ only if $n \geq m$. Thus, the function takes two parameters *n* and *m*, and a third parameter which is the proof that $n \geq m$.

This third parameter *depends* on the two previous parameters, and thus the function is said to be a dependently-typed function. In Coq, it would be written as follows:

```
Definition my_sub (n m : nat) (pf : m <= n) : nat := n − m.
```

Even though, in its definition body, the `my_sub` function simply appeals to the Coq built-in subtraction function, passing a proof that *m* is less than or equal to *n* adds a constraint to the computation of the subtraction. One can see how dependent types can help check that the parameters of programs meet some properties at definition time. Constraining the type of parameters during the definition of programs reduces the proof efforts afterwards, but adds programming complexities at the moment of the definition. Thus, there is a trade-off between using dependent types to constraint the structures and programs at the moment of their definition, or letting the structures and programs as constraint loose as possible at the cost of having to prove much more properties afterwards.

To conclude the subject of dependent types, we often use *sigma* types to define a type of elements that meet a given property. Sigma types are *constructivist* versions, coming from the intuitionistic logic, of existential logic formulas. A sigma type expresses the dependence between a parameter and a proof of a given property that possibly depends on another parameter. As so, sigma types are useful to express intentional sets (cf. Section 2.1.2). In the Coq standard library, the definition of the sigma type is as follows:

```
Inductive sig (A:Type) (P:A → Prop) : Type := exist : forall x:A, P x → sig P.
```

The `sig` type only constructor takes an element x of type A along with a proof that x meets a certain property P. For instance, if we want to define the type of natural numbers that are strictly greater than zero, we can do it as follows:

```
Definition natstar := sig nat (fun n : nat ⇒ n > 0).
```

The property passed as the second argument of the `sig` type is expressed by a lambda abstraction (denoted by the `fun` keyword) that takes a parameter n of type `nat` and returns a proof that n is strictly greater than zero. The Coq standard library defines a notation to write sigma types as intensional sets. Thus, we can write the `natstar` type as follows:

```
Definition natstar2 := { x : nat | x > 0}.
```

We can leverage sigma types to rewrite the `my_sub` function presented above. In the following version, the type of the *m* parameter carries the proof that *m* is less than or equal to *n*:

```
Definition my_sub2 (n : nat) (m : { x : nat | x <= n }) : nat := n − (proj1_sig m).
```

Here, we can no longer directly subtract *n* with *m* as the type of *m* is no longer `nat` but { x : nat | x <= n }. We have to extract the first part of the *m* parameter with the help of the `proj1_sig` function. The first part of an element of the { x : nat | x <= n } type corresponds to the natural number x verifying the following property x <= n, and the second part corresponds to the proof that x verifies the property.

Chapter 3

Implementation of the HILECOP Petri nets

In this chapter, we present the input formalism of our transformation function: Synchronously executed Interpreted Time Petri Nets with priorities (SITPNs). The formalization of the SITPN structure and semantics is mainly the result of two former Ph.D. theses [71, 78]. However, we contributed to the simplification and clarification of both the definition of the SITPN structure and its semantics. Moreover, we added complementary definitions that are required to express the semantic preservation theorem about the HILECOP model-to-text transformation (cf. Chapter 6). Our main contribution in this part lies in the implementation of the SITPN structure and semantics with the Coq proof assistant. This chapter is structured as follows: Section 3.1 is a reminder on the PN formalism and also gives an informal presentation of SITPNs; Section 3.2 provides the formal definitions of the SITPN structure and semantics; Section 3.3 deals with the implementation of SITPNs with the Coq proof assistant.

3.1 Informal presentation of Synchronously executed Petri nets

Here, fundamentals on the Petri net formalism are outlined, and certain classes of Petri nets are described more precisely. Then, the specificities of the Petri nets used to design the behavior of electronic components in the HILECOP methodology are presented. For more information on the topic of Petri nets, the reader can refer to [40], [84], or [43].

3.1.1 Preliminary notions on Petri nets

Petri nets (PNs), invented by C. A. Petri [93], have been designed to model a broad range of *dynamic* systems: resource sharing between concurrent processes [40], behavior of agents in multi-agent systems [28], behavior of digital components [113]. A Petri net is a directed graph composed of two types of node: place nodes (*circles*) and transition nodes (*squares* or *lines*). As shown in Figure 3.1, place nodes usually represent a part of the state of the modelled system, here, the states of two computer processes and a semaphore; transition nodes usually refer to events triggering the system evolution (or state changing). In Figure 3.1, places p_0, p_3 and *sem* are marked with tokens, represented by filled black circles. This means that places p_0, p_3 and *sem* are currently active. The distribution of tokens over places is called the *marking* of the

net. The marking of a Petri net reflects the overall state of the modelled system at a certain moment in its activity cycle. We will see later that there exists a lot of different classes of PNs. Figure 3.1 presents an example of the most simple form of PN, namely, the *place-transition* PN. In this chapter, when no precision is given on the class of PN considered, a PN refers to a *place-transition* PN.
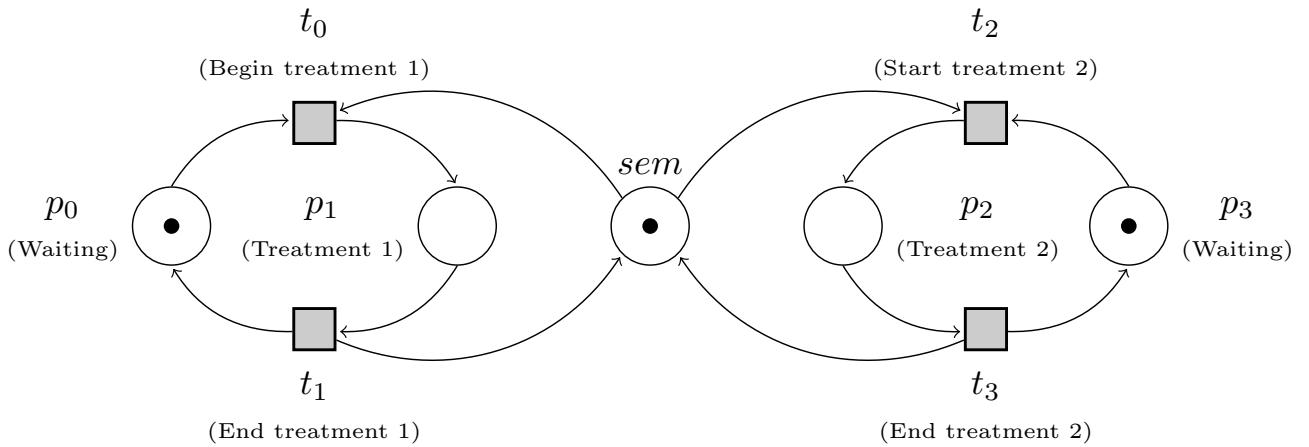
FIGURE 3.1: An example of Petri net. The semaphore place *sem* prevents the parallel execution of *Treatment 1* (place p_1) and *Treatment 2* (place p_2).

Edges

In a Petri net, directed edges link together places and transitions. Places cannot be linked to other places, and the same stands for transitions. There are two kinds of edges, *pre* or *incoming* edges, going from a place to a transition, and *post* or *outcoming* edges, going from a transition to a place. Places linked to a transition *t* by incoming (resp. outcoming) edges will be referred to as the *input* places (resp. *output* places) of *t*. The same stands for the transitions linked to a place *p*. For instance, in Figure 3.1, p_0 and *sem* are the input places of t_0, and p_1 is the output place of t_0; t_1 and t_3 are the input transitions of place *sem*, and t_0 and t_2 are the output transitions of *sem*. Some weight –a natural number– is associated to the edges of a Petri net. If no label appears on the edge then one is the default weight. Petri nets are said to be *generalized* when the weight of the edges are possibly greater than one.

Transition firing

In a Petri net, the marking evolves based on a token consumption-production system. Transitions consume tokens from their input places, and produce tokens to their output places. This whole process is called *transition firing*. In order to be *firable*, a transition must be *sensitized* (or *enabled*), meaning that the number of tokens in each of its input places must be equal or greater than the weight of the associated incoming edges. For instance, in Figure 3.1, the transition t_0 is sensitized because the weight of the arcs (p_0, t_0) and (*sem*, t_0) is of one (default value), and place p_0 and *sem* are marked with one token. As a counter example, transition t_3 is not

sensitized because its input place p_2 holds no token, where at least one token is expected for t_3 to be sensitized. Depending on the class of PNs that is considered, other parameters affect the *firability* of transitions (see interpreted Petri nets, time Petri nets and Section 3.1.2). When a sensitized transition is fired, tokens are retrieved from their input places (as many tokens as the weight of the input arcs) and produced in their output places (as many tokens as the weight of the output arcs). This process represents the occurrence of an event (denoted by the transition) triggering the evolution of the system from one state to another. Figure 3.2 shows the state of the PN of Figure 3.1 after the firing of the transition t_0.
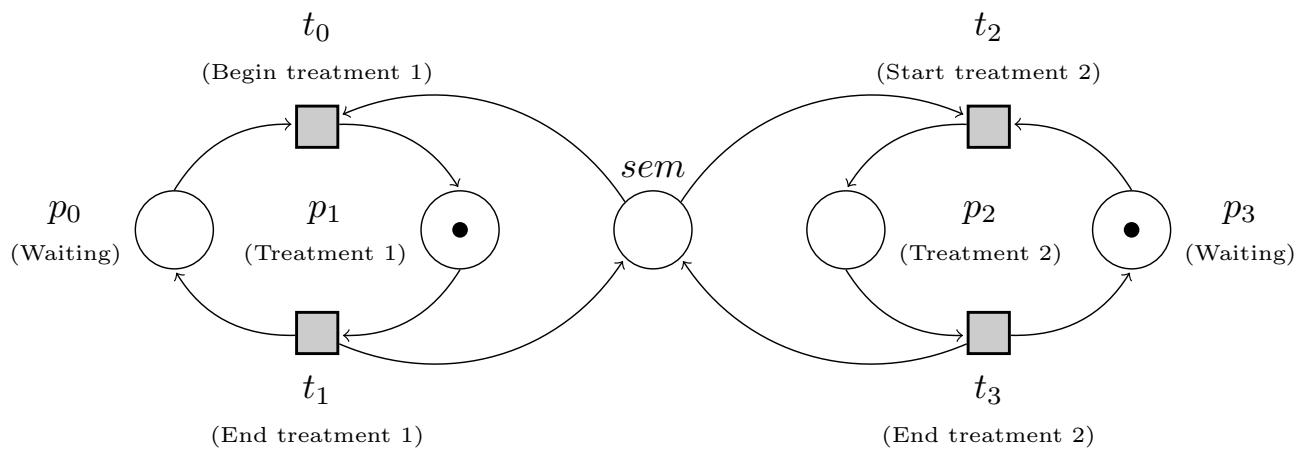
FIGURE 3.2: The PN of Figure 3.1 after the firing of transition t_0.

In Figure 3.2, the tokens in the input places of t_0, i.e. places p_0 and *sem* have been consumed, and one token has been produced in the output place p_1. The current marking indicates that the task "Treatment 1" is being performed (place p_1 is active).

In Figure 3.1, transition t_0 and t_2 are enabled at the same time. However, the *standard* semantics of PNs is such that only one transition can be fired in that case. Either t_0 consumes the token in place *sem* or t_2 does, but never both. Thus, the transition firing process in the standard PN semantics is a nondeterministic process. From the marking of Figure 3.1, two markings are reachable: the marking resulting of the firing of transition t_0 and the one resulting of the firing of transition t_2. Also, in standard PNs, the transition firing process is asynchronous; as soon as a transition is enabled, the transition firing process can be triggered.

Extended Petri nets

The class of *extended* Petri nets introduces the inhibitor and test edges. As shown in Figure 3.3, test arcs are represented with a black-filled circle head and inhibitor arcs with a white-filled circle head. Inhibitor and test edges are incoming edges, always coming from a place toward a transition.
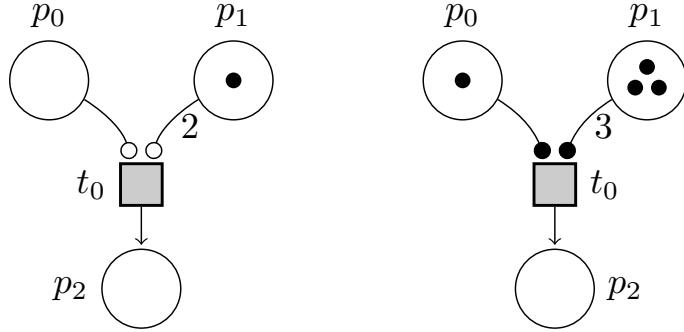
FIGURE 3.3: Two examples of extended Petri nets; on the left side, a PN with inhibitor arcs; on the right side, a PN with test arcs.

The particularity of the inhibitor and test edges is that they are not consuming tokens in input places after the firing of a transition. They are just testing the number of tokens in incoming places to determine if the transition is enabled. Inhibitor arcs ensure that the number of tokens in input places is strictly lower than their weights; test arcs ensure that the number of tokens in incoming places is equal or greater than their weights. Therefore, on the left side of Figure 3.3, transition t_0 is sensitized because there is strictly less than one token in place p_0 and strictly less than two tokens in place p_1. On the right side of Figure 3.3, transition t_0 is sensitized because there is at least one token in place p_0 and three tokens in place p_1.

Interpreted Petri nets

As stated in [40], Interpreted Petri Nets (IPN) "can be applied to various interpretations according to the use wished to be made of it". In its general definition, an IPN is associated with a finite set of variables V, a finite set of operations O, and a finite set of conditions C. Operations of the O set are associated with places and triggered when the places become marked. The execution of operations affects the value of the variables, and the value of conditions depends on Boolean expressions computed upon the variables. Conditions are associated with transitions and become involved in the firing process. Thus, in an IPN, a transition is firable if:

– It is enabled.

– All its associated conditions are `true`.

Among other applications, IPNs are handy to model the behavior of hardware controllers. Thus, interpretation aspects have been naturally introduced to the HILECOP high-level models, which are models of hardware systems. The HILECOP version of IPNs refines the concepts of the general definition. In this version, the set of variables corresponds to the set of VHDL signals that are handled by the model; a signal can be an input port, an output port or an internal signal of the modeled hardware circuit. The operations are separated in two kinds, namely: actions and functions. Actions (or continuous operations) are associated to the places; all the actions associated to a place p are activated as long as p is marked (i.e. as long as p holds a token). Functions (or discrete operations) are associated to the transitions; when a transition t is fired, all functions associated to t are executed once.

Figure 3.4 illustrates the use of actions, functions and conditions in an interpreted Petri net as applied in the HILECOP high-level models.
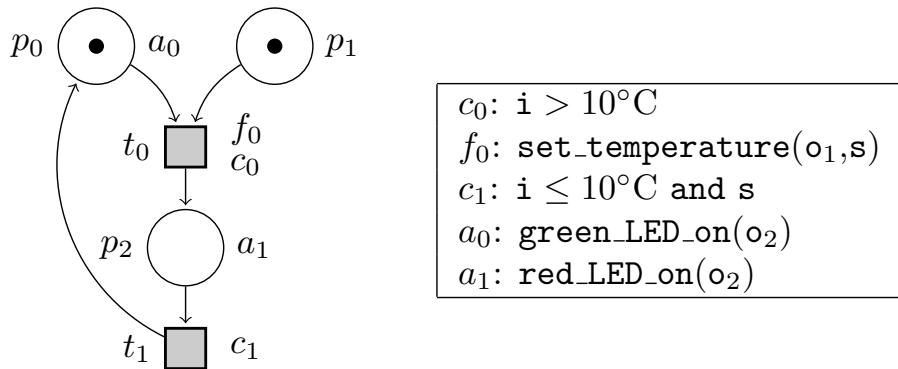
FIGURE 3.4: An example of interpreted Petri net; on the left side, the interpreted Petri net; on the right side, examples of tests associated to conditions and operations associated to actions and functions.

In Figure 3.4, the set of VHDL signals, on which the interpretation elements act upon, is $\{i, s, o_1, o_2\}$. Here, signal i is an input port of the hardware, s is an internal signal, and o_1 and o_2 are two output ports. The action a_0 is activated as place p_0 is marked by one token; thus, the operation green_LED_on(o_2) is currently executed. Also, function f_0 will be executed (i.e. operation set_temperature(o_1, s)) at the firing of t_0, that is if condition c_0 is true and t_0 is sensitized. On the right side of Figure 3.4, we associate Boolean expressions with conditions; these expressions depend on the value of the signals declared by the model. Also, we associate actions and functions with operations that handle the signals of the model which are passed as inputs. Concretely, in the HILECOP high-level models, functions and actions are declared as VHDL procedures. Listing 3.1 gives one possible implementation of the set_temperature operation as a VHDL procedure; the set_temperature operation is associated with function f_0 in Figure 3.4.

```vhdl
1  procedure set_temperature(signal tmp : out integer; signal flag : inout std_logic) is
2  begin
3    if flag = '1' then
4      tmp ⇐ 30;
5      flag ⇐ '0';
6    else
7      tmp ⇐ 10;
8      flag ⇐ '1';
9    endif;
10   return;
11 end set_temperature;
```

LISTING 3.1: An example of VHDL procedure implementing the operation set_-temperature associated with the function f_0.

In Listing 3.1, the `set_temperature` procedure declares two parameters: the `tmp` signal which is a write-only signal of type `integer`, and the `flag` signal which is a both readable and writable signal of the Boolean type (`std_logic` in VHDL). The `set_temperature` procedure checks the value of the `flag` signal and assigns a new value to the `tmp` and `flag` signals accordingly. The $\Leftarrow$ operator is the assignment operator for signals in the VHDL syntax (more on that in Chapter 4).

Therefore, to compute the evolution of an IPN, we must be able to interpret the content of operations associated with actions and functions, and also to evaluate the Boolean expressions associated with conditions. This implies the definition of interpretation rules that give an execution semantics to operations and expressions. For now, we consider a simplified version of the interpretation that permits us not to bother with the semantics of operations and Boolean expressions. In fact, we do not consider the set of VHDL signals as a part of the HILE-COP PN structure; thus, we are not interested in the representation of the Boolean expressions associated with conditions, nor in the VHDL procedures that implement functions and actions. Regarding conditions, we consider that they directly receive their value from an environment that would have computed in our stead the values of the Boolean expressions. Thus, we no more have to consider the Boolean expressions associated with conditions, and only have to rely on the values given by the environment. Regarding actions and functions, we are only interested in the fact that a given action/function is activated/executed but no more in actually executing the associated operation.

Time Petri nets

In a time Petri net (TPN), time intervals are associated to transitions. The goal is to constrain the firing of a transition to a certain time window. As shown in Figure 3.5, time intervals are of the form $[a, b]$, where $a \in \mathbb{N}^*$ and $b \in \mathbb{N}^* \sqcup \{\infty\}$. Other definitions of time intervals exist for TPNs (e.g. with real numbers), but here we will only consider the latter definition. In Figure 3.5, time counters are represented in red between diamond brackets. The current value of time counters is part of the state of the TPN, along with its current marking, whereas time intervals are part of the static structure of the TPN.
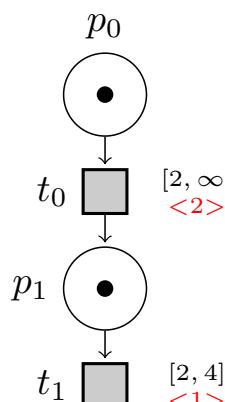
FIGURE 3.5: An example of time Petri net. The value of time counters appears in red.

For each sensitized transition associated with a time interval, time counters are incremented at a certain time step, previously defined by the designer. For instance, in the case of SITPNs, i.e. Petri nets used in the HILECOP methodology, the reference time step for the increment of time counters is the clock cycle.

When a transition associated with a time interval is fired or disabled, a reset order is sent to the transition to set its time counter to zero. In time Petri nets, a transition is firable if:

– It is enabled.

– Its time counter value is within its time interval.

For instance, in Figure 3.5, only transition t_0 is firable. Moreover, there are several possible firing policies for TPNs. Here, we will only consider the *imperative* firing policy: as soon as a time counter reaches the lower bound of a time interval, the associated transition must be fired if all the other firability conditions are verified.

Petri nets with priorities

Two transitions are in structural conflict if they have a common input place connected through a *basic* arc (i.e. neither inhibitor nor test arc). When two transitions in structural conflict are firable at the same time and if the firing of one of the transitions disables the other, then, the conflict becomes *effective*. In a Petri net with priorities, it is possible to specify a firing priority in the case where the conflict between two transitions becomes effective. In that case, the transition with the highest firing priority will always be fired first. Figure 3.6 illustrates the application of a priority relation to solve the effective conflict between two transitions.

(a) Without priorities. (b) With priorities.
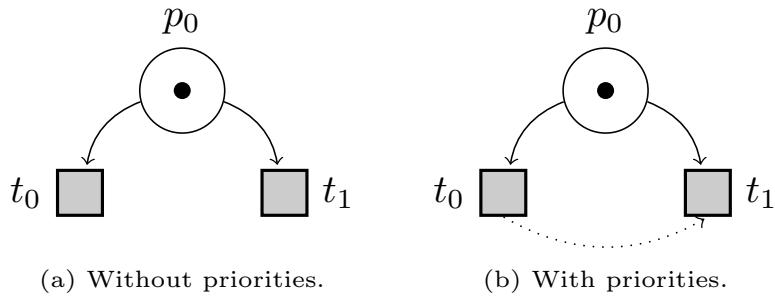
FIGURE 3.6: An example of transitions in structural and effective conflict. In sub-figure (b), the dotted arrow represents the priority relation between t_0 and $t1$. The transition with the highest firing priority is at the source of the arrow; here, transition t_0.

3.1.2 Particularities of SITPNs

Here, we will informally present the specificities of the Petri nets describing the internal behavior of the HILECOP high-level model components. These Petri nets are called: Synchronously executed, extended, generalized, Interpreted, Time Petri Nets with priorities or SITPNs. SIT-PNs are a combination of multiple classes of PNs, namely: extended PNs, generalized PNs,

interpreted PNs, time PNs and PNs with priorities. These classes were presented in the above section. We will now talk about another aspect of SITPNs that constitutes the originality of the formalism compared to the standard PN semantics: its synchronous execution.

The class of interpreted Petri nets increases the expressiveness of the HILECOP high-level models. However, to ensure the safe execution of functions after the synthesis of the designed circuit, the whole system must be synchronized with a clock signal [71]. As a consequence, a clock signal also regulates the evolution of SITPNS (i.e. it is a part of their semantics). The evolution of a SITPN is *synchronized* with two clock events: the rising edge and the falling edge of the signal. Figure 3.7 depicts the process of state evolution, following the clock signal.

Updates the marking, the reset order values and the function execution status.

Updates the condition values, the time counter values, and the action activation status.
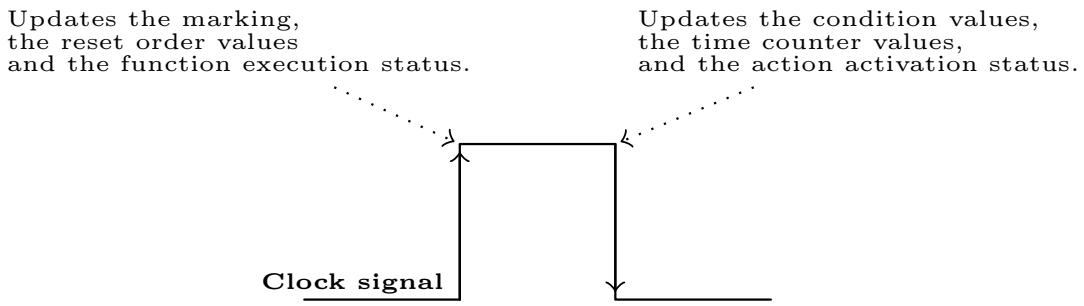
Clock signal

FIGURE 3.7: Evolution of an SITPN synchronized with a clock signal.

Considering the different classes of PNs that define SITPNs, the state of a SITPN is characterized by its marking, the value of time counters, the reset orders assigned to time counters, the execution/activation status of actions/functions (Boolean values), and the value of conditions (also Boolean). As shown in figure 3.7, the state evolution process of a SITPN is divided into two steps. The rising edge of the clock signal triggers the marking update, which is the consequence of transition firing; all transitions that have been fired or disabled by the firing process receive reset orders; all functions associated with fired transitions are executed. Then, on the falling edge of the clock signal, the environment provides a new value to each condition. The falling edge triggers the evolution of the time counter values; values are incremented, reset, or stalling in the case where a time counter has reached the upper bound of its associated time interval (see the following remark on locked time counters). Finally, all actions associated with marked places are activated. Figure 3.8 gives an example of the evolution of the state of a given SITPN through one clock cycle. The aim of this figure and the explanation that follows is to give some hints to the reader about the semantics of SITPNs before giving its formal definition in Section 3.2.4.
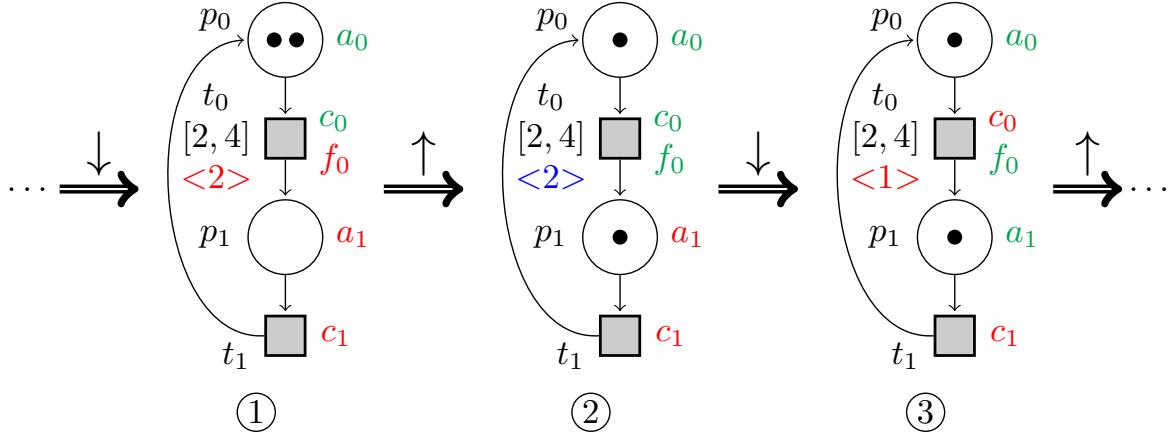
FIGURE 3.8: Evolution of a SITPN over one clock cycle. Conditions appear in green when their value is true and in red otherwise; actions and functions appear in green when they are activated/executed and in red otherwise; time counters appear in red and between diamond brackets; time counters appear in blue when they receive reset orders.

From Step 1 to Step 2, the rising edge of the clock signal triggers the SITPN state evolution. Here, transition t_0 is fired. At Step 1, transition t_0 gathers all the necessary conditions to trigger the firing process, namely:

– t_0 is enabled by the current marking.

– Condition c_0 is true (appears in green).

– The value of t_0's time counter is within the associated time interval ($2 \in [2, 4]$).

As a consequence, one token is consumed in place p_0 and one token is produced in place p_1. Also, function f_0 is executed at the occurrence of the rising edge of the clock signal, and thus, f_0 appears in green at Step 2. Due to the firing of t_0 at the rising edge, a reset order is sent to the time counter of t_0, and it appears in blue at Step 2. From Step 2 to Step 3, the falling edge updates the action activation status: a_0 stays activated as place p_0 is still marked; a_1 becomes newly activated as p_1 is marked. The value of time counters is updated: t_0's time counter is set to zero as the transition previously received a reset order. However, as t_0 is still enabled by the new marking, its time counter is incremented. Thus, the resulting time counter value at Step 3 is of one (i.e. result of reset plus increment). Also, the environment provides a new value to each condition. As a consequence, condition c_0 takes the value false and condition c_1 keeps the same value.

A remark on priorities

The semantics of synchronous execution is that all transitions are fired at the same time. In Figure 3.9, transitions t_0 and t_1 are both sensitized by place p_0, and consequently are both fired at the same time. The system acts as if two tokens were available in place p_0, one for the firing of t_0 and another for the firing of t_1.
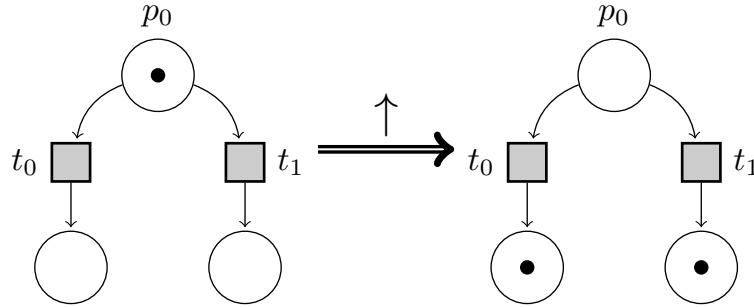
FIGURE 3.9: Double consumption of one token in a SITPN. On the left side, the current marking before the firing of t_0 and t_1; on the right side, the marking resulting of the firing of t_0 and t_1. The arrow indicates the occurrence of a rising edge that triggers the firing process.

In the context of a SITPN, a branching like the one of Figure 3.8, normally interpreted as a disjunctive branching, takes the semantics of a conjunctive branching when no priority are prescribed between the conflicting transitions. To avoid the phenomenon of "double consumption" of tokens, we enforce the resolution of any structural conflict by means of mutual exclusion or through the application of priorities. This policy about the resolution of structural conflicts is part of the definition of a well-defined SITPN presented in Section 3.2.6. The property of well-definition is mandatory to produce safe models of digital systems.

When a structural conflict between transitions is solved with priorities, the firing process follows a slightly different mechanism. As illustrated in Figure 3.10, to determine which transitions of t_0, t_1 and t_2 must be fired, a *residual marking* is computed by following the priority order. For each transition of the group t_0, t_1 and t_2, the residual marking represents the remaining tokens in p_0 after the firing of transitions with a higher firing priority. Thus, in the semantics of SITPNs, we add an extra condition to the firing of a transition: to be fired, a transition must be:

– enabled by the current marking

– must have all its conditions valuated to `true`

– must have its time counter within its time interval

– *and* must be enabled by the residual marking.

The computation of the residual marking only involves the consumption phase of the firing process; tokens are withdrawn from places, but none are generated.
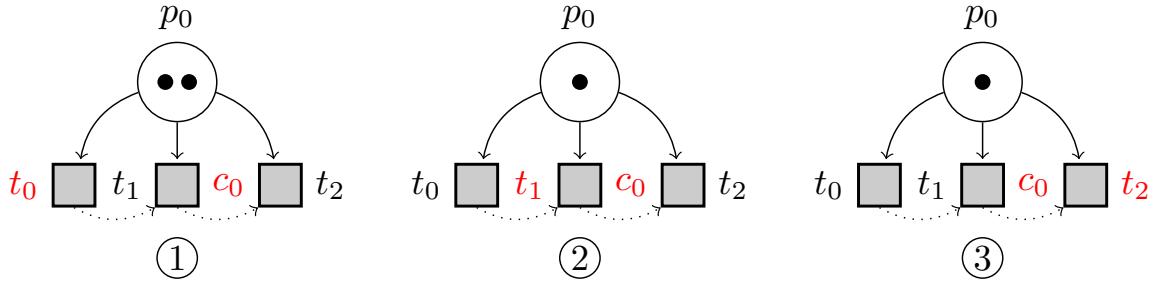
FIGURE 3.10: Computation of the residual marking for a group of conflicting transitions. At ① (resp. ② and ③), place p_0 holds the residual marking for transition t_0 (resp. t_1 and t_2). Condition c_0 is in red to indicate that its current value is `false`.

In Figure 3.10, the residual marking for t_0 corresponds to the marking obtained after the firing of all transitions with a higher priority. As t_0 is the transition with the highest firing priority, the residual marking for t_0 is equal to the current marking. Transition t_0 gathers all the conditions to be firable and is enabled by the residual marking; thus, t_0 will be fired on the next rising edge. The residual marking for t_1 is the marking obtained after the firing of t_0, i.e. the only transition with a higher priority. As illustrated at ②, t_1 is enabled by the residual marking. However, t_1 does not gather all the conditions to be firable as the value of condition c_0 is `false`. Thus, t_1 will not be fired on the new rising edge. The residual marking for t_2 is obtained after the firing of t_0 only. Even though transition t_1 has a higher firing priority than t_2, t_1 is not a member of the set of fired transitions. Thus, t_1 is not taken into account in the computation of the residual marking for t_2. The residual marking at ③ enables transition t_2, and as t_2 gathers all the conditions to be firable, then t_2 will be fired on the next rising edge.

Locked time counters

SITPNs inherit the properties of time PNs and interpreted PNs. The phenomenon of *locked* time counters is a consequence of this inheritance. As illustrated in Figure 3.11, the value of a time counter can overreach the upper bound of its associated time interval. This situation can only arise if a condition hinders the firing of a given transition while the considered transition is still enabled by the marking. As a consequence, the time counter will be incremented at every clock cycle until the upper bound of the time interval is overreached. Then, at this point, the time counter is said to be *locked* and its value will no more evolve.
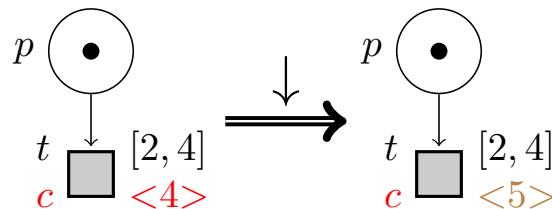
FIGURE 3.11: An example of locked time counter. Condition c is equal to `false` and thus appears in red.

In Figure 3.11, condition c is valuated to `false` before the falling edge of the clock signal. Thus, transition t can not be fired but is still enabled by the marking. On the next falling edge, the time counter of transition t is incremented and overreaches the upper bound of interval $[2, 4]$ and thus becomes locked. If the designer of the model has not anticipated the case of a locked time counter, and has not provided an alternative to disable place p in that case, then the transition t will never be firable again.

3.2 Formalization of the SITPN structure and semantics

We hope that the reader has now a fair understanding of the concepts underlying the SITPNs and of the dynamics governing the SITPN state evolution process. In this section, we give the formal definition of the SITPN structure and of its execution semantics. We also introduce the concept of a *well-defined* SITPN at the end of the section.

3.2.1 SITPN structure

The structure of SITPNs is formally defined as follows:

Definition 14 (SITPN). *A synchronously executed, extended, generalized, interpreted, and time Petri net with priorities is a tuple* $<P, T, pre, post, M_0, \succ, \mathcal{A}, \mathcal{C}, \mathcal{F}, \mathbb{A}, \mathbb{C}, \mathbb{F}, I_s>$, *where we have:*

1. $P = \{p_0, \ldots, p_n\}$, *a finite set of places.*

2. $T = \{t_0, \ldots, t_m\}$, *a finite set of transitions.*

3. $pre \in P \to T \nrightarrow (\mathbb{N}^* \times \{\texttt{basic}, \texttt{inhib}, \texttt{test}\})$, *the function associating a weight to place-transition edges.*

4. $post \in T \to P \nrightarrow \mathbb{N}^*$, *the function associating a weight and a type to transition-place edges.*

5. $M_0 \in P \to \mathbb{N}$, *the initial marking of the SITPN.*

6. $\succ \subseteq (T \times T)$, *the priority relation, which is a partial order over the set of transitions.*

7. $\mathcal{A} = \{a_0, \ldots, a_i\}$, *a finite set of continuous actions.*

8. $\mathcal{F} = \{f_0, \ldots, f_k\}$, *a finite set of functions (instantaneous actions).*

9. $\mathcal{C} = \{c_0, \ldots, c_j\}$, *a finite set of conditions.*

10. $\mathbb{A} \in P \to \mathcal{A} \to \mathbb{B}$, *the function associating actions to places.* $\forall p \in P, \forall a \in \mathcal{A}, \mathbb{A}(p, a) = \texttt{true}$, *if a is associated to p,* $\mathbb{A}(p, a) = \texttt{false}$ *otherwise.*

11. $\mathbb{F} \in T \to \mathcal{F} \to \mathbb{B}$, *the function associating functions to transitions.* $\forall t \in T, \forall f \in \mathcal{F}$, $\mathbb{F}(t, f) = \texttt{true}$, *if f is associated to t,* $\mathbb{F}(t, f) = \texttt{false}$ *otherwise.*

12. $\mathbb{C} \in T \to \mathcal{C} \to \{-1, 0, 1\}$, *the function associating conditions to transitions.* $\forall t \in T$, $\forall c \in \mathcal{C}, \mathbb{C}(t, c) = 1$, *if c is associated to t,* $\mathbb{C}(t, c) = -1$, *if* $\bar{c}$ *is associated to t,* $\mathbb{C}(t, c) = 0$ *otherwise.*

13. $I_s \in T \nrightarrow \mathbb{I}^+$, *the partial function associating static time intervals to transitions, where* $\mathbb{I}^+ \subseteq (\mathbb{N}^* \times (\mathbb{N}^* \sqcup \{\infty\}))$.

In Definition 14, the structure holds the *static* elements of a SITPN model, i.e. all the elements which value does not evolve with the execution of the model. Therefore, the value of time counters associated with transitions does not appear in the SITPN structure. As the value of time counters is *dynamic*, i.e. it evolves with the execution of an SITPN model, it is a part of the SITPN state.

Definition 15 (Time transitions). *For a given sitpn* $\in$ *SITPN, T_i denotes the definition domain of I_s, i.e. the set of transitions associated with a time interval, referred to as time transitions.*

In the current formal definition of the SITPN structure, and as discussed in Section 3.1.1, we do not consider the set of VHDL signals manipulated by a SITPN model. As a consequence, the structure holds neither the association between conditions and boolean expressions, and nor the association between actions/functions and operations (i.e. VHDL procedures that act upon signal values) that would be necessary in the presence of the set of VHDL signals. In this simplified version of the SITPN structure, conditions, actions and functions are only considered as finite sets of indexed elements associated with the places and transitions of an SITPN.

3.2.2 SITPN State

The SITPN semantics describes the evolution of the state of an SITPN through a given number of clock cycles; thus, we must first define the SITPN state structure:

Definition 16 (SITPN State). *For a given sitpn* $\in$ *SITPN, let S(sitpn) be the set of possible states of sitpn. An SITPN state $s \in S(sitpn)$ is a tuple $<M, I, reset_t, ex, cond>$, where:*

1. $M \in P \to \mathbb{N}$ *is the current marking of sitpn.*

2. $I \in T_i \to \mathbb{N}$ *is the function mapping time transitions to their current time counter value.*

3. $reset_t \in T_i \to \mathbb{B}$ *is the function mapping time transitions to time counter reset orders (defined as Booleans).*

4. $ex \in \mathcal{A} \sqcup \mathcal{F} \to \mathbb{B}$ *is the function representing the current activation (resp. execution) state of actions (resp. functions).*

5. $cond \in \mathcal{C} \to \mathbb{B}$ *is the function representing the current value of conditions (defined as Booleans).*

Notation 3 (SITPN state and fields). *In the rest of memoir, we refer to a specific field of a SITPN state s with the infix pointed notation, e.g. s.M refers to the marking of state s, and s.M(p) denotes the marking of a given place p at state s; s.I refers the function yielding the value of time counters at state s, and s.I(t) denotes the value of the time counter associated with the transition t at state s.*

At the beginning of its execution, a SITPN model is associated with an initial state defined as follows:

Definition 17 (Initial state). *For a given $sitpn \in SITPN$, $s_0 \in S(sitpn)$ is the initial state of sitpn, such that $s_0 = < M_0, O_{\mathbb{N}}, O_{\mathbb{B}}, O_{\mathbb{B}}, O_{\mathbb{B}} >$, where M_0 is the initial marking of the SITPN, $O_{\mathbb{N}}$ is a function that always returns 0, $O_{\mathbb{B}}$ is a function that always returns* `false`*.*

3.2.3   Preliminary definitions and fired transitions

Before formalizing the full SITPN semantics, we must introduce some definitions and notations, especially the definition of a *firable* and a *fired* transition. We use the two following notations to simplify the formalization of the SITPN semantics.

Notation 4 (Relations between markings). *For all relation $\mathcal{R}$ existing between two marking functions M and M', the expression $\mathcal{R}(M, M')$ is a notation for $\forall p \in P, \mathcal{R}(M(p), M'(p))$. For instance, $M' = M - \sum_{t_i \in T'} pre(t_i)$ is a notation for $\forall p \in P, M'(p) = M(p) - \sum_{t_i \in T'} pre(p, t_i)$ where $T' \subseteq T$.*

Notation 5 (Sum expressions and arc types). *Many times in this document, we need to express the number of tokens coming to or from places, after the firing of a certain subset of transitions. To do so, we use two kinds of sum expression:*

1. *The first kind of expression computes a number of output tokens. For instance, for a given place p, $\sum_{t \in T'} pre(p, t)$ where $T' \subseteq T$.*

   The expression $\sum_{t \in T'} pre(p, t)$ is a notation for $\sum_{t \in T'} \begin{cases} \omega \ if \ pre(p, t) = (\omega, \texttt{basic}) \\ 0 \ otherwise \end{cases}$

   When computing a sum of output tokens (i.e. resulting of a firing process), we want to add to the sum the weight of the arc between place p and a transition $t \in T'$ only if there exists an arc of type `basic` *from p to t (remember that the test and inhibitor never lead to the withdrawal of tokens during the firing process). Otherwise, we add 0 to the sum as it is a neutral element of the addition operator over natural numbers.*

2. *The second kind of expression computes a number of input tokens. For instance, for a given place p, $\sum_{t \in T'} post(p, t)$ where $T' \subseteq T$.*

> *The expression $\sum\limits_{t \in T'} post(p,t)$ is a notation for $\sum\limits_{t \in T'} \begin{cases} \omega \text{ if } post(t,p) = \omega \\ 0 \text{ otherwise} \end{cases}$*
>
> *Here, we add the weight of the arc from t to p only if there exists such an arc; we add 0 to the sum otherwise.*
>
> *Therefore, in the rest of the document, we will use the conciser notations $\sum\limits_{t \in T'} pre(p,t)$ to denote an output token sum, and $\sum\limits_{t \in T'} post(t,p)$ to denote an input token sum.*

We give the formal definition of the sensitization of a transition by a given marking as follows:

> **Definition 18** (Sensitization). *A transition $t \in T$ is said to be sensitized, or enabled, by a marking M, which is noted $t \in Sens(M)$, if $\forall p \in P, \forall \omega \in \mathbb{N}^*, \big(pre(p,t) = (\omega, \texttt{basic}) \vee pre(p,t) = (\omega, \texttt{test})\big) \Rightarrow M(p) \geq \omega$, and $pre(p,t) = (\omega, \texttt{inhib}) \Rightarrow M(p) < \omega$.*

We give the formal definition of a *firable* transition at a given SITPN state as follows:

> **Definition 19** (Firability). *A transition $t \in T$ is said to be firable at a state $s = <M, I, reset_t, ex, cond>$, which is noted $t \in Firable(s)$, if $t \in Sens(M)$, and $t \notin T_i$ or $I(t) \in I_s(t)$, and $\forall c \in \mathcal{C}, \mathbb{C}(t,c) = 1 \Rightarrow cond(c) = 1$ and $\mathbb{C}(t,c) = -1 \Rightarrow cond(c) = 0$.*

As explained in Section 3.1.2, the firability conditions are not sufficient for a transition to be fired. A transition must also be enabled by the residual marking to go through the firing process. Definition 20 gives the formal definition of a fired transition at a given SITPN state:

> **Definition 20** (Fired). *A transition $t \in T$ is said to be fired at the SITPN state $s = <M, I, reset_t, ex, cond>$, which is noted $t \in Fired(s)$, if $t \in Firable(s)$ and $t \in Sens\big(M - \sum\limits_{t_i \in Pr(t)} pre(t_i)\big)$, where $Pr(t) = \{t_i \mid t_i \succ t \wedge t_i \in Fired(s)\}$.*

One can notice that the definition of the set of fired transitions is recursive. To compute the residual marking necessary to the definition of a fired transition, the *Pr* set must be defined. For a given transition t, the *Pr* set represents all the transitions with a higher firing priority than t that are also fired transitions; hence the recursive definition. As the priority relation is a partial order over the finite set of transitions, all transitions have a finite set of transitions with a higher firing priority. Thus, the computation of the set of fired transitions always terminates.

In Definition 20, the marking $M - \sum\limits_{t_i \in Pr(t)} pre(t_i)$ formally qualifies the residual marking for a given transition t and at a given SITPN state s.

3.2.4   SITPN Semantics

We formalize the semantics of a given SITPN as a transition system. The SITPN state transition relation defined in the SITPN semantics has two cases of definition, one for each clock event. The SITPN state transition relation describes the evolution of the state of a SITPN.

Definition 21 (SITPN Semantics). *The semantics of a given $sitpn \in SITPN$ is the transition system $<L, E_c, \rightarrow>$ where:*

- *$L \subseteq \{\uparrow, \downarrow\} \times \mathbb{N}$ is the set of transition labels. A label is a couple (clk, τ) composed of a clock event $clk \in \{\uparrow, \downarrow\}$, and a time value $\tau \in \mathbb{N}$ expressing the current count of clock cycles.*

- *$E_c \in \mathbb{N} \rightarrow C \rightarrow \mathbb{B}$ is the environment function, which gives (Boolean) values to conditions (C) depending on the count of clock cycles ($\mathbb{N}$).*

- *$\rightarrow \subseteq S(sitpn) \times L \times S(sitpn)$ is the SITPN state transition relation, which is noted $E_c, \tau \vdash s \xrightarrow{clk} s'$ where $s, s' \in S(sitpn)$ and $(clk, \tau) \in L$, and which is defined as follows:*

  * *$\forall \tau \in \mathbb{N}, \forall s, s' \in S(sitpn)$, we have $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$, where $s = <M, I, reset_t, ex, cond>$ and $s' = <M, I', reset_t, ex', cond'>$, if:*

    (1) cond' is the function giving the (Boolean) values of conditions that are extracted from the environment E_c at the clock count τ, i.e.:

    $$\forall c \in C, \; cond'(c) = E_c(\tau, c).$$

    (2) All the actions associated with at least one marked place in the marking M are activated, i.e.:

    $$\forall a \in \mathcal{A}, \; ex'(a) = \sum_{p \in marked(M)} \mathbb{A}(p, a) \text{ where } marked(M) = \{p' \in P \mid M(p') > 0\}.$$

    (3) All the time transitions that are sensitized by the marking M and received the order to reset their time intervals, have their time counter reset and incremented, i.e.:

    $$\forall t \in T_i, \; t \in Sens(M) \wedge reset_t(t) = \texttt{true} \Rightarrow I'(t) = 1.$$

    (4) All the time transitions that are sensitized by the marking M, and did not receive a reset order, increment their time counters if time counters are still active, i.e.:

    $$\forall t \in T_i, t \in Sens(M) \wedge reset_t(t) = \texttt{false} \wedge [I(t) \leq u(I_s(t)) \vee u(I_s(t)) = \infty]$$
    $$\Rightarrow I'(t) = I(t) + 1.$$

(5) All the time transitions verifying the same conditions as above, but with locked counters, keep having locked counters (values are stalling), i.e.:

$$\forall t \in T_i, t \in Sens(M) \land reset_t(t) = \texttt{false} \land I(t) > u(I_s(t)) \land u(I_s(t)) \neq \infty$$
$$\Rightarrow I'(t) = I(t).$$

(6) All the time transitions disabled by the marking M have their time counters set to zero, i.e.:

$$\forall t \in T_i, t \notin Sens(M) \Rightarrow I'(t) = 0.$$

$* \ \forall \tau \in \mathbb{N}, \forall s, s' \in S(sitpn)$, *we have* $E_c, \tau \vdash s \xrightarrow{\uparrow} s'$, *where* $s = < M, I, reset_t, ex, cond >$ *and* $s' = < M', I, reset_t', ex', cond >$, *if:*

(7) M' is the new marking resulting from the firing of all the transitions contained in Fired(s), i.e.:

$$\forall p \in P, \ M'(p) = M(p) - \sum_{t \in Fired(s)} pre(p, t) + \sum_{t \in Fired(s)} post(t, p).$$

(8) A time transition receives a reset order if it is fired at state s, or, if there exists a place p connected to t by a `basic` *or* `test` *arc and at least one output transition of p is fired and the transient marking of p disables t; no reset order is sent otherwise:*

$$\forall t \in T_i, t \in Fired(s)$$
$$\lor \big(\exists p \in P, \omega \in \mathbb{N}^*,$$
$$[pre(p, t) = (\omega, \texttt{basic}) \lor pre(p, t) = (\omega, \texttt{test})]$$
$$\land \sum_{t_i \in Fired(s)} pre(p, t_i) > 0$$
$$\land M(p) - \sum_{t_i \in Fired(s)} pre(p, t_i) < \omega \big)$$
$$\Rightarrow reset_t'(t) = \texttt{true} \ and \ reset_t'(t) = \texttt{false} \ otherwise.$$

(9) All functions associated with at least one fired transition are executed, i.e:

$$\forall f \in \mathcal{F}, \ ex'(f) = \sum_{t \in Fired(s)} \mathbb{F}(t, f).$$

We inherit from [71] and [78], the form of Definition 21. In this thesis, we prefer to use rule instances to define execution relations, or relations that are involved in an operational semantics. Thus, Definition 21 can be equivalently represented with the following rule instances, where the premises of the rules refer to the premises of Definition 21:

$$
\begin{array}{c}
\textsc{FallingEdge} \\
\underline{\qquad (1) \qquad (2) \qquad (3) \qquad (4) \qquad (5) \qquad (6) \qquad} \\
E_c, \tau \vdash s \xrightarrow{\downarrow} s'
\end{array}
\qquad\qquad
\begin{array}{c}
\textsc{RisingEdge} \\
\underline{\qquad (7) \qquad (8) \qquad (9) \qquad} \\
E_c, \tau \vdash s \xrightarrow{\uparrow} s'
\end{array}
$$

Premises (1) to (6) describe the SITPN state evolution at the falling edge of the clock signal. Premises (1) and (2) deal with the update of condition values and the activation status of actions. Note that in Premise (2) (and also in Premise (9)), the sum expression corresponds to the Boolean sum expression, i.e. the application of the or operator over the elements of the iterated set. Premises (3), (4), (5) and (6) focus on the update of time counter values. In Premise (4) of the SITPN semantics, the *active* time counters refer to the time counters that have not yet over-reached the upper bound of their associated time interval. Of course, a time counter is always active when the upper bound is infinite. In Premise (5), the *locked* time counters refer to the time counters that have overreached the upper bound of their associated time interval. Of course, time counters can never be locked in the presence of an infinite upper bound. In Premises (4) and (5), for a given time interval i, $u(i)$ denotes the upper bound of the time interval, and $l(i)$ denotes the lower bound of the time interval.

Premises (7) to (9) describe the SITPN state evolution at the rising edge of the clock signal. Premise (7) corresponds to the marking update. The computation of the new marking uses the set of fired transitions at state s, i.e. $Fired(s)$. Premise (9) deals with the update of the function execution status. Premise (8) computes the reset orders for time transitions. There are two cases where a time transition receives the order to reset its time counter. First, if the transition is one of the fired transitions at state s, then its time counter must be reset on the next falling edge. Second, if the transition is disabled in a *transient* manner, then its time counter must also be reset. Figure 3.12 illustrates the case of a transition disabled by the *transient* marking, i.e. the marking obtained after the token consumption phase of the firing process.
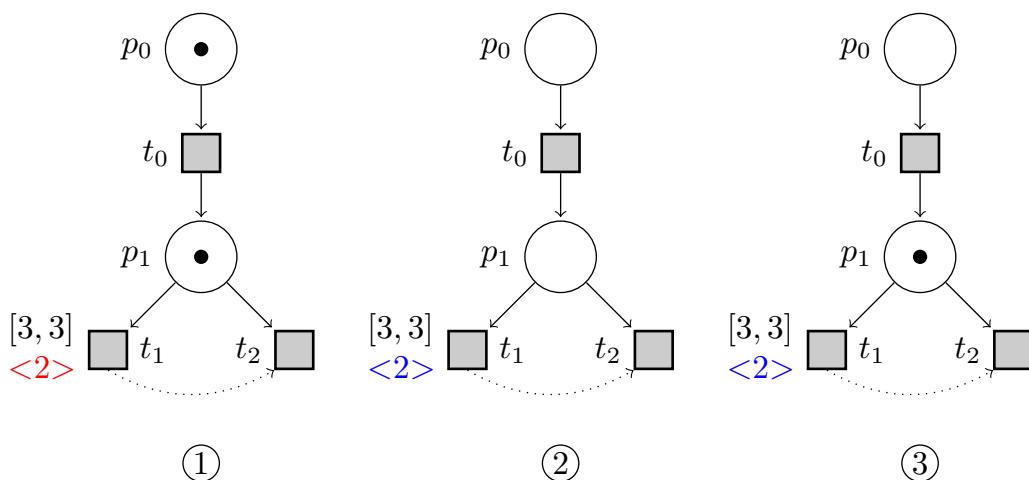
FIGURE 3.12: An example of transition that receives a reset order after being dis-
abled by the transient marking. At ①, the marking before the firing of transitions
t_0 and t_2; at ②, the transient marking; at ③, the marking at the end of the firing
process.

In Figure 3.12, the situation at ① describes the state of the SITPN before a rising edge. Given the current SITPN state at ①, transition t_0 and t_2 will be fired on the next rising edge event. Situation ② depicts the marking obtained after the consumption phase of the firing process (once the rising edge occurred), i.e. the so-called *transient* marking. Situation ③ corresponds to the marking at the end of the firing process, where t_0 and t_2 have been fired. At ③, transition t_1 is enabled by the marking. However, at ②, the transient marking disables t_1 and thus t_1 must receive a reset order (represented by a blue time counter). This reset order will be taken into account at the next falling edge event, and the time counter associated with transition t_1 will then be reset.

Contributions to the SITPN semantics

We brought the following changes to the SITPN semantics that was defined in [71] and [78]:

– We clarified the definition of the set of fired transitions. In the former SITPN semantics, four premises were dedicated to the computation of the set of fired transitions in the definition of the SITPN state transition relation on falling edge. We removed these premises from the definition, and made a *standalone* definition of the set of fired transitions that only depends on a given SITPN state (cf. Definition 20).

– We completed Premise (8) with the condition $\sum_{t_i \in Fired(s)} pre(p, t_i) > 0$. This condition is mandatory to perform the proof of semantic preservation.

– We added Premise (6) to the definition of the SITPN state relation on falling edge. This premise is also mandatory to perform the proof of semantic preservation.

3.2.5 SITPN Execution

As a part of the SITPN semantics, we define here the SITPN execution and SITPN full execution relations. These relations bind a given SITPN to an execution trace, i.e. a time-ordered list of states. This execution trace represents the successive states of the SITPN during its execution for a given number of clock cycles. These definitions are additional elements corresponding to our own contribution to the formalization of the SITPN semantics. These two relations provide a small-step semantics to the SITPNs, given that we are interested in keeping the intermediary states in an execution trace.

Definition 22 (SITPN execution). *For a given $sitpn \in SITPN$, a starting state $s \in S(sitpn)$, a clock cycle count $\tau \in \mathbb{N}$, and an environment $E_c \in \mathbb{N} \to \mathcal{C} \to \mathbb{B}$, $sitpn$ yields the execution trace θ from starting state s, written $E_c, \tau \vdash sitpn, s \to \theta$, by following the two rules below:*

EXECUTIONEND
$$\overline{E_c, 0 \vdash sitpn, s \to [\,]}$$

EXECUTIONLOOP

$$\frac{E_c, \tau \vdash sitpn, s \xrightarrow{\uparrow} s' \quad E_c, \tau \vdash sitpn, s' \xrightarrow{\downarrow} s'' \quad E_c, \tau - 1 \vdash sitpn, s'' \rightarrow \theta}{E_c, \tau \vdash sitpn, s \rightarrow (s' :: s'' :: \theta)} \tau > 0$$

The EXECUTIONEND rule states that the execution of a $sitpn \in SITPN$, starting from a state $s \in S(sitpn)$ in the environment $E_c \in \mathbb{N} \rightarrow \mathcal{C} \rightarrow \mathbb{B}$, yields an empty execution trace if the clock count comes down to 0.

The EXECUTELOOP rule describes how the execution trace related to the execution of a $sitpn \in SITPN$ is built in the case where the clock count τ is greater than zero. The final execution trace is composed of a head state s', followed by state s'' and the tail trace θ. The :: operator builds a new trace by adding a new element at the head of an existing trace. Starting from state s, $sitpn$ reaches state s' after a rising edge event; then from state s', it reaches state s'' after a falling edge event. Finally, the execution trace θ is obtained through the recursive call to the SITPN execution relation where $sitpn$ is executed during $\tau - 1$ cycles starting from state s''.

Definition 23 (SITPN full execution). *For a given $sitpn \in SITPN$, a clock cycle count $\tau \in \mathbb{N}$, and an environment $E_c \in \mathbb{N} \rightarrow \mathcal{C} \rightarrow \mathbb{B}$, $sitpn$ yields the execution trace θ starting from its initial state $s_0 \in S(sitpn)$ (as defined in Definition 17), written $E_c, \tau \vdash sitpn \rightarrow \theta$, by following the two rules below:*

FULLEXEC0

$$\frac{}{E_c, 0 \vdash sitpn \xrightarrow{full} [s_0]}$$

FULLEXECCONS

$$\frac{E_c, \tau \vdash s_0 \xrightarrow{\downarrow} s \quad E_c, \tau - 1 \vdash sitpn, s \rightarrow \theta_s}{E_c, \tau \vdash sitpn \xrightarrow{full} (s_0 :: s_0 :: s :: \theta_s)} \tau > 0$$

The FULLEXECCONS rule of the SITPN full execution relation (Definition 23) appeals to the SITPN execution relation (Definition 22). However, the definition of the SITPN full execution relation is necessary because the first cycle of execution, starting from the initial state s_0, is particular. As a matter of fact, no transitions are fired during the first rising edge. Thus, the first rising edge does not change the initial state s_0. This is why the execution trace of Rule FULLEXECCONS begins with two states s_0, thus representing the idle first rising edge.

3.2.6 Well-definition of a SITPN

To be able to transform a given SITPN into a VHDL design and also to perform the proof of semantic preservation, a SITPN must verify some properties ensuring its *well-definition*. Here, we formalize the predicate stating that a given SITPN is well-defined.

The main interest of the well-definition predicate is to prevent the phenomenon of the "double consumption" of tokens at the execution of a SITPN. In a well-defined SITPN, a conflict resolution strategy must be applied to every group of transitions in structural conflict. We must be able to decide which transition in a conflicting pair will be fired when the conflict becomes effective. Thus, we give the formal definitions of a conflicting pair of transitions and a conflict group.

> **Definition 24** (Conflict). *For a given sitpn $\in$ SITPN, two transitions $t, t' \in T$ are in conflict if there exist a place $p \in P$ and two weights $\omega, \omega' \in \mathbb{N}^*$ such that $pre(p, t) = (\omega, \texttt{basic})$ and $pre(p, t') = (\omega', \texttt{basic})$.*

A conflict group qualifies a finite set of transitions that are all in conflict with each other through at least a common input place. In Figure 3.13, the set $\{t_0, t_3, t_1\}$ is a conflict group. The formal definition of a conflict group is as follows:

> **Definition 25** (Conflict Group). *For a given sitpn $\in$ SITPN, $T_c \subseteq T$ is a conflict group if there exists a place p such that $\forall t \in T, \big(\exists \omega \in \mathbb{N}^*,\ pre(p, t) = (\omega, \texttt{basic})\big) \Leftrightarrow t \in T_c$.*

Contrary to the statement made in [71, p. 67], we no more consider the notion of conflict as being transitive. To illustrate this, Figure 3.13 shows two conflict groups: $\{t_0, t_3, t_1\}$ and $\{t_1, t_2\}$. In a well-defined SITPN (see Section 3.2.6), all conflicts in a conflict group must be considered, i.e. for all pair of transitions in the group the conflict must be solved. However, we no more consider transitions t_0 and t_2, and t_3 and t_2, as in conflict. It was believed by the author of [71] that, if no conflict resolution technique was applied between transitions in the same situation as t_0 and t_2, and t_3 and t_2, then this could result in the double-consumption of a token, or in the case where a transition is not elected to be fired even though it ought to be. However, the author does not provide an example where such a situation arises. We argue that such a situation can never arise and contrive to prove it later. Therefore, we no more consider the construction of merged conflict group (i.e, conflict groups must be merged into one if their intersection is not empty; e.g, $\{t_0, t_1, t_2\}$ in Figure 3.13) as being necessary. As a consequence, the definition of a conflict group is simpler than in [71] and does not impact the HILECOP model-to-text transformation.
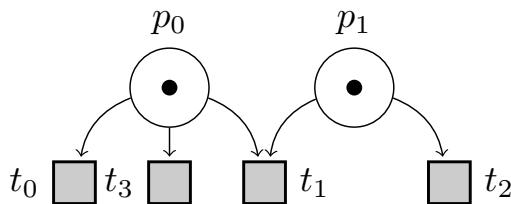
FIGURE 3.13: An example of two separate conflict groups, namely: $\{t_0, t_3, t_1\}$ and $\{t_1, t_2\}$.

When the conflict between a pair of transitions becomes effective, there are two ways to be sure that only one transition will be fired. The first way is to define a firing order through a priority relation. The second way is to use a mean of mutual exclusion. A mean of mutual exclusion ensures that the two transitions of a conflicting pair will never be firable at the same time. We only consider two ways of mutual exclusion, namely: mutual exclusion with complementary conditions and mutual exclusion with inhibitor arcs. Here, we give the formal definition of these two means of mutual exclusion.

Definition 26 (Mutual exclusion with complementary conditions). *Given two conflicting transitions t_0 and t_1, t_0 and t_1 are in mutual exclusion with complementary conditions if there exists $c \in \mathcal{C}$ such that $(\mathbb{C}(t_0,c) = 1 \wedge \mathbb{C}(t_1,c) = -1)$ or $(\mathbb{C}(t_0,c) = -1 \wedge \mathbb{C}(t_1,c) = 1)$.*

Definition 27 (Mutual exclusion with an inhibitor arc). *Given two conflicting transitions t_0 and t_1, t_0 and t_1 are in mutual exclusion with an inhibitor arc if there exists $p \in P$ and $\omega \in \mathbb{N}^*$ such that $(pre(p,t_0) = (\omega,\texttt{basic}) \vee pre(p,t_0) = (\omega,\texttt{test})) \wedge pre(p,t_1) = (\omega,\texttt{inhib})$ or $(pre(p,t_1) = (\omega,\texttt{basic}) \vee pre(p,t_1) = (\omega,\texttt{test})) \wedge pre(p,t_0) = (\omega,\texttt{inhib})$.*

Figure 3.14 illustrates the two means of mutual exclusion that can be applied to solve a conflict between two transitions.
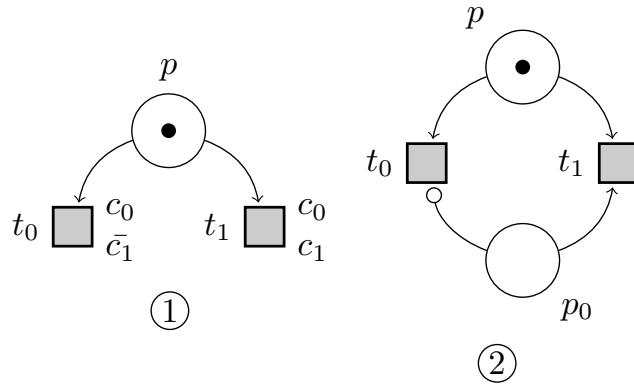
FIGURE 3.14: Examples of conflicting transitions in mutual exclusion. At ①, an example of mutual exclusion with complementary conditions; at ②, an example of mutual exclusion with an inhibitor arc.

In Figure 3.14, in situation ①, condition c_1 is associated to t_1 and the complementary condition is associated to t_0 thus creating the mutual exclusion. In situation ②, the arcs (p_0,t_0) and (p_0,t_1) ensure the mutual exclusion between transitions t_0 and t_1. Note that in the structure of mutual exclusion with an inhibitor arc, the weight of the inhibitor arc and of the one of the basic or test arc must be the same; otherwise, the mutual exclusion is not effective.

A given $sitpn \in SITPN$ is well-defined if it enforces some properties needed on the HILECOP source models before the transformation into VHDL. If the properties, given in Definition 28, are not ensured, they will lead to compile-time errors during the transformation of the SITPN into a VHDL design.

Definition 28 (Well-defined SITPN). *A given $sitpn \in SITPN$ is well-defined if:*

– *$T \neq \varnothing$, the set of transitions must not be empty.*

– *$P \neq \varnothing$, the set of places must not be empty.*

> – *There is no isolated place, i.e, a place that has neither input nor output transitions:*
> $\nexists p \in P, input(p) = \varnothing \wedge output(p) = \varnothing$, *where* $input(p)$ *(resp.* $output(p)$*) denotes the set of input (resp. output) transitions of p.*
>
> – *There is no isolated transition, i.e, a transition that has neither input nor output places:*
> $\nexists t \in T, input(t) = \varnothing \wedge output(t) = \varnothing$, *where* $input(t)$ *(resp.* $output(t)$*) denotes the set of input (resp. output) places of t.*
>
> – *For all conflict group as defined in Definition 25, either all conflicts (i.e. for all pair of transitions in the conflict group) are solved by one of the mean of mutual exclusion, or, the priority relation is a strict total order over the transitions of the conflict group.*

3.2.7   Boundedness of a SITPN

We conclude the formalization of the SITPN structure and semantics by the expression of the boundedness of a SITPN model with respect to its execution trace. In the manner of the well-definition property, the boundedness of a SITPN model is a mandatory condition to prove the semantic preservation theorem (cf. Remark 9 in Chapter 6). A SITPN model is bounded if there exists a *bound* for the number of tokens that the places can hold in the course of the execution of the model; formally:

> **Definition 29** (Bounded SITPN). *A given sitpn $\in$ SITPN is said to be bounded if for all execution environment $E_c \in \mathbb{N} \to \mathcal{C} \to \mathbb{B}$, clock cycle count $\tau \in \mathbb{N}$, execution trace $\theta \in \texttt{list}(S(sitpn))$ such that $E_c, \tau \vdash sitpn \xrightarrow{full} \theta$, then there exists a bound $k \in \mathbb{N}$ such that for all $p \in P$ and $s \in \theta, s.M(p) \leq k$.*

We extend the definition of a bounded SITPN model to a version where the bound denoting the maximal marking of each place of the model is passed through a function $b \in P \to \mathbb{N}$.

> **Definition 30** (Bounded SITPN through a maximal marking function). *A given sitpn $\in$ SITPN is said to be bounded through the maximal marking function $b \in P \to \mathbb{N}$, written $\lceil sitpn \rceil^b$, if for all execution environment $E_c \in \mathbb{N} \to \mathcal{C} \to \mathbb{B}$, clock cycle count $\tau \in \mathbb{N}$, execution trace $\theta \in \texttt{list}(S(sitpn))$ such that $E_c, \tau \vdash sitpn \xrightarrow{full} \theta$, then for all $p \in P$ and $s \in \theta, s.M(p) \leq b(p)$.*

3.3   Implementation of the SITPN structure and semantics

In this section, we present our mechanization of the SITPN structure and semantics using the Coq proof assistant. The source code is available to the reader at the address `https://github.com/viampietro/ver-hilecop`. More precisely, the implementation of the SITPN

structure and semantics is to be found under the `sitpn/dp` directory. We made a first implementation of SITPNs without the use of dependent types. For this first version, we also implemented a SITPN interpreter (a so-called *token player*) and proved that the interpreter is sound and complete w.r.t the SITPN semantics. This first implementation of the SITPNs and the formal proof of soundness and completeness are available at https://github.com/viampietro/sitpns. Here, we are only presenting the second version of the implementation of the SITPN structure and semantics, i.e. an implementation with dependent types.

3.3.1   Implementation of the SITPN and the SITPN state structure

Listing 3.2 presents the implementation of the SITPN structure as a Coq record type. The implementation is almost similar to the formal definition of the SITPN structure given in Definition 14.

```
1   Record Sitpn := BuildSitpn {

2

3     places : list nat;
4     transitions : list nat;
5     P := { p | (fun p0 ⇒ In p0 places) p };
6     T := { t | (fun t0 ⇒ In t0 transitions) t };

7

8     pre : P → T → option (ArcT ∗ ℕ∗);
9     post : T → P → option ℕ∗;
10    M₀ : P → nat;
11    Iₛ : T → option TimeInterval;

12

13    conditions : list nat;
14    actions : list nat;
15    functions : list nat;
16    𝒞 := { c | (fun c0 ⇒ In c0 conditions) c };
17    𝒜 := { a | (fun a0 ⇒ In a0 actions) a };
18    ℱ := { f | (fun f0 ⇒ In f0 functions) f };

19

20    ℂ : T → 𝒞 → MOneZeroOne;
21    𝔸 : P → 𝒜 → bool;
22    𝔽 : T → ℱ → bool;

23

24    pr : T → T → Prop;

25

26  }.
```

LISTING 3.2: Implementation of the SITPN structure in Coq.

We use lists of natural numbers, i.e. `list nat` in Coq, to define the finite sets of places (Line 3), transitions (Line 4), actions (Line 14), conditions (Line 13) and functions (Line 15) in the `Sitpn` record. We want to use these finite sets in the signature of functions appearing in the structure (e.g. use the finite set of places P in the signature of the initial marking $M_0 \in P \to \mathbb{N}$).

However, we can not use the `places` field to, for instance, give a type to the initial marking M_0. That is, we can not write $M_0 : \texttt{places} \rightarrow \texttt{nat}$, because `places` does not denote a set but an instance of lists of natural numbers. Thus, leveraging the `sig` type, we define the finite set P as the subset of natural numbers that are members of the `places` list (Line 5). We use the `In` relation defined in the Coq standard library to express the membership of a natural number regarding the elements of the `places` list. Also, the `ArcT` type (Line 8) implements the set $\{\texttt{inhib}, \texttt{test}, \texttt{basic}\}$; the `TimeInterval` type (Line 11) implements the set $\mathbb{I}^+$ of time intervals, and the `MOneZeroOne` type (Line 20) implements the set $\{0, 1, -1\}$. The priority relation is implemented by the `pr` function (Line 24) taking two transitions in parameter and projecting to the type of logical propositions, i.e. the `Prop` type.

Listing 3.3 presents the implementation of the SITPN state structure as a Coq record type.

```
1  Record SitpnState (sitpn : Sitpn) := BuildSitpnState {
2
3    M :  P sitpn → nat;
4    I :  Tᵢ sitpn → nat;
5    reset : Tᵢ sitpn → bool;
6    cond : C sitpn → bool;
7    ex :  A sitpn + F sitpn → bool;
8
9  }.
```

LISTING 3.3: Implementation of the SITPN state structure in Coq.

The `SitpnState` type definition depends on a SITPN given as a parameter; it is an example of dependent type. Projection functions are automatically generated to access the attributes of a record at the declaration of a type with the `Record` keyword. Thus, in Listing 3.3, we can refer to the set of places of `sitpn` with the term `P sitpn`. The term T_i `sitpn` denotes the set of time transitions of `sitpn`. The set of time transitions, i.e. T_i `sitpn` in Listing 3.3, for a given SITPN `sitpn` is declared as a `sig` type qualifying to the subset of transitions with an associated time interval.

3.3.2 Implementation of the SITPN semantics

Here, we present our implementation of the SITPN semantics. In Listing 3.4, we give an excerpt of the implementation of the SITPN state transition relation, i.e. the core of the SITPN semantics.

```
1  Inductive SitpnStateTransition
2    (sitpn : Sitpn) (Eₒ : nat → C sitpn → bool) (τ : nat) (s s' : SitpnState sitpn) :
3    Clk → Prop :=
4  | SitpnStateTransition_falling :
5
6      (* Premise (2) *)
7      (forall a marked sum,
8        Sig_in_List (P sitpn) (fun p ⇒ M s p > 0) marked →
9        BSum (fun p ⇒ 𝔸 p a) marked sum →
```

```
10            ex s′ (inl a) = sum) →
11
12        (* Premises (3), (4), (5) and (6) *)
13        (forall (t : Ti sitpn), ∼Sens (M s) t → I s′ t = 0) →
14        (forall (t : Ti sitpn), Sens (M s) t → reset s t = true → I s′ t = 1) →
15        (forall (t : Ti sitpn),
16            Sens (M s) t →
17            reset s t = false →
18            (TcLeUpper s t ∨ upper t = i+) → I s′ t = S (I s t)) →
19        (forall (t : Ti sitpn),
20            Sens (M s) t →
21            reset s t = false →
22            (upper t <> i+ ∧ TcGtUpper s t) → I s′ t = S (I s t)) →
23
24        (** Conclusion *)
25        SitpnStateTransition E_c τ s s′ ↓
26
27    | SitpnStateTransition_rising:
28
29        (** Premise (7) *)
30        (forall fired, IsNewMarking s fired (M s′)) →
31
32        (* Premise (9) *)
33        (forall f fired sum,
34            IsFiredList s fired →
35            BSum (fun t ⇒ F t f) fired sum →
36            ex s′ (inr f) = sum) →
37
38        (* Conclusion *)
39        SitpnStateTransition E_c τ s s′ ↑.
```

LISTING 3.4: Excerpt of the implementation of the SITPN state transition relation
in Coq.

The SITPN state transition relation is implemented in Coq as an inductive type with two constructors, i.e. one for each clock event. The relation has 6 parameters: an SITPN, an environment E_c, a clock count τ, two SITPN states s and s′ and a clock event. Note that the two states s and s′ are bound to the SITPN parameter through their type, i.e. `SitpnState` sitpn.

In the construction case `SitpnStateTransition_falling`, we give the implementation of Premises (2), (3), (4), (5) and (6) defined in the SITPN semantics. The sum term of Premise (2), i.e. $\sum_{p \in marked(M)} \mathbb{A}(p, a)$, is implemented by Lines 8 and 9. At Line 8, the `Sig_in_List` predicate states that all the inhabitant of the P sitpn type (i.e. the places of sitpn) that verify the property (`fun p ⇒ M s p > 0`) (i.e. the marking of a place is greater than zero at state s) are members of the `marked` list. Because we cannot iterate over the elements of a given sig type, we use the `Sig_in_List` relation to convert a sig type into a list. Lists are iterable by definition. At Line 9, the `BSum` relation states that sum is the Boolean sum obtained by applying the function

(`fun p ⇒ A p a`) to the elements of the `marked` list. Premises (3), (4), (5) and (6) are almost similar in their implementation to the description of Definition 21. The Coq term `Sens (M s) t` implements the term $t \in Sens(M)$. Due to the particular nature of the upper bound of a time interval, i.e. defined over the set $\mathbb{N}^* \sqcup \{\infty\}$, the test that the current time counter of a given transition t is less than or equal to the upper bound is implemented by a separate predicate `TcLeUpper`. Similarly, the `TcGtUpper` predicate implements the inverse test.

In the construction case `SitpnStateTransition_rising`, we give the implementation of Premises (7) and (9) defined in the SITPN semantics. In the implementation of Premise (7), the `IsNewMarking` predicate represents the expression:

$$\forall p \in P,\ M'(p) = M(p) - \sum_{t \in Fired(s)} pre(p,t) + \sum_{t \in Fired(s)} post(t,p).$$

In its definition, the `IsNewMarking` predicate first checks that the `fired` list implements the set of fired transitions at state `s`. Then, it builds the marking at state `s'` for each place p, i.e. (`M s'`), by consuming and producing a number of tokens starting from the marking of p at state `s`. The `fired` list is helpful to qualify the input token sum and the output token sum for a given place. Similarly to the implementation of Premise (2), the implementation of Premise (9) at Line 33 relies on the `BSum` predicate to compute the Boolean sum $\sum_{t \in Fired(s)} \mathbb{F}(t,f)$. The term `IsFiredList s fired` states that the `fired` list implements the set of fired transitions at state `s`, so we can use the `fired` list to compute the above sum.

3.4 Conclusion

The class of SITPNs is a particular class of PNs used to model the behavior of components in the HILECOP high-level models. The synchronous evolution of SITPNs constitutes the originality of the model compared to the standard PNs semantics. In this chapter, we gave an informal and formal presentation of SITPNs and their execution semantics. Two previous Ph.D. theses contributed, for the most part, to the formalization of the SITPN structure and semantics. However, we helped simplify the semantics of SITPNs. We passed from 14 rules in the definition of the SITPN semantics given in [71] to 9 rules in our current definition of semantics. Also, as presented in at the end of Section 3.2.4, we completed some rules when they happened to be insufficient to prove the theorem of behavior preservation. Finally, we defined the execution relations for the SITPN semantics and formalized the well-definition property for the SITPN structure.

Our other contribution was to implement the SITPN structure and semantics with the Coq proof assistant. There are two implementations: one with and one without dependent types. For the version without dependent types, we implemented a SITPN interpreter or token player. We also proved a soundness and completeness theorem between the interpreter and the formalized SITPN semantics. The first implementation of the SITPNs in Coq represents 5000 lines of specification and 7000 lines of proof. The second implementation, which was presented in Section 3.3, leverages dependent types. This implementation is closer to the formal definition given in Definition 14. We chose this implementation to mechanize the proof of the behavior preservation theorem (see Chapter 6).

Chapter 4

$\mathcal{H}$-VHDL: a target hardware description language

The main purpose of this chapter is to present the target language of the HILECOP transformation, i.e. the VHDL language. The formalization and the implementation of the VHDL language syntax and semantics is mandatory to reason about the programs generated by the HILECOP model-to-text transformation. Thus, we want the reader to clearly understand the structure and the semantics of the language to be able to fully grab the proof of semantic preservation presented in Chapter 6. Specifically, we present here the $\mathcal{H}$-VHDL language, our own synthesizable subset of the VHDL language. This subset permits to encode the programs generated by the HILECOP transformation. We devise a formal semantics for $\mathcal{H}$-VHDL which is a simplification of the simulation semantics of the VHDL language. The formalization of the $\mathcal{H}$-VHDL semantics and its implementation is one contribution of this thesis. The chapter is structured as follows. In Section 4.1, we give an informal presentation of the VHDL language syntax and semantics. In Section 4.2, we present the state of the art pertaining to the formalization of the VHDL language semantics. In Section 4.3, 4.4, 4.5 and 4.6, we give the full formalization of the $\mathcal{H}$-VHDL language, a subset of the VHDL language. Section 4.7 illustrates the formal $\mathcal{H}$-VHDL semantics with an example. Finally, Section 4.8 outlines the implementation of the $\mathcal{H}$-VHDL syntax and semantics with the Coq proof assistant.

The HILECOP transformation generates a VHDL design implementing an input SITPN model. The transformation generates and connects the component instances of two previously defined VHDL designs: the place design, i.e. a VHDL implementation of a SITPN place, and the transition design, i.e. a VHDL implementation of a SITPN transition. These designs were defined by the INRIA CAMIN team at the creation of the HILECOP methodology. In the following sections, we will be using fragments of the definition of the place and transition designs to illustrate the content of VHDL programs and the rules of the VHDL language semantics. The reader will find the source code of the place and transition designs in concrete and abstract syntax in Appendices A and B.

4.1 Presentation of the VHDL language

The intent here is to give an overview of the VHDL language, its purpose, its main syntactic constructs, and an informal description of its semantics as presented in the Language Reference

Manual (LRM) [64]. The VHDL language offers a lot of possibility in terms of hardware (and even software) description. Here, we are not trying to be exhaustive in our presentation of the language. We will only maintain our description of the VHDL concepts in the scope that is of interest to us. The readers that are interested in learning more about the VHDL language can refer to [64], [5] and [92].

4.1.1 Main concepts

The VHDL acronym stands for Very high speed integrated circuit Hardware Description Language. The main purpose of the VHDL language is to describe hardware circuits.

A top-level VHDL program is called a *design*. A VHDL design is composed of two descriptive parts. The first part is called the entity and describes the interfaces of a circuit, namely: the input and output ports, and the generic constants. Listing 4.1 is an excerpt of the `transition` design's entity that defines the generic constants, the input and output port interfaces of the design. The generic clause of the entity holds the declaration of the generic constants. Figure 4.1 is a visual representation of the interfaces of the `transition` design.

The purpose of generic constants is either to represent some dimensions of the design (e.g. the size of ports, internal signals...) or to represent constant values used throughout the design. In Listing 4.1, one can see that the `conditions_number` generic constant gives a dimension to the type of the `input_conditions` input port, which is an array of Boolean values with indexes ranging from 0 to `conditions_number-1` (that is the meaning of "`std_logic_vector (conditions_number-1 downto 0)`"). The port clause holds the declaration of input and output ports of the design. The `in` keyword indicates the declaration of an input port and the `out` indicates the declaration of an output port.

```
1  entity transition is
2    generic (
3      transition_type : transition_t := NOT_TEMPORAL;
4      input_arcs_number : natural := 1;
5      conditions_number : natural := 1;
6      maximal_time_counter : natural := 1
7    );
8    port (
9      clock : in std_logic;
10     reset_n : in std_logic;
11     input_conditions : in std_logic_vector(conditions_number−1 downto 0);
12     time_A_value : in natural range 0 to maximal_time_counter;
13     time_B_value : in natural range 0 to maximal_time_counter;
14     input_arcs_valid : in std_logic_vector(input_arcs_number−1 downto 0);
15     reinit_time : in std_logic_vector(input_arcs_number−1 downto 0);
16     priority_authorizations : in std_logic_vector(input_arcs_number−1 downto 0);
17     fired : out std_logic
18   );
19 end transition;
```

LISTING 4.1: The entity part of the transition design in concrete VHDL syntax.
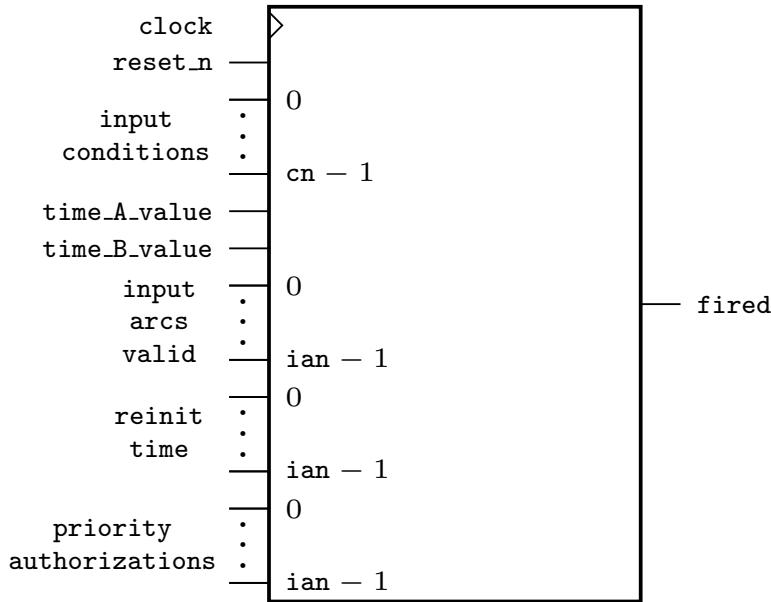
FIGURE 4.1: A representation of the `transition` design entity. On the left side, the input port interface of the `transition` design; cn stands for `conditions_number` and ian stands for `input_arcs_number`, i.e. two of the generic constants declared in the generic clause of the `transition` design entity; the numbers at the right of the input pins represent the pin indexes. On the right side, the output port interface of the `transition` design.

The second part of a VHDL design is called the architecture. The architecture describes the internal behavior of the design. It declares all the internal signals, i.e. the wires, involved in the description of the design behavior. Then, there are three ways to describe the behavior itself: by using processes, by instantiating other designs (also called, component instantiations), or by combining both techniques (the latter option is chosen in the VHDL designs generated by the HILECOP transformation).

Behavior specification with processes

The first way to specify the behavior of a design is through the description of processes. Processes are concurrent statements that describe the wiring or the operations performed on the signals of a given design. These operations are described by sequential statements in the body of processes. A process declares a sensitivity list that corresponds to the signals read in its statement body; also, it possibly declares internal variables. Listing 4.2 gives an excerpt of the `transition` design architecture containing the declarative part of the architecture (i.e. the declaration of internal signals) and three of the processes describing the `transition` design behavior, namely: the `condition_evaluation` process, the `firable` process and the `fired_-evaluation` process. In Listing 4.2, Lines 2 to 8 correspond to the declaration of the internal signals of the `transition` design. Line 11 starts the declaration of the `condition_evaluation` process. The sensitivity list of the `condition_evaluation` process holds one signal, the `input_-conditions` input port. The value of the `input_conditions` input port is read is the process

body; then, as a design rule, it must be declared in the sensitivity list. The process defines a local variable `v_internal_condition` at Line 12. At Line 13, the `begin` keyword starts the declaration of the process body, i.e. the block of sequential statements performing operations on the signals of the `transition` design.

```vhdl
1  architecture transition_architecture of transition is
2     signal s_condition_combination : std_logic;
3     signal s_enabled : std_logic;
4     signal s_firable : std_logic;
5     signal s_firing_condition : std_logic;
6     signal s_priority_combination : std_logic;
7     signal s_reinit_time_counter : std_logic;
8     signal s_time_counter : natural range 0 to maximal_time_counter;
9  begin
10
11    condition_evaluation : process (input_conditions)
12       variable v_internal_condition : std_logic;
13    begin
14       v_internal_condition := '1';
15
16       for i in 0 to conditions_number − 1 loop
17          v_internal_condition := v_internal_condition and input_conditions(i);
18       end loop;
19
20       s_condition_combination ⇐ v_internal_condition;
21    end process condition_evaluation;
22
23    ...
24
25    firable : process (reset_n, clock)
26    begin
27       if (reset_n = '0') then
28          s_firable ⇐ '0';
29       elsif falling_edge(clock) then
30          s_firable ⇐ s_firing_condition;
31       end if;
32    end process firable;
33
34    fired_evaluation : process (s_firable, s_priority_combination)
35    begin
36       fired ⇐ s_firable and s_priority_combination;
37    end process fired_evaluation;
38
39  end transition_architecture;
```

LISTING 4.2: An excerpt of the architecture part of the transition design in concrete VHDL syntax.

In the statement body of a process, the designer can use control flow statements common to most of the generic programming languages (if statement, for loops...), and also statements that are specific to the VHDL language. The most representative statement, and the one of interest to us, is the *signal assignment* statement. The signal assignment statement relates a given signal identifier to a source expression. For instance, at Line 20 of Listing 4.2, the signal assignment statement, represented with the $\Leftarrow$ operator, assigns the value of the internal variable v_-internal_condition to the s_condition_combination signal. The v_internal_variable that itself holds the Boolean product between the subelements of the input_conditions input port performed in the for loop of Lines 16 to 18.

When considering a VHDL design in the point of view of hardware synthesis, a signal assignment statement specifies a wiring between a target signal identifier and other source signals. Figure 4.2 gives a synthesis-oriented view of the processes described in Listing 4.2.
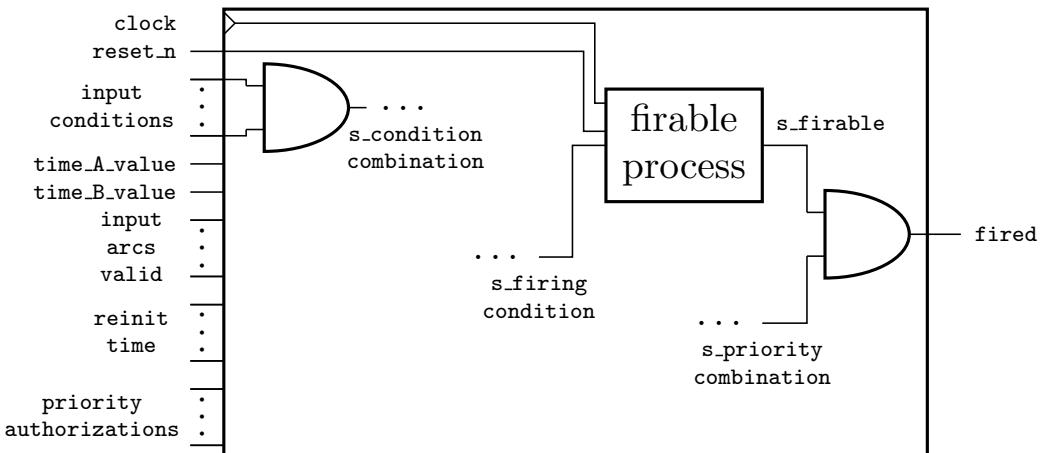
FIGURE 4.2: A representation of three of the processes defining the transition design architecture. On the left side, the condition_evaluation process connecting the input_conditions input port to the s_condition_combination internal signal; the firable process in the middle; on the right side, the fired_evaluation process connecting the s_firable and the s_priority_combination signals to the fired output port.

In Figure 4.2, the condition_evaluation process is represented as an "and" port performing the product over the elements of the input_conditions input port. The fired_evaluation process is a simple "and" gate connecting the fired output port to the s_firable and s_-priority_combination internal signals. The fired_evaluation and the condition_evaluation processes are combinational processes. They describe the value of an output signal based on the value of input signals. For instance, the value of the s_condition_combination signal is a function of the value of the input_conditions input port such that:

$$\texttt{s\_condition\_combination} = \prod_{i=0}^{\texttt{conditions\_number}-1} \left(\texttt{input\_conditions}[i] \right)$$

This equation always holds, and we refer to it as a combinational equation. Here, the `input_conditions` input port is a composite signal, meaning that it is composed of multiple subelements (multiple pins) each having a unique index. We denote the value of the subelement of index i in a composite signal a with the square-bracketed notation $a[i]$.

Also presented in Figure 4.2, the `firable` process is a *synchronous* process. It is executed only at the occurrence of the falling edge event of the `clock` signal, and thus represents a *memory* point. In its statement body (Line 30 of Listing 4.2), the `firable` process assigns the value of the internal signal `s_firing_condition` to the signal `s_firable` only at the occurrence of the falling edge of the `clock` signal (captured by the expression `falling_edge(clock)` where `falling_edge` is a primitive function of the VHDL language). In the point of view of simulation, there are no distinction between synchronous processes and *combinational* processes. However, in the point of view of synthesis, processes responding to a `clock` signal follow the rules of the synchronous (or sequential) logic, whereas, combinational processes follow the rules of combinational logic.

To complete the presentation of the statements to be found in the body of processes, the VHDL language is also equipped with `timing` constructs, i.e. statements that explicitly specify an amount of time in a given time unit. The signal assignment statement possibly specifies a time clause indicating when the assignment must be performed. For instance, the signal assignment statement specifying that the value of signal `b` must be assigned to signal `a` in 3 milliseconds takes the form: $a \Leftarrow b$ `in 3 ms`. When no time clause is specified for a signal assignment statement, we talk about a δ-delay signal assignment, i.e. the application of the signal assignment is related to some δ interval corresponding the time of propagation through a wire. When a time clause is specified, we talk about a unit-delay signal assignment. δ-delay signal assignments are synthetizable, meaning they have an equivalent implementation on a physical device, whereas, unit-delay signal assignments can not be synthetized. Unit-delay signal assignments do not appear neither in the VHDL designs generated by HILECOP transformation nor in the declaration of the `place` and `transition` designs. We are only mentioning their existence because they play a part in the simulation algorithm described in Section 4.1.2.

Behavior specification with design instances

The second way to specify the behavior of a design is to use other designs, or rather instances of other designs, as components. In that case, the design is said to be composite as it embeds instances of other designs in its own behavior. Also, a design at the highest level of embedding, i.e. that is not instantiated as a part of another design's behavior, is called a *top-level* design. The design instantiation, or component instantiation, statement permits to instantiate a design in an embedding architecture. When instantiating a design with a design instantiation statement, the designer provides the component instance with an identifier. Then, the instance must be dimensioned; this is performed through a generic map that associates the generic constants of the design being instantiated to a static value. Finally, the designer specifies how the component instance is connected to the other elements of the architecture. A port map associates the input ports and output ports of the component instance to expressions or to the signals of the embedding architecture. Listing 4.3 shows an example of instantiation of the HILECOP's

`transition` design. This instance is involved in the definition of the behavior of an embedding design called `toplevel`.

```
1   architecture toplevel_architecture of toplevel is
2   begin
3     ...
4     id_t : entity transition
5     generic map (
6       transition_type ⇒ NOT_TEMPORAL,
7       input_arcs_number ⇒ 1,
8       conditions_number ⇒ 1,
9       maximal_time_counter ⇒ 1
10    )
11    port map (
12      clock ⇒ clock,
13      reset_n ⇒ reset_n,
14      time_A_value ⇒ 0,
15      time_B_value ⇒ 0,
16      input_conditions(0) ⇒ id_0,
17      input_arcs_valid(0) ⇒ id_1,
18      priority_authorizations(0) ⇒ '1',
19      reinit_time(0) ⇒ id_2,
20      fired ⇒ id_3
21    );
22    ...
23  end toplevel_architecture;
```

LISTING 4.3: An example of design instantiation statement in the architecture of the `toplevel` design. Here, the design being instantiated is the transition design.

In Listing 4.3, the `transition` component instance (TCI) has the identifier id_t. Following the `entity` keyword is the name of the design being instantiated; here, the `transition` design. Then, the generic map associates the generic constants of the transition design (i.e. the left side of the arrow, also called the *formal* part) to static values (i.e. the right side of the arrow called the *actual* part). This permits the dimensioning of the component instance. For example, remember that the `input_arcs_number` generic constant value determines the number of elements in the composite input ports `input_arcs_valid`, `priority_authorizations` and `reinit_time` (cf. Figure 4.1). The port map associates the input ports of the `transition` design to expressions. For instance, the `time_A_value` input port is connected to the constant value 0, and the `input_conditions` input port is connected to the internal signal id_0 at index 0. The port map also associates the output ports with signal identifiers. Contrary to the association of input ports, output ports can not be associated to expressions. An output port association describes a direct wiring. In the port map described in Listing 4.3, the association `fired` $\Rightarrow id_3$ expresses that the `fired` output port is connected to the signal id_3, where signal id_3 is defined in the embedding design. Figure 4.3 illustrates the `transition` component instance id_t and the wiring of its input and output port interfaces inside the `toplevel` design.
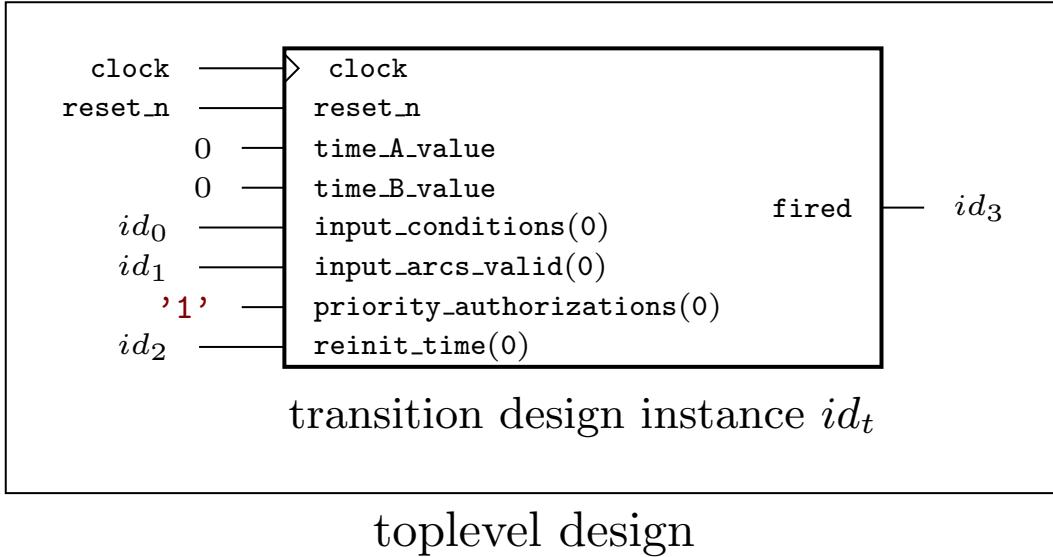
FIGURE 4.3: Visual representation of a design instantiation statement. Here, the figure represents the transition design instance described in Listing 4.3.

4.1.2   Informal semantics of the VHDL language

There are two approaches to the description of circuits with the VHDL language. The first aims at the simulation of the described circuits, and the second aims at the synthesis of the described circuits on physical supports. These two approaches arise from the practice and the use of the VHDL language by electronicians. Even though, in practice, there are two ways to consider a VHDL design, i.e. a synthesis-oriented way and a simulation-oriented way, the LRM does not define a synthesis-oriented semantics for the VHDL language. A synthesis-oriented semantics gives an interpretation to a design by describing an equivalent in a lower level formalism, closer to the physical circuit. For instance, the Verilog language gives a synthesis-oriented semantics to its programs by defining an equivalent RTL level description [66]. The LRM gives an informal semantics to VHDL designs through the definition of a simulation algorithm [64, p.167]. The purpose of simulation is to compute the evolution of the values of signals during a certain time interval. Through the simulation process, the designer is able to control the behavior of the modeled circuits and to detect flaws in the evolution of the signal values.

Former to the simulation, the LRM defines an elaboration phase. The elaboration phase operates syntactic and semantic controls over the design code. It also describes code transformations over the design's behavioral part to obtain a simulation-ready execution model. More specifically, the elaboration phase builds the simulation environment and the default simulation state associated with the design under simulation. The simulation environment is built based on the declarative parts of the top-level design; it maps the signals to their types. In the default simulation state, each signal is associated with a current value (i.e. the default value of the signal's type) and with a driver. A driver maps time points to values and the association between a given time point and a signal value is called a transaction. The need for drivers to express the values associated with a given signal is explained by the presence of unit-delay

signal assignments. A unit-delay signal assignment specify a time clause indicating when a giving assignment must be performed, e.g. $a \Leftarrow b$ in 3ms (signal a takes the value of signal b in 3 milliseconds). Thus, when a unit-delay signal assignment is executed in the course of a simulation, its effect is to change the driver of the target signal by posting a new transaction. For instance, let T_c by the current simulation time, the execution of statement $a \Leftarrow$ true in 2ns sets a new transaction in the driver of signal a. The new transaction associates the value true to the time point $T_c + 2$ns. Note that without unit-delay signal assignments, i.e. without a specified time clause, drivers are not needed as all assignments take effect at the current simulation time. Moreover, the elaboration checks the well-formedness of the design by performing static type-checking on the behavioral part of the design. It also checks that the connection between signals respect certain rules, for instance, that there are no multiply-driven signals, i.e. signals that are written to by multiple processes. Finally, the elaboration operates some transformations over the VHDL code, and thus builds the *execution* model. To summarize, all concurrent statements of the behavioral part are transformed until the top-level design behavior is only composed of processes.

After the elaboration, the top-level design, or rather its corresponding execution model, is ready to be simulated. Two entities are involved in the simulation: the *sea* of processes obtained after the elaboration of the top-level design's behavior, and a *kernel* process. The kernel process orchestrates the simulation; it handles the time of the simulation, i.e. it holds a variable describing the current time of the simulation, and controls the execution of processes. Figure 4.4, which is an excerpt from [15], describes the structure of the VHDL simulation algorithm.
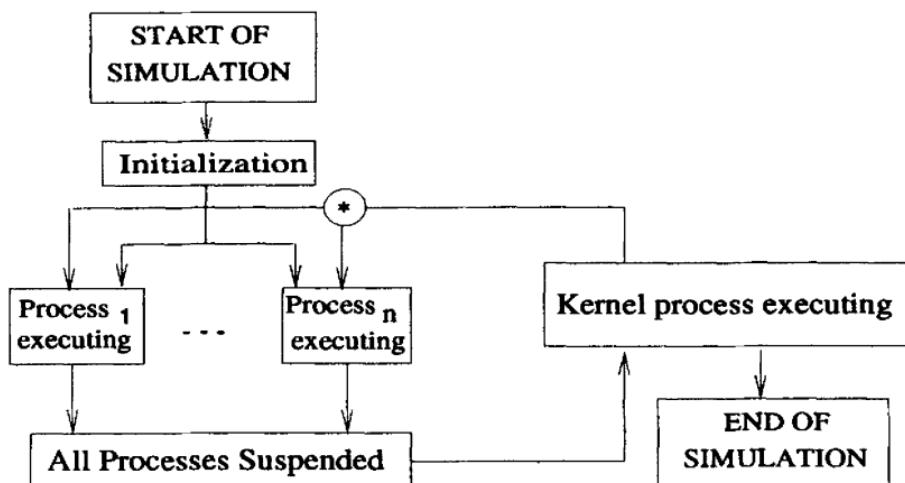
FIGURE 4.4: The VHDL simulation loop. Excerpt from [15].

The simulation starts with an initialization phase. During the initialization phase all processes are run exactly once. Then, the simulation cycles are structured as follows. All processes execute their statement body concurrently. New transactions are posted in the drivers of signals for every interpreted signal assignment statement. The execution goes on until all processes have executed their statement body and then have reached a suspension state. When, all processes are suspended, the kernel process takes over. Figure 4.5 shows the activity diagram associated with the kernel process.
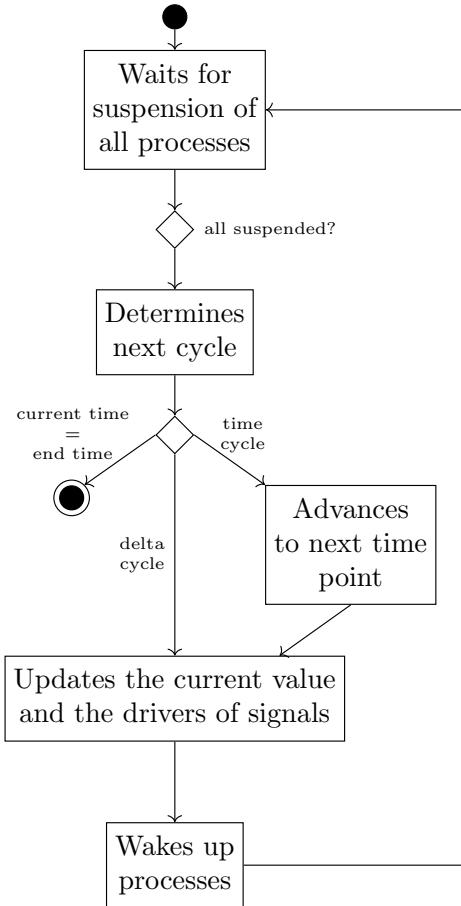
FIGURE 4.5: The activity diagram of the kernel process. Square boxes represent
activities, diamond nodes are decision nodes. The black circle at the top represents
the starting point of the activities; the other black circle in the middle of the diagram
represents the end of all activities.

As shown in Figure 4.5, after the suspension of all processes, the kernel process will then
determine the kind of simulation cycle that will be performed next. There are two kinds of
cycles: delta cycles or time cycles. If the value of a signal changes at the current time point,
i.e. its driver holds a transaction at the current time point with a new value, then a delta cycle
must be performed. Then, the simulation time does not change. The kernel process updates
the current value of signals and their drivers, and wakes up the processes sensitive to the
signals that obtained new values. The repetition of multiple delta cycles corresponds to the
stabilization of signal values, i.e. the propagation of values through the wires, that takes effect
in a negligible δ time. If all signal values are stable at the current time point, then a time cycle
must be performed. The kernel process looks up the drivers for the next time point where the
value of a given signal will change. Then, the kernel process advances the simulation time to
this next time point before updating the signal values and resuming the execution of processes.
The simulation goes on like this, alternating between delta and time cycles, until the current
time value reaches the time specified for the end of the simulation.

4.2 Choosing a formal semantics for VHDL

In the previous section, we presented the main concepts underlying the VHDL language and its informal semantics. We want to prove that the HILECOP transformation that generates VHDL code from SITPNs preserves the behavior of the initial model (i.e, the SITPN model) into the generated VHDL program. A formal semantics for the VHDL language is therefore a necessary element to be able to reason about the generated VHDL programs, and moreover to be able to compare their behaviors with the behaviors of the source SITPN models. Keeping that in mind, which formal semantics should we consider for VHDL?

The same holds for any task: there is a trade off between finding a tool designed by others that will fit our needs, and creating our own tool that will mitigate the gaps between our needs and what is available in the literature. In the present case, the tool is a formal semantics for VHDL. Adopting a fully-set semantics found in the literature as a base ground for the implementation of a formal semantics for VHDL has multiple perks. First, it reduces the formalization effort, which is not a lesser point considering that the proof ahead might be long and must still be completed within the time span of the thesis. Still, the semantics would need to be implemented in Coq, if no implementation exists (or not written in Coq). Second, the formal semantics of programming languages found in the literature are often general in their approach, this to provide a generic framework to reason about programs. However, we must not lose sight of our goal which is to prove behavior preservation; a generic formal semantics could turn out to be too complex, or necessitate too much tweaking and thus hinder the fulfillment of our task. On the other side, creating our own formal semantics for VHDL, based on the work of others, is the best way to fit our needs in compliance with our final aim. However, the pitfalls are that the resulting semantics might prove to be very specific, therefore preventing others from using it. Also, a work of formalization would be necessary which, as we already stated, would be time-consuming. In order to determine whether we ought to use an existing semantics or design a new one, we must first clearly specify our needs regarding the VHDL language.

4.2.1 Specifying our needs: HILECOP and VHDL

Two elements are of major influence to the specification of our needs for a formal semantics: first, the context of HILECOP and the specificities of the VHDL programs that are generated; second, the context of theorem proving. These two aspects entail the following considerations.

The need for coverage

The HILECOP methodology generates particular VHDL programs. Even if some transformations can be operated on the generated programs to simplify them, the looked-for formal semantics must be able to deal with a certain subset of the VHDL language. Especially, this subset must include:

– 0-delay (or δ-delay) signal assignments (equivalent to unit-delay signal assignment with a "0 ns" after clause)

– component instantiation statements with generic constant and port mapping

– entity's generic constant clauses (declaration of generic constants in a design entity)

HILECOP's VHDL programs only deal with 0-delay signal assignments because they are the only kind of signal assignments that can be synthesized. As a matter of fact, the industrial compiler/synthesizer used in the HILECOP methodology only accepts VHDL programs with no timing constructs (i.e, no delayed signal assignments) as inputs.

Regarding component instantiation statements, the VHDL LRM describes a way to transform these statements into equivalent process statements and block constructs [64, p. 141] during the elaboration of the design. However, we want to preserve the hierarchical structure provided by the component instantiation statements arguing that it will be easier to compare the state of a given SITPN model with a VHDL design state with an explicit hierarchical structure. Indeed, there exists a mapping between places and transitions of an SITPN and their mirror (generated by the transformation) place and transition component instances (PCIs and TCIs). This one-to-one correspondence might turn out to be handy to perform the proof of behavior preservation. Obviously, the semantics must cover the evaluation of process statements which are the core concurrent statements of VHDL programs.

The types of signals and variables used in HILECOP VHDL designs must have finite ranges of values. For instance, a VHDL signal that ranges over $\mathbb{N}$ cannot be synthesized on a physical circuit. Indeed, $\mathbb{N}$ has an infinite number of values, and would therefore require an infinite number of latches to be physically implemented. Moreover, as the number of latches used to implement a digital circuit greatly impacts the power consumption of the circuit, the types of signals and variables must be as constrained as possible to optimize the dimensioning of the circuit. The generic constants, declared in the entity part of a design, are involved in the dimensioning of the circuit. The generic constants define the bound of the array and natural range types for the different signals and variables declared in the `place` and `transition` designs' architecture. When a place or a transition component is instantiated, that is during the transformation of the SITPN model into VHDL code, its generic constants receive values via a generic map; we call it the dimensioning of the component instance. Therefore, generic constant clauses must belong to the subset of the VHDL language covered by the semantics.

The need for a synchronous execution

The second property of HILECOP's generated VHDL programs is their synchronous execution. The digital circuits designed with the HILECOP methodology are all synchronously executed on physical target. The generated VHDL designs declare a clock signal as an input port of their entity port interface. Thus, the behavioral part of the designs contains two kinds of processes: *synchronous* processes, i.e. processes that are sensitive to the clock signal, and *combinational* processes, i.e. processes that are not sensitive to the clock signal, and that are permanently running until the stabilization of the signal values. Synchronous processes react to the events of the clock signal, i.e. the rising and the falling edge, and possess blocks of sequential statements that are only executed at the precise moment of the clock event[1]. Therefore, we need a

---

[1]These blocks are guarded by the expressions `rising_edge(clk)` and `falling_edge(clk)`.

semantics that is able to deal with synchronism, and that explicitly integrates the synchronization with a clock signal into the expression of the simulation cycle.

A last consideration pertains to whether or not the VHDL semantics must explicitly handle errors. As the SITPN semantics does not include the production of error values, the handling of errors by the VHDL semantics is not a mandatory aspect.

Qualifying criterions

We here give the list of the qualifying criterions that will help to analyze the different VHDL semantics encountered in the literature and that are presented in the next section. The three most relevant criterions are:

– *Synchronism.* Regarding this criterion, there are three possibilities:

  – Synchronism is not expressible in the considered VHDL semantics; this completely disqualifies the adoption of the semantics.

  – Synchronism is expressible in the considered VHDL semantics. Synchronism is expressible if time-steps are handled in the semantics, at least to be able to represent clock events.

  – Synchronism is explicit, i.e. the simulation loop is built around the occurrences of clock events.

  We will foster the semantics that explicitly formalize a synchronized execution of a VHDL design.

– *Component instantiation.* Either the semantics handle the component instantiation statement in its simulation rules, or component instantiation statements must be transformed in order to be executed. We will foster the semantics that handle component instantiation statements without transformation.

– *Elaboration.* This criterion pertains to the formalization of the elaboration phase as integrated to the VHDL semantics. This criterion also expresses the ability of the semantics to handle constrained types, i.e. arrays and natural ranges, and generic constant clauses that are both dealt with during the *elaboration* phase. Either the semantics handle these constructs or it does not. Of course, we will foster the first kind of semantics.

4.2.2 Looking for an existing formal semantics

Here, we give a summary of the work found in the literature pertaining to the formalization of the VHDL language semantics. Articles are gathered and presented depending on the type of semantics used in the formalization (operational, denotational, axiomatic...). Each semantics is analyzed regarding the needs that were previously expressed.

Denotational semantics

Some authors have been interested in giving a formal denotational semantics to VHDL. In a general manner, these authors want to reason about VHDL programs: prove properties over a VHDL program, prove that two programs are equivalent…

In [51], the authors give a denotational semantics to the VHDL language within the Focus [41] framework, a method for the development of distributed systems. Signal values and their evolution through time are represented as streams of values. Statements are denoted as stream-processing functions. Processes are stream-processing functions that take input signal streams (signals of the sensitivity list) and yields transaction traces (i.e, waveforms) over output signals (i.e, signal that are written by the process). Transaction traces are merged together as the result of the concurrent execution of processes. The authors only consider 0-delay signal assignments in their semantics, stating that it is sufficient to "consider time at a logical level to model both synchronous and asynchronous designs". However, some transformations must be applied to a design that has a synchronous execution to express its equivalent only with 0-delay signal assignments. Therefore, this semantics does not express synchronism of execution in an explicit manner. Moreover, the component instantiation statements are not dealt with, and no mention is made of the elaboration phase.

In [24], the authors give a denotational, yet relational, semantics to the VHDL language. A state of a VHDL design is represented by a function binding signals to values; a worldline is a time-ordered list of states. Statements (including processes) are denoted in the semantics by a relation that binds an input couple, composed of a time point and a worldline, to an output couple of the same type. Multiple input and output couples possibly satisfy the relation denoting a particular statement; thus, the semantics is nondeterministic. The semantics tries to abstract from the formalization of the simulation cycle as it is done in the LRM. The authors want to establish a semantics that is abstract enough to be able to compare all other works of formalization with the authors' semantics. The authors also give an axiomatic semantics (i.e, in the Hoare logic style) which is proved to be sound and complete with the first denotational semantics. A Prolog [33] implementation of the axiomatic semantics is given. Regarding our needs, the semantics only deals with unit-delay signal assignments. However, this semantics enables the representation of a δ-delay signal assignment with a unit-delay signal assignment adorned with a "`after 0 ns`" time clause. The hierarchical structure of designs is not preserved, and, although expressible, the semantics does not explicitly express a synchronous simulation cycle.

The denotational semantics expressed in [89] uses interval temporal logic as an underlying model. Leveraging this underlying model, the authors are interested in proving some properties over VHDL designs to help compilers to optimize the code, for instance, by using rewrite rules proved to be valid against the model. Some of the proofs laid out by the authors are embedded in PVS [88]. The expression of the dynamic model uses many concepts described in the LRM, like drivers, port association, driving and effective values for signals. The semantics deals with both unit-delay and δ-delay signal assignments. The semantics works on fully-elaborated designs, therefore, it does not deal with component instantiation statements. Moreover, interval temporal logic is useful to reason on the VHDL designs in the presence of delays, however, it looses its interest for designs presenting only 0-delay assignments.

In [17], the author states that "denotational semantics is more adequate for mathematical

reasoning". The author formalizes the VHDL semantics to prove the equivalence between VHDL programs (for instance, a specification and an implementation). What is of major interest regarding our needs is that the author has expressed a simulation cycle for synchronous designs. Therefore, a distinction is made between combinational and synchronous processes in the abstract syntax. Moreover, this work formalizes the elaboration part of a VHDL design former to the simulation; also, the elaboration keeps the hierarchical setting of the VHDL design, that is component instantiation statements are not replaced by processes. Due to the time abstraction, the semantics only deals with 0-delay signal assignments. It is explained by the fact that the reference time-unit is the clock period (i.e, the only known time-step), and the advancing of time, happening during the simulation cycle as described in the LRM, is captured within the setting of the simulation cycle.

Operational semantics

Multiple works formalize an operational semantics for VHDL. These works are interested in the formal description of the VHDL simulator. The aim is to devise a formal semantics that acts as a formal specification for a simulator.

In [23], a formal description of a *functional* semantics for VHDL is laid out based on stream-processing functions. The semantics is expressed with the functional programming language Gofer [67], thus enabling the computation of execution traces, that is, the computation of the streams representing the values taken by signals over time. As in the former work of the same author [24], only unit-delay signal assignments are dealt with, however, this time the author describes a deterministic operational semantics. Regarding our needs, this work is neither interested in preserving the hierarchical structure of VHDL designs, and no mention is made regarding how a design is elaborated, nor in expressing an explicit synchronous simulation cycle.

In [15], the authors formalize the simulation loop of the LRM using Evolving Algebra machines (EA-machines). All important constructs of the VHDL language are represented as records; processes are represented as concurrent agents running pseudo-codes, and the simulation control flow is passed to and fro between the kernel process (i.e, the simulation orchestrator) and the rest of the processes that execute the design behavior. This semantics implements closely the simulation loop as described in the LRM. Therefore, it is very rich and deals with most of the VHDL constructs, including the two time paradigms of the language (i.e. δ time and unit time). Moreover, the semantics works on fully-elaborated designs, therefore, component instantiation statements are omitted. However, a synchronous execution is fully expressible even if not explicitly embedded in the expression of the simulation loop.

In [109], the author presents a natural semantics for VHDL. The simulation loop is expressed by inference rules, and the execution of processes is based on the events over signals of their corresponding sensitivity lists. The execution of statements computes transaction traces, that is, the drivers of the signals. The semantics deals both with unit and delta delay signal assignments. Regarding our needs, this semantics does not entirely cover the subset of VHDL we are interested in. Component instantiation statements are not dealt with. A synchronous execution is expressible within the semantics, although it would be hidden in the inference rule formalizing the generic simulation loop. Also, the semantics does not provide its simulation loop with

a simulation horizon (a maximum number of simulation cycles). The simulation ends when signal values evolve no more.

In [54], the author presents an operational semantics for VHDL in the small-step style. The semantics follows closely the simulation cycle described in the LRM; however it is very concise and clear. The covered VHDL subset comprises both unit and delta-delay signal assignments. There is an interesting discussion about the non-determinism of VHDL, since it is a concurrent programming language: it entails that non-determinism is only existent at the processes level, that is, internal sequential statement of processes can be executed in a nondeterministic manner (referred to as A actions, that is, *internal* actions). But at every delta or time step (referred to as δ and T actions) of the execution, the design state can be computed in a deterministic manner, since all processes have reached a suspension point at the end of their inner body. The author is interested in comparing the behaviors of two VHDL designs by proving that some relation of equivalence holds between the two. He describes two strategies to compare VHDL programs. The first one is bisimulation; it is based on the comparison of the sequence of actions (either A, δ or T actions) performed by the two programs. The second one is observational equivalence; it is based on the observation of the value of the output signals of two VHDL programs (the observees), that receive values in their input signals from another VHDL program (the observer). The observer stimulates the entries of the observees and reaches a success state based on its observations of the value of the outputs. Regarding our needs, this semantics permits the description of our synchronous simulation cycle. However, like most of the semantics presented here, the component instantiation statement is not supported as it stands, but it is rather transformed into the equivalent processes statements. Small-step semantics is not needed in our case because we are only interested in the values of signals at the delta and time steps (for us, time steps correspond to clock events). We are not interested in capturing the design states in the middle of the execution of a process body. We are more interested in "weak bisimulation", therefore forsaking the internal actions performed by a VHDL design. In [107], the authors extend the work of [54], especially by handling shared variables, in the presence of which a VHDL program can have a concrete nonderterministic behavior. The authors are also interested in the equivalence between two VHDL programs, and they are interested in determining a unique meaning property for VHDL programs. The unique meaning property states that the execution of a VHDL design in the presence of shared variables is unique. This work is interesting as it points out the fact that the VHDL language is not only subject to "*benign* nondeterminism". By benign nonderterminism, the authors of [107] mean that the only moment where the state of a VHDL design can not be decided in a deterministic way is when the processes are in the middle of the execution of their statement body. However, the state of a VHDL design at that moment is of no interest; it corresponds to nothing regarding the concrete functioning of a hardware circuit. Also, two different processes can never be writing to the same signal at the same time. If such a design happens, this is a case of *multiply-driven* signal, which is utterly forbidden. So, there can be no nondeterminism, regarding the value of a signal, coming from the concurrent execution of two processes (at least when shared variables are not involved).

Translational semantics

Another kind of semantics, called "translational", formalizes the VHDL language semantics by translating a VHDL design into another formal model. Thus, the semantics of VHDL is modeled by the translation and the formal semantics of the target model. The target model has the ability to model concurrency, which is one of the specificity of VHDL. Moreover, target models are chosen regarding the tools they provide for analysis, and thus, a translational semantics for VHDL is often related to model checking considerations.

In [97], the author expresses the formal semantics of VHDL by translating a VHDL design into a corresponding *flowgraph*. All VHDL constructs, ranging from sequential statements to concurrent processes, are expressed with individual flowgraphs that are then composed together through their interfaces. The simulation cycle of VHDL is also encoded by means of connected flow graphs: one for the "execution part" of the semantics, that is, all processes run until suspension, and one for the update part (i.e, the kernel process). Flowgraphs come with a large amount of tools for analysis, and this translational semantics is involved in the setting of a framework to reason about VHDL programs using multiple techniques (automatic theorem proving, model checking...). All these techniques rely on the flowgraph formalism.

In [45], the author introduces a translational semantics for VHDL based on deterministic finite-state automatons. Again, the reason for using such automatons lies in the existence of many analysis tools. Moreover, forcing the generation of deterministic automatons improves the time execution of model-checking techniques. The translation is performed on an elaborated VHDL design; a data space stores the values of signals and variables, and automatons represent the control-flow of VHDL statements. Each VHDL statement is associated to a specific automaton; sequence of statements are achieved by automaton composition. The simulation kernel is also represented by a specific automaton. Processes are composed together with respect to synchronization states, i.e. states that permit to pass the control from one process to another, therefore achieving determinism in the control flow of the overall automaton.

In [87], the author presents a translation from VHDL to Coloured Petri Nets (CPNs) thus giving a formal semantics to the VHDL constructs. The author approach to the VHDL semantics is a strict translation of the "event-based" VHDL simulator by means of Petri nets. The author translates VHDL execution models (sea of processes) into CPNs, and also translates the kernel process into a CPN. The kernel process has previously been expressed as a VHDL process so that the translation into CPN is similar to the translation of other processes. Signals are not represented in the subnets, instead, three shared variables depict the signal states: one variable for the driving, one for the effective and one for the current value of a given signal (see [64, p.167] for the details on the values associated with signals during the simulation). Color domains of places in the subnets represent the different types of VHDL domains. Variables are represented by tokens. Values in drivers are represented by sequences of transactions (equivalent to waveforms); the author defines a set of functions that are convenient to handle sequences of transactions. Sequential statements are partitioned into two kinds: control flow (if, loop, case...) and notation (operations on signals and variables) nets. Processes subnets are made by the fusing of each sequential statements in the process body. There is a special *Resume* place that can be set by the kernel process to resume the activity of a process. Concurrency is not discussed here, as the Petri net models are inherently concurrent models. The kernel process is

a broad CPN having some of its places interfaced with the process subnets. The decoloration of the Petri net enables the analysis of the model and the detection of dead-locks.

In [42], the author gives a formal semantics to VHDL by transforming a VHDL design into an abstract machine, i.e. defined by a set of inputs, outputs, states and transition function over states and outputs. The author is interested in the verification of properties over VHDL designs (temporal properties) or to prove equivalence between designs (bisimulation). To operate this transformation, only a subset of VHDL is considered, otherwise a finite-state representation is not reachable. The covered VHDL subset consists of objects with finite types, and no quantitative timing constructs (no after clause in signal assignments). The transformation generates a decision diagram (i.e. a control flow graph) and a state space for each process defined in the design's behavior. The decision diagram encodes the transition function over states and outputs. Process statements are composed with a special composition operator to obtain a global abstract machine. Moreover, the article lays out a method to transform a block statement into an abstract machine. The initiative is to be noticed as there are only a few papers, dealing with the formalization of the VHDL semantics, that are interested in such hierarchical constructs as block or component instantiation statements. The article concludes with an expression of the space of complexity entailed by the transformation of a VHDL design into an abstract machine.

Although the translational semantics described above meet most of the qualifying criterions in relation to our needs, we are not especially interested in implementing one of these. The main reason being that it would require to implement the transformation from the abstract VHDL syntax to the target model, in addition to the implementation of the semantics of the target model.

Table 4.1 summarizes the analysis of the VHDL semantics encountered during our literature review. Table 4.1 compares the different VHDL semantics in relation to our qualifying criterions (see Section 4.2.1).

		Fuchs and Mendler [51]	Breuer et al. [24]	Pandey et al. [89]	Borrione and Salem [17]	Breuer et al. [23]	Börger et al. [15]	Van Tassel [109]	Goossens [54]	Reetz and Kropf [97]	Döhmen and Herrmann [45]	Olcoz [87]	Déharbe and Borrione [42]
Semantics Description	Kind	D	D, A	D	D	O	O	O	O	T	T	T	T
	Purpose	AR, ATP	AR	AR	AR	SS	SS	SS, ITP	SS, MC	ATP, MC, ITP	MC, ITP	MC	MC
Qualifying Criterions	Component Instantiation	T	T	T	N	T	T	T	T	T	T	T	N
	Synchronism	NE	NE	NE	Ex	E	E	E	E	E	E	E	NE
	Elaboration	×	×	×	✓	×	×	✓	×	×	×	×	✓
Extra. Informations.	Impl. Technology	Focus [41]	Prolog [33]	PVS [88]	?	Gofer [67]	?	HOL [63]	?	HOL [63]	?	?	?
	Particular Model or Data Types	Stream Processing	No	Interval Temporal Logic	No	Stream Processing	Evolving Algebra Machines	Natural Semantics (big-step)	Structural Semantics (small-step)	Flow Graphs	Finite-State Automatons	Colored Petri Nets	Abstract Machines and Decision Diagrams

TABLE 4.1: A comparative summary on VHDL formal semantics.

- Kind : D (Denotational) - A (Axiomatic) - O (Operational) - T (Translational).

- Purpose : AR (Abstract Reasoning) - ATP (Automatic Theorem Proving) - SS (Simulator Specification) - ITP (Interactive Theorem Proving) - MC (Model Checking).

- Component Instantiation : T (statement is *Transformed* into equivalent processes) - N (statement is *Natively* taken into account in the semantics).

- Synchronism : E (Expressible within the semantics) - NE (Not Expressible within the semantics) - Ex (Explicitly built in the semantics).

To summarize, we are interested in a semantics built for the purpose of interactive theorem proving (ideally, with an existing implementation in the Coq proof assistant). Most important, the formal semantics must be able to deal with the expression of synchronous designs, that is, designs synchronized with a clock signal. Therefore, a synchronous simulation cycle must be at least expressible within the semantics. Moreover, the semantics must handle component instantiation statements as they are, that is, without transforming them into equivalent processes. As a bonus, the semantics should formalize the elaboration part of VHDL semantics.

In Table 4.1, cells are colored in green when the cell's content foster the adoption of the semantics, in yellow when the content does not go towards the adoption of the semantics but is not disqualifying, and red when the content is a disqualifying criterion. Regarding the semantics adoption, cells are labelled in light grey when their content is neutral. Now comparing the entries of Table 4.1 with the expression of our needs, we can discard the semantics with a cell labelled in red, that is, most of the denotational semantics; moreover, all translational semantics are disqualified for the previously mentioned reasons. The candidate semantics are the operational semantics, plus the denotational semantics by Borrione and Salem [17], the only semantics that formalizes an explicitly synchronous simulation cycle. The semantics that is the most likely to be adopted is the Borrione and Salem's semantics. However, we prefer an operational setting for our semantics. To lower down the complexity of proofs, we really need a semantics that builds the synchronism into its simulation cycle, therefore putting aside all the intricacies of the full-blown VHDL simulation cycle. Moreover, the big-step style for an operational semantics is more relevant to us; as stated before, we are not interested in the intermediary states of computation that a small-step style semantics considers. Based on these observations, we have decided to formalize our own VHDL semantics inspired from the semantics of Borrione and Salem's [17] and Van Tassel's [109]. The following sections are dedicated to the presentation of the syntax and semantics of a subset of VHDL that we baptize $\mathcal{H}$-VHDL. $\mathcal{H}$-VHDL embeds the subset of VHDL that we are interested in when considering the VHDL designs generated by the HILECOP transformation.

4.3   Abstract syntax of $\mathcal{H}$-VHDL

In this section, we describe the abstract syntax of $\mathcal{H}$-VHDL, a subset of VHDL covering all the constructs present in the programs generated by the HILECOP transformation. Terminals of the language are written in `typewriter` font, or are enclosed in simple quotes for symbols with no typewriter representation. The a^* denotes a possibly empty repetition of the element a; the a^+ denotes a non-empty repetition of a.

4.3.1   Design declaration

Similarly to [109], we define the *design* construct in the $\mathcal{H}$-VHDL's abstract syntax which has no equivalent in the concrete syntax of VHDL.

In the above entry, id_e indicates the entity identifier and id_a the architecture identifier of the declared design. The gens entry corresponds to the generic clause, i.e. the declaration list for the generic constants of the design. A generic constant is declared via the gdecl entry; a

design   ::=   design id_e id_a gens ports sigs cs
gens   ::=   gdecl*
ports   ::=   pdecl*
sigs   ::=   sdecl*

gdecl   ::=   (id, τ, e)
pdecl   ::=   ((in|out), id, τ)
sdecl   ::=   (id, τ)

generic constant declaration is a triplet composed of an identifier, a type indication and an expression denoting the generic constant's default value. The ports entry holds the declaration of the input and output ports of the design. A port declaration (i.e. the pdecl entry) is a triplet composed of a port type, i.e. in or out, an identifier, and a type indication. The sigs entry is the list declaring the internal signals of the design. An internal signal declaration entry (i.e. sdecl) is a couple composed of an identifier and a type indication. The cs entry represents the concurrent statements composing the behavior of the design.

4.3.2 Concurrent statements

cs   ::=   psstmt | cistmt | cs || cs | null

In $\mathcal{H}$-VHDL, two kinds of concurrent statements are available to describe the behavior of a design: process statements, represented by the psstmt entry, and component instantiation statements, represented by the cistmt entry. Concurrent statements are composable through the || operator. We add the null statement to the $\mathcal{H}$-VHDL abstract syntax to help represent idle behaviors.

Process statement

psstmt   ::=   process (id_p, sl, vars, ss)
sl   ::=   id*
vars   ::=   vdecl*
vdecl   ::=   (id, τ)

A process statement declares a sensitivity list, i.e. the sl entry, which is a possibly empty set of signal identifiers. In order to be well-formed, the signals composing a sensitivity list must be either internal signals or input ports of the design. As a good practice, all signals which value is read in the sequential statement body of the process must appear in the sensitivity list. The process possibly declares a set of internal variables, i.e. the vars entry. A variable declaration entry is a couple composed of a variable identifier and a type indication. The ss entry represents the sequence of statements composing the body of the process, i.e. the part that will be executed during the simulation.

Component instantiation statement

The VHDL LRM defines two kinds of component instantiation statement (CIS): the instantiation of a component [64, p.139] and the instantiation of a design entity [64, p.141]. The component instantiation statement used in the $\mathcal{H}$-VHDL abstract syntax corresponds to the instantiation of a design entity.

cistmt	::=	comp (id_c, id_e, g, i, o)
g	::=	$assoc_g^*$
i	::=	$assoc_{ip}^*$
o	::=	$assoc_{op}^*$
$assoc_g$	::=	(id,e)
$assoc_{ip}$	::=	(name,e)
$assoc_{op}$	::=	(id,(name|open)) | (id(e),name)

In the cistmt entry, the identifier id_c represents the name of component instance. Identifier id_e points out the name of the design, i.e. the entity identifier, being instantiated here. The g entry describes the list of associations between generic constant identifiers and expressions. The i entry is the list of associations between input port identifiers (or indexed identifiers) and expressions. The o entry is the list of associations between output port identifiers (or indexed identifiers) and signal names, or the open keyword. Associating the open keyword with an output port identifier indicates that the port is not connected. The left element of an association is called the *formal* part, and the right element of an association is called the *actual* part.

4.3.3   Sequential statements

ss	::=	name $\Leftarrow$ e | name := e | if (e) ss [ss] | for (id,e,e) ss
		| falling ss | rising ss | rst ss ss' | ss; ss | null

The ss entry defines the sequential statements that compose the body of processes. The signal assignment statement is represented with the $\Leftarrow$ operator; the variable assignment statement with the := operator. Also, we devise three control flow statements that have no equivalent in the VHDL syntax: the falling block statement, the rising block statement and the rst block (or reset block) statement. The falling ss statement (resp. rising ss) declares a block of sequential statements to be executed only at the falling edge (resp. rising edge) of the clock signal (see Section 4.6.5). Also, the rst ss ss' statement declares two blocks, the first one must be executed during the initialization phase of the simulation; otherwise, the second one is executed (see Section 4.6.4). These introduced constructs are equivalent to specific if-else statements that are commonly used in the body of a synchronous process. The rst ss ss' statement is equivalent to:

```
if rst = '0' then ss else ss' end if;
```

In the above listing, '0' is the equivalent to the Boolean value `false` in concrete VHDL syntax. More details are given, in Section 4.6.4, regarding the semantics of the above statement in relation to the initialization phase that starts a design's simulation.

The `rising` ss and `falling` ss statements are equivalent to the following `if` statements:

```
if rising_edge(clk) then ss end if;
if falling_edge(clk) then ss end if;
```

In the above listing, the `rising_edge` (resp. `falling_edge`) primitive yields `true` if a rising edge event (resp. falling edge event) occurred in the `clk` signal passed as input. More details are given, in Section 4.6.5, regarding the semantics of the above statements in relation to the *clock* phases happening during a simulation cycle.

4.3.4 Expressions, names and types

$$
\begin{array}{lll}
\text{e} & ::= & \text{e and e} \mid \text{e or e} \mid \text{not e} \mid \text{e = e} \mid \text{e} \neq \text{e} \\
& & \mid \text{e < e} \mid \text{e <= e} \mid \text{e > e} \mid \text{e >= e} \mid \text{e + e} \mid \text{e - e} \\
& & \mid \text{name} \mid \text{natural} \mid \text{boolean} \mid (\text{e}^{+})
\end{array}
$$

$$
\begin{array}{lll}
\text{name} & ::= & \text{id} \mid \text{id(e)} \\
\text{boolean} & ::= & \text{true} \mid \text{false} \\
\tau & ::= & \text{boolean} \mid \text{natural (e, e)} \mid \text{array} (\tau, \text{e}, \text{e})
\end{array}
$$

The expression entry, i.e. e, declares a set of operators over Boolean expressions, and natural numbers expressions. The natural non-terminal represents the set of natural numbers ($\mathbb{N}$). The id non-terminal represents the set of identifiers, comparable to the set of non-empty strings, or any infinitely enumerable set. In the following sections, concrete identifiers will be written in `typewriter` font, e.g. the `place` and `transition` design identifiers.

The τ entry corresponds to the type indication associated with the declaration of a generic constant, a port or an internal signal. The considered types are the *Boolean* type, the constrained natural type, and the array type. The constrained natural type, i.e. `natural(e,e)`, defines a finite interval of natural numbers; the left-most expression of the range constraint denotes the lower bound of the interval, and the second one denotes the upper bound of the interval. The array type indication, i.e. `array(`τ`, e, e)`, denotes a non-empty set of elements of type τ. The elements are indexed with respect to the specified *index* constraint. The left-most expression of the index constraint denotes the starting index (possibly different from 0) and the right-most expression denotes the final index.

4.4 Preliminary definitions

4.4.1 Semantic domains

Let *id* denote the set of identifiers in the semantic domain. We write *prefix-id* to denote arbitrary subsets of the *id* set. The *type* and *value* semantic types are defined as follows:

TABLE 4.2: The *type* and *value* semantic types.

type	::=	*bool* | *nat*(*n*, *n*) | *array*(*type*, *n*, *n*)
value	::=	*b* | *n* | *arr*
b	::=	'⊤' | '⊥'
n	::=	0 | 1 | ... | NATMAX
arr	::=	(*value*$^+$)

In Table 4.2, the *type* type is in any way similar to the τ entry of the H-VHDL abstract syntax. However, all constraint bounds, that were expressions in the constrained natural and the array type indications, have been evaluated to natural numbers. NATMAX denotes the maximum value for a natural number. The NATMAX value depends on the implementation of the VHDL language; NATMAX must at least be equal to $2^{31} - 1$. Note that the *array* value contains at least one value as an array's index range contains at least one index.

4.4.2   Elaborated design and design state

Now, let us define the structure of *elaborated design*. An elaborated design is built during the elaboration of a H-VHDL design (see Section 4.5). Then, the elaborated design will act as a runtime environment in the expression of the simulation rules. Let *ElDesign* be the set of elaborated designs. An elaborated design is a composite environment built out of multiple sub-environments. Each sub-environment is a table, represented as a function, mapping identifiers of a certain category of constructs (e.g, input port identifiers) to their declaration information (e.g, type indication for input ports). We represent an elaborated design as a record where the fields are the sub-environments. An elaborated design is defined as follows:

Definition 31 (Elaborated Design). *An elaborated design* $\Delta \in ElDesign$ *is a record* <*Gens*, *Ins*, *Outs*, *Sigs*, *Ps*, *Comps*> *where:*

— *Gens* ∈ *generic-id* → (*type* × *value*) *is the function yielding the type and the value of generic constants.*

— *Ins* ∈ *input-id* → *type is the function yielding the type of input ports.*

— *Outs* ∈ *output-id* → *type is the function yielding the type of output ports.*

— *Sigs* ∈ *declared-signal-id* → *type is the function yielding the type of declared signals.*

— *Ps* ∈ *process-id* → (*variable-id* → (*type* × *value*)) *is the function associating process identifiers to their local environment.*

— *Comps* ∈ *component-id* → *ElDesign is the function mapping component instance identifiers to their own elaborated design version.*

We assume that there is no overlapping between the identifiers of the sub-environments (i.e, an identifier belongs to at most one sub-environment), and also between the identifiers of

the sub-environments and the identifiers of local environments. When there is no ambiguity, we write $\Delta(x)$ to denote the value returned for identifier x, where x is looked up in the appropriate field of Δ. We write $x \in \Delta$ to state that identifier x is defined in the domain of one of Δ's field. We note $\Delta(x) \leftarrow v$ the overriding of the value associated to identifier x with value v in the appropriate field of Δ, $\Delta \cup (x, v)$ to note the addition of the mapping from identifier x to value v in the appropriate field of Δ, that assuming $x \notin \Delta$. We write $x \in \mathcal{F}(\Delta)$, where $\mathcal{F}$ is a field of Δ, when more precision is needed regarding the lookup of identifier x in the record Δ.

Now let us define the run-time state of a design, i.e. the state that describes the value of signals and component instances in the course of a simulation. Let Σ be the set of design states. A design state of $\sigma \in \Sigma$ is defined as follows:

> **Definition 32** (Design state). *A design state $\sigma \in \Sigma$ is a record $<\mathcal{S}, \mathcal{C}, \mathcal{E}>$ where:*
>
> $- \mathcal{S} \in signal\text{-}id \rightarrow value$, *is the function yielding the current values of the design's signals (ports and declared signals).*
>
> $- \mathcal{C} \in component\text{-}id \rightarrow \Sigma$, *is the function yielding the current state of component instances.*
>
> $- \mathcal{E} \subseteq signal\text{-}id \sqcup component\text{-}id$, *is the set of signal and component instance identifiers that generated an event at the current design state.*

The *signal-id* subset is the disjoint union of *input-id*, *output-id* and *declared-signal-id*. When there is no ambiguity regarding which store a given identifier belongs, we use $\sigma(id)$ to denote the value associated to an identifier in the signal store $\mathcal{S}$ or in the component store $\mathcal{C}$ fields. We write $id \in \sigma$ to state that an identifier is defined in either the signal store $\mathcal{S}$ or the component store $\mathcal{C}$ fields. Also, when there is no ambiguity, we rely on indices or exponents to qualify the signal store, the component instance store and the set of events of a given design state. For instance, $\mathcal{C}_0$ denotes the component instance store of design state σ_0, and $\mathcal{E}'$ denotes the set of events of design state σ', etc.

> **Notation 6** (No events design state). *The function $NoEv \in \Sigma \rightarrow \Sigma$ returns a design state similar to the one passed in parameter but with an empty set of events. I.e, for all design state $\sigma \in \Sigma$ s.t. $\sigma = <\mathcal{S}, \mathcal{C}, \mathcal{E}>$, $NoEv(\sigma) = <\mathcal{S}, \mathcal{C}, \varnothing>$.*

4.5 Elaboration rules

The goal of the elaboration phase is to build an elaborated design Δ along with a *default* state σ_e out of a $\mathcal{H}$-VHDL design d and for a given design store $\mathcal{D}$. The elaboration relation performs type-checking operations over the declarative and behavioral parts of the design. Even though the elaboration of a design is described in the LRM, the formalization of this phase has been performed in few works only [17, 42, 109], and never in a setting that covers both syntactical well-formedness and type-checking of the designs. We are interested in the formalization of the elaboration phase because we are interested in the *well-formedness* of the programs generated

by the HILECOP transformation. Here, the term well-formedness refers to a syntactically valid design, w.r.t. the syntactic rules of the VHDL language, and to a well-typed design, w.r.t. the typing rules defined in the LRM. Formalizing the elaboration phase is also a way to define how the runtime environment and the runtime state of the simulation are built. For now, we haven't tackled down the proof that the $\mathcal{H}$-VHDL designs generated by HILECOP are elaborable, i.e. syntactically well-formed and well-typed. As explained in Chapter 6, this task is foreseen in our work perspectives. In our own formalization of the elaboration phase, and contrary to what is prescribed by the LRM [64, p. 166], we are not dealing with the transformation of the component instantiation statements into block statements. We prefer to preserve the hierarchical structure of the design (i.e. its composite structure) during its elaboration. We argue that dealing with component instantiation statements instead of block statements does not add complexity to the semantics of the $\mathcal{H}$-VHDL simulation rules.

In the following sections, the green frames give additional explanations about the premises of the rule instances; the red frames bring additional explanations about the side conditions of the rules.

4.5.1   Design elaboration

One way to define a design's behavior is through the instantiation of subcomponents which are instances of other designs. Each component instance declares the entity identifier that points out to the specific design being instantiated. Therefore, for each instantiation, the associated design must be known through the definition of a global design declaration environment called a *design store*. A design store is defined as follows:

> **Definition 33** (Design store). *A design store $\mathcal{D} \in$ entity-id $\to$ design is a function mapping design identifiers (i.e. the entity identifier of designs) to their corresponding representation in abstract syntax. As a prerequisite to the elaboration of HILECOP-generated designs (i.e, resulting from the transformation of a SITPN into a $\mathcal{H}$-VHDL design), a particular design store $\mathcal{D}_{\mathcal{H}}$ is defined. Design store $\mathcal{D}_{\mathcal{H}}$ binds the* `transition` *and* `place` *identifiers to the definition of the* `place` *and* `transition` *designs in $\mathcal{H}$-VHDL abstract syntax:*
> $\mathcal{D}_{\mathcal{H}} :=$ $\{$(transition, design transition transition_architecture $gens_t$ $ports_t$ $sigs_t$ cs_t),
>   (place, design place place_architecture $gens_p$ $ports_p$ $sigs_p$ cs_p)$\}$
>
> *The full definition of the* `place` *and* `transition` *designs in abstract syntax are given in Appendices A and B.*

At the beginning of the elaboration phase, a function $\mathcal{M}_g \in$ *generic-id $\nrightarrow$ value* mapping the top-level design's generic constants to values is passed as an element of the environment. The $\mathcal{M}_g$ function is referred to as the *dimensioning* function.

Rule DESIGNELAB defines the design elaboration relation. It relates a $\mathcal{H}$-VHDL design to its resulting elaborated version and default design state that were built in the context of the design store $\mathcal{D}$ and the dimensioning function $\mathcal{M}_g$.

DESIGNELAB

$$\frac{\Delta_{\varnothing}, \mathcal{M}_g \vdash \text{gens} \xrightarrow{egens} \Delta \quad \Delta, \sigma_{\varnothing} \vdash \text{ports} \xrightarrow{eports} \Delta', \sigma \quad \Delta', \sigma \vdash \text{sigs} \xrightarrow{esigs} \Delta'', \sigma' \quad \mathcal{D}, \Delta'', \sigma' \vdash \text{cs} \xrightarrow{ebeh} \Delta''', \sigma''}{\mathcal{D}, \mathcal{M}_g \vdash \text{design id}_e \text{ id}_a \text{ gens ports sigs cs} \xrightarrow{elab} \Delta''', \sigma''}$$

$\Delta_{\varnothing}$ denotes an empty elaborated design, that is an elaborated design initialized with empty fields (empty tables). In the same manner, $\sigma_{\varnothing}$ denotes an empty design state. The effect of the *egens*, *eports*, *esigs* and *ebeh* relations that respectively deal with the elaboration of the generic constants, the ports, the architecture declarative part and the behavioral part of the design, are made explicit in the following sections.

4.5.2 Generic clause elaboration

The *egens* relation elaborates the list of generic constant declarations, i.e. the generic clause of a design declaration. The *egens* relation is defined through the GENELABDIMEN, GENELAB-DEFAULT and GENELABCOMP rules. The elaboration of a generic constant declaration consists in:

1. Transforming the type indication associated with the constant into a semantic type.

2. Checking that the default value, and/or the value associated with the constant in the dimensioning function, is well-typed.

3. Adding the couple constant identifier and *(type,value)* to the *Gens* sub-environment of Δ.

> **Premises**
>
> - *etype$_g$* transforms a type indication, specifically attached to a generic constant declaration, into a *type* instance and checks its well-formedness (see Section 4.5.5).
>
> - The *e* relation links an expression *e* to its value *v* in a given context (see Section 4.6.9). The context of evaluation for an expression is composed of a given elaborated design, a given design state, and given local environment. We omit the thesis symbol and symbols at the left of the thesis when they refer to empty structures. For instance, $e \xrightarrow{e} v$ is a notation for $\Delta_{\varnothing}, \sigma_{\varnothing}, \Lambda_{\varnothing} \vdash e \xrightarrow{e} v$.
>
> - *SE$_l$* states that an expression is *locally* static (see Section 4.5.9).
>
> - $v \in_c T$ and $\mathcal{M}(\text{id}_g) \in_c T$ checks that the default value and the value yielded by the dimensioning function belongs to the type of the declared generic constant (see Section 4.5.8).

Side conditions

The expression $id_g \notin \Delta$ checks that the generic constant identifier id_g is not already defined in the domain of one sub-environment of the elaborated design Δ.

GENELABDIMEN

$$\frac{\vdash \tau \xrightarrow{etype_g} T \quad \text{e} \xrightarrow{e} v \quad SE_l(\text{e}) \quad \begin{array}{c} v \in_c T \\ \mathcal{M}(id_g) \in_c T \end{array} \quad \begin{array}{c} id_g \notin \Delta \\ \\ id_g \in \mathcal{M} \end{array}}{\Delta, \mathcal{M} \vdash (id_g, \tau, \text{e}) \xrightarrow{egens} \Delta \cup (id_g, (T, \mathcal{M}(id_g)))}$$

The GENELABDEFAULT rule states that the value of a generic constant is defined by its type's default implicit value when no value is specified by the dimensioning function $\mathcal{M}$.

GENELABDEFAULT

$$\frac{\vdash \tau \xrightarrow{etype_g} T \quad \vdash \text{e} \xrightarrow{e} v \quad SE_l(\text{e}) \quad v \in_c T \quad \begin{array}{c} id_g \notin \Delta \\ \\ id_g \notin \mathcal{M} \end{array}}{\Delta, \mathcal{M} \vdash (id_g, \tau, \text{e}) \xrightarrow{egens} \Delta \cup (id_g, (T, v))}$$

GENELABCOMP

$$\frac{\Delta, \mathcal{M} \vdash \text{gdecl} \xrightarrow{egens} \Delta' \quad \Delta', \mathcal{M} \vdash \text{gens} \xrightarrow{egens} \Delta''}{\Delta, \mathcal{M} \vdash \text{gdecl, gens} \xrightarrow{egens} \Delta''}$$

4.5.3   Port clause elaboration

The *eports* relation elaborates each port declaration defined in a design's port clause. For each port declaration, the *eports* relation transforms the port's type indication into a semantic type and retrieves the implicit default value of this type. Then, the *eports* relation adds the binding between the input (resp. output) port identifier and its type to the *Ins* (resp. *Outs*) sub-environment of the elaborated design structure Δ. It also adds the binding between the input (resp. output) port identifier and its implicit default value to the default design state σ.

Premises

– The *etype* relation associates a type indication to its corresponding semantic type and checks its well-formedness (see Section 4.5.5).

– The *defaultv* relation associates a given semantic type to its implicit *default* value.

Side conditions

The expression id $\notin \sigma$ checks that the identifier *id* is not already defined in the domain of

> the signal store or the component store of the design state σ. It is a shorthand notation to
> $id \notin \text{dom}(\mathcal{S}) \cup \text{dom}(\mathcal{C})$ where $\sigma = <\mathcal{S}, \mathcal{C}, \mathcal{E}>$.

INPORTELAB

$$\frac{\Delta \vdash \tau \xrightarrow{etype} T \quad \Delta \vdash T \xrightarrow{defaultv} v}{\Delta, \sigma \vdash (\text{in}, id, \tau) \xrightarrow{eports} \Delta \cup (id, T), \sigma \cup (id, v)} \quad \begin{array}{l} id \notin \Delta \\ id \notin \sigma \end{array}$$

OUTPORTELAB

$$\frac{\Delta \vdash \tau \xrightarrow{etype} T \quad \Delta \vdash T \xrightarrow{defaultv} v}{\Delta, \sigma \vdash (\text{out}, id, \tau) \xrightarrow{eports} \Delta \cup (id, T), \sigma \cup (id, v)} \quad \begin{array}{l} id \notin \Delta \\ id \notin \sigma \end{array}$$

PORTELABCOMP

$$\frac{\Delta, \sigma \vdash \text{pdecl} \xrightarrow{eports} \Delta', \sigma' \quad \Delta', \sigma' \vdash \text{ports} \xrightarrow{eports} \Delta'', \sigma''}{\Delta, \sigma \vdash \text{pdecl}, \text{ports} \xrightarrow{eports} \Delta'', \sigma''}$$

4.5.4 Architecture declarative part elaboration

The *esigs* relation elaborates each internal signal declaration defined in the declarative part of a design's architecture. For each signal declaration, the *esigs* relation transforms the signal's type indication into a semantic type and retrieves the implicit default value of this type. Then, the *esigs* relation adds the binding between the signal identifier and its type to the *Sigs* sub-environment of the elaborated design structure Δ. It also adds the binding between the signal identifier and its implicit default value to the default design state σ.

SIGELAB

$$\frac{\Delta \vdash \tau \xrightarrow{etype} T \quad \Delta \vdash T \xrightarrow{defaultv} v}{\Delta, \sigma \vdash (id, \tau) \xrightarrow{esigs} \Delta \cup (id, T), \sigma \cup (id, v)} \quad \begin{array}{l} id \notin \Delta \\ id \notin \sigma \end{array}$$

SIGELABCOMP

$$\frac{\Delta, \sigma \vdash \text{sdecl} \xrightarrow{esigs} \Delta', \sigma' \quad \Delta', \sigma' \vdash \text{sigs} \xrightarrow{esigs} \Delta'', \sigma''}{\Delta, \sigma \vdash \text{sdecl}, \text{sigs} \xrightarrow{esigs} \Delta'', \sigma''}$$

4.5.5 Type indication elaboration

The *etype* relation checks the well-formedness of a type indication τ, and transforms it into a semantic *type* (as defined in Table 4.2). A type indication τ is well-formed in the context Δ if τ denotes the `boolean` keyword or the `natural` or `array` keywords with a *well-formed* constraint, and a well-formed element type in the `array` case.

$$
\begin{array}{c}
\text{ETYPEBOOL} \\
\hline
\Delta \vdash \texttt{boolean} \xrightarrow{etype} bool
\end{array}
\qquad
\begin{array}{c}
\text{ETYPENAT} \\
\Delta \vdash (\mathrm{e}, \mathrm{e}') \xrightarrow{econstr} (v, v') \\
\hline
\Delta \vdash \texttt{natural}(\mathrm{e}, \mathrm{e}') \xrightarrow{etype} nat(v, v')
\end{array}
$$

$$
\begin{array}{c}
\text{ETYPEARRAY} \\
\Delta \vdash \tau \xrightarrow{etype} T \qquad \Delta \vdash (\mathrm{e}, \mathrm{e}') \xrightarrow{econstr} (v, v') \\
\hline
\Delta \vdash \texttt{array}(\tau, \mathrm{e}, \mathrm{e}') \xrightarrow{etype} array(T, v, v')
\end{array}
$$

The *econstr* relation checks that a constraint is well-formed and evaluates the constraint bounds. A constraint is well-formed in the context Δ if:

- Its bounds are globally static expressions [64, p.36] conforming to the $nat(0, \texttt{NATMAX})$ type after evaluation.

- Its lower bound value is inferior or equal to its upper bound value.

> **Remark 1** (Type of constraints). *As the VHDL language reference stays unclear about the type of range and index constraints [64, p.33], we add the restriction that range and index constraints must have bounds of the $nat(0, \texttt{NATMAX})$ type, i.e. the interval of natural numbers representable with the VHDL language.*

> **Premises**
>
> – The $\in_c$ relation states that a given value conforms to a given type (see Section 4.5.8).
>
> – The SE_g relation states that an expression is *globally* static (see Section 4.5.9).

$$
\begin{array}{c}
\text{ECONSTR} \\
\Delta \vdash SE_g(\mathrm{e}) \qquad \Delta \vdash \mathrm{e} \xrightarrow{e} v \qquad v \in_c nat(0, \texttt{NATMAX}) \\
\Delta \vdash SE_g(\mathrm{e}') \qquad \Delta \vdash \mathrm{e}' \xrightarrow{e} v' \qquad v' \in_c nat(0, \texttt{NATMAX}) \\
\hline
\Delta \vdash (\mathrm{e}, \mathrm{e}') \xrightarrow{econstr} (v, v')
\end{array} \quad v \le v'
$$

When considering a type indication in a generic constant declaration, the definition of well-formedness differs slightly from the general definition. A type indication τ associated to a generic constant declaration is well-formed if τ denotes the `boolean` keyword, or the `natural` keyword with a *well-formed* constraint. A generic constant can not be associated with a composite type indication (i.e. an `array` type). The $etype_g$ relation is specially defined to check the well-formedness of a type indication associated with a generic constant declaration.

$$
\text{ETYPEGBOOL} \over \vdash \texttt{boolean} \xrightarrow{etype} bool
$$

$$
\text{ETYPEGNAT} \\
{\Delta \vdash (\text{e},\text{e}') \xrightarrow{econstr_g} (v,v') \over \vdash \texttt{natural}(\text{e},\text{e}') \xrightarrow{etype} nat(v,v')}
$$

The *econstr$_g$* relation checks that a *generic* constraint (i.e, a constraint appearing in a type indication associated with a generic constant declaration) is well-formed and evaluates the constraint bounds. A *generic* constraint is well-formed if:

- Its bounds are locally static expressions [64, p.36] conforming to the $nat(0, \texttt{NATMAX})$ type after evaluation.

- Its lower bound value is inferior or equal to its upper bound value.

$$
\text{ECONSTRG} \\
{SE_l(\text{e}) \quad \vdash \text{e} \xrightarrow{e} v \quad\quad v \in_c nat(0, \texttt{NATMAX}) \atop SE_l(\text{e}') \quad \vdash \text{e}' \xrightarrow{e} v' \quad v' \in_c nat(0, \texttt{NATMAX}) \over \vdash (\text{e},\text{e}') \xrightarrow{econstr_g} (v,v')} \; v \le v'
$$

4.5.6 Behavior elaboration

The *ebeh* relation elaborates each concurrent statement composing the behavioral part of a design.

Elaboration of concurrent statements

The elaboration of the composition of concurrent statements is performed sequentially.

$$
\text{CSPARELAB} \\
{\mathcal{D},\Delta,\sigma \vdash \text{cs} \xrightarrow{ebeh} \Delta',\sigma' \quad \mathcal{D},\Delta',\sigma' \vdash \text{cs}' \xrightarrow{ebeh} \Delta'',\sigma'' \over \mathcal{D},\Delta,\sigma \vdash \text{cs} \,||\, \text{cs}' \xrightarrow{ebeh} \Delta'',\sigma''}
$$

$$
\text{CSNULLELAB} \over \mathcal{D},\Delta,\sigma \vdash \texttt{null} \xrightarrow{ebeh} \Delta,\sigma
$$

Process statement elaboration

To elaborate a process statement, the *ebeh* relation associates the process identifier with a local environment in the *Ps* sub-environment of Δ. The *ebeh* builds the local environment based on the process's local variable declaration list (see the *evars* relation). The *ebeh* relation also checks that the sequential statements composing the body of the process are well-typed (see the *valid$_{ss}$* relation in Section 4.5.11).

> **Premises**
>
> The $\mathtt{valid}_{ss}$ relation states that a sequential statement is well-typed in the context Δ, σ, Λ, where Λ is the local variable environment deduced from the elaboration of the process declarative part.

> **Side conditions**
>
> $\mathrm{sl} \subseteq Ins(\Delta) \cup Sigs(\Delta)$ indicates that the sensitivity list sl must only contain *readable* signal identifiers, that is, input ports and internal signals.

PSELAB

$$\frac{\Delta, \Lambda_{\varnothing} \vdash \mathtt{vars} \xrightarrow{evars} \Lambda \qquad \Delta, \sigma, \Lambda \vdash \mathtt{valid}_{ss}(\mathtt{ss})}{\mathcal{D}, \Delta, \sigma \vdash \mathtt{process}\ (\mathit{id}_p,\ \mathtt{sl},\ \mathtt{vars},\ \mathtt{ss}) \xrightarrow{ebeh} \Delta \cup (\mathit{id}_p, \Lambda), \sigma} \quad \begin{array}{l} \mathit{id}_p \notin \Delta \\[4pt] \mathrm{sl} \subseteq Ins(\Delta) \cup Sigs(\Delta) \end{array}$$

Process declarative part elaboration

The *evars* relation builds a local environment out of a process declarative part. For each local variable declaration, the *evars* transforms the type indication associated with the variable identifier into a semantic type and retrieves the implicit default value of this type. Then, the *evars* relation adds the binding between the variable identifier, and the couple *(type,value)* to the local environment Λ.

VARELAB

$$\frac{\Delta \vdash \tau \xrightarrow{etype} T \qquad \vdash T \xrightarrow{defaultv} v}{\Delta, \Lambda \vdash (\mathrm{id}, \tau) \xrightarrow{evars} \Lambda \cup (\mathit{id}, (T, v))} \quad \begin{array}{l} \mathrm{id} \notin \Lambda \\[4pt] \mathrm{id} \notin \Delta \end{array}$$

VARELABCOMP

$$\frac{\Delta, \Lambda \vdash \mathtt{vdecl} \xrightarrow{evars} \Lambda' \qquad \Delta, \Lambda' \vdash \mathtt{vars} \xrightarrow{evars} \Lambda''}{\Delta, \Lambda \vdash \mathtt{vdecl}, \mathtt{vars} \xrightarrow{evars} \Lambda''}$$

Component instantiation statement elaboration

To elaborate a component instantiation statement, the *ebeh* relation first builds a dimensioning function $\mathcal{M}$ out of the component instance's generic map. Then, the design associated with the entity identifier declared by the component instance (i.e. id_e) is looked up and retrieved from the design store $\mathcal{D}$. Then, the *ebeh* relation appeals to the *elab* relation to build an elaborated version Δ_c and a default design state σ_c for the retrieved design given the specific dimensioning function $\mathcal{M}$. Finally, the component instance identifier id_c is bound to its elaborated version Δ_c in the *Comps* sub-environment of Δ, and is bound to its own default design state σ_c in the component store $\mathcal{C}$ of σ. Consequently, the definition of the *elab* and *ebeh* relations is mutually recursive.

> **Premises**
>
> - The *emapg* relation builds a function $\mathcal{M} \in generic\text{-}id \nrightarrow value$ out of a generic map (see the definition below).
>
> - $\mathtt{valid}_{ipm}$ (resp. $\mathtt{valid}_{opm}$) states that an input port map (resp. output port map) is valid, i.e. well-formed and well-typed (see Section 4.5.10).

> **Side conditions**
>
> $\mathcal{M} \subseteq Gens(\Delta_c)$ checks that the generic map g contains references to known generic constant identifiers only.

COMPELAB

$$\frac{\begin{array}{ccc} \mathcal{M}_\varnothing \vdash \mathtt{g} \xrightarrow{emapg} \mathcal{M} & \Delta, \Delta_c, \sigma \vdash \mathtt{valid}_{ipm}(\mathtt{i}) & \\ \mathcal{D}, \mathcal{M} \vdash \mathcal{D}(\mathrm{id}_e) \xrightarrow{elab} \Delta_c, \sigma_c & \Delta, \Delta_c \vdash \mathtt{valid}_{opm}(\mathtt{o}) & \begin{array}{l} \mathrm{id}_c \notin \Delta, \mathrm{id}_c \notin \sigma \\ \mathrm{id}_e \in \mathcal{D} \end{array} \end{array}}{\mathcal{D}, \Delta, \sigma \vdash \mathtt{comp}\,(\mathrm{id}_c, \mathrm{id}_e, \mathtt{g}, \mathtt{i}, \mathtt{o}) \xrightarrow{ebeh} \Delta \cup (id_c, \Delta_c), \sigma \cup (id_c, \sigma_c) \quad \mathcal{M} \subseteq Gens(\Delta_c)}$$

A port map is a mapping between expressions and signals coming from an embedding design (Δ) and the ports of an internal component instance (Δ_c). The formal part of a port map entry (i.e, the left part) belongs to the internal component, whereas the actual part (i.e, the right part) refers to the embedding design. Therefore, we need both Δ and Δ_c to verify if a port map is well-typed leveraging the $\mathtt{valid}_{ipm}$, or the $\mathtt{valid}_{opm}$, relation.

> **Remark 2** (Valid generic map). *In Rule* COMPELAB, *note that we are not checking the validity of the generic map g. In case of an ill-formed generic map, an inconsistent mapping $\mathcal{M}$ is generated by the* emapg *relation. In the presence of an ill-formed dimensioning function, the* elab *relation is never derivable. Therefore, the* elab *relation does an implicit validity check on the generic map.*

The *emap$_g$* relation builds a dimensioning function out of generic map. For each association of the generic map, the *emap$_g$* relation evaluates the actual part of the association, and adds a binding between the generic constant identifier and its value to the dimensioning function $\mathcal{M}$.

ASSOCGELAB

$$\frac{SE_l(\mathtt{e}) \qquad \mathtt{e} \xrightarrow{e} v}{\mathcal{M} \vdash (\mathrm{id}_g, \mathtt{e}) \xrightarrow{emapg} \mathcal{M} \cup (\mathrm{id}_g, v)}\ \mathrm{id}_g \notin \mathcal{M}$$

GMELAB

$$\frac{\mathcal{M} \vdash \mathrm{assoc}_g \xrightarrow{emapg} \mathcal{M}' \qquad \mathcal{M}' \vdash \mathrm{gmap} \xrightarrow{emapg} \mathcal{M}''}{\mathcal{M} \vdash \mathrm{assoc}_g, \mathrm{gmap} \xrightarrow{emapg} \mathcal{M}''}$$

An assoc$_g$ entry doesn't allow indexed identifiers in its formal part, due to the restriction of generic constants to scalar types. Note that this restriction is not imposed by the LRM. We choose to adopt this simplification of the VHDL syntax since the case of generic constants with composite types is never encountered in the VHDL programs generated by HILECOP.

4.5.7   Implicit default value

According to the VHDL LRM, at the declaration of a port, a signal or a variable, these items must receive an implicit default value depending on their types [64, p.61, 64, 173]. The *defaultv* relation determines the default value for a given type.

$$\frac{\text{DEFAULTVBOOL}}{bool \xrightarrow{defaultv} \bot} \qquad \frac{\text{DEFAULTVCNAT}}{nat(n,m) \xrightarrow{defaultv} n} \; n \leq m$$

$$\frac{\text{DEFAULTVCARR}}{array(T,n,m) \xrightarrow{defaultv} \texttt{create\_array}(size,T,v)} \; \begin{array}{l} n \leq m \\ size = (m-n)+1 \end{array}$$

The $\texttt{create\_array}(size, T, v)$ expression yields an array of size $size$, containing elements of type T, where each element is initialized with the value v.

4.5.8   Typing relation

The typing relation $\in_c$ checks that a given value conforms to a given type.

$$\frac{\text{ISBOOL}}{b \in_c bool} \; b \in \mathbb{B} \qquad \frac{\text{ISCNAT}}{n \in_c nat(l,u)} \; n \in [l,u] \qquad \frac{\text{ARRAY}}{v_i \in_c T}{\Delta \vdash (v_1,\dots,v_n) \in_c array(T,l,u)} \; \begin{array}{l} i = 1,\dots,n \\ n = (u-l)+1 \end{array}$$

4.5.9   Static expressions

Static expressions are either locally static or globally static; the LRM defines locally static and globally static expressions as follows.

Locally static expressions

An expression is *locally* static if:

- It is composed of operators and operands of a *scalar* type (i.e, `natural` or `boolean`).

- It is a *literal* of a scalar type.

The SE_l relation, defined by the following rules, states that an expression is locally static.

$$\frac{\text{LSENAT}}{SE_l(n)} \; n \in \mathbb{N} \quad \frac{\text{LSEBOOL}}{SE_l(b)} \; b \in \mathbb{B} \quad \frac{\text{LSENOT}}{SE_l(e)}{SE_l(\texttt{not e})} \quad \frac{\text{LSEBINOP}}{SE_l(e) \quad SE_l(e')}{SE_l(\texttt{e op e}')} \; op \in \{ +, -, =, \neq, <, \leq, >, \geq, \texttt{and}, \texttt{or} \}$$

Globally static expressions

An expression is *globally* static in the context Δ if:

- It is a generic constant.

- It is an array aggregate composed of globally static expressions.

- It is a locally static expression.

The SE_g relation, defined by the following rules, checks that an expression is globally static in a given context Δ.

$$
\frac{\text{GSELOCAL}}{SE_l(e)} \qquad \frac{\text{GSEGEN}}{\Delta \vdash SE_g(\text{id}_g)} \ \ \text{id}_g \in Gens(\Delta) \qquad \frac{\text{GSEAGGREGATE}}{\Delta \vdash SE_g((e_1, \ldots, e_n))} \ i = 1, \ldots, n
$$

4.5.10 Valid port map

Valid input port map

The `valid`$_{ipm}$ predicate states that an input port map is valid in the context Δ, Δ_c, where Δ is the embedding design structure and Δ_c denotes the component instance, owner of the input port map, if:

- All ports defined in Δ_c are exactly mapped once in the input port map.

- For each input port map entry, the formal and actual part are of the same type.

> **Premises**
>
> - $list_{ipm}$ builds a set $\mathcal{L} \subset id \sqcup (id \times \mathbb{N})$ out of the input port map.
>
> - `check`$_{pm}$ checks the validity of a port map based on the corresponding port list (here, the input ports of Δ_c) and the set built by the $list_{ipm}$ relation.

$$
\frac{\text{VALIDIPM}}{\Delta, \Delta_c, \sigma, \mathcal{L}_\emptyset \vdash i \xrightarrow{\ list_{ipm}\ } \mathcal{L} \quad \text{check}_{pm}(Ins(\Delta_c), \mathcal{L})}{\Delta, \Delta_c, \sigma \vdash \text{valid}_{ipm}(i)}
$$

The $list_{ipm}$ relation builds a set composed of identifiers and/or couples *(identifier, natural number)* collected from the identifiers and indexed identifiers found in the formal parts of an input port map. It also checks, for each association of the input port map, that the expression of the actual part is of the same type than the identifier or indexed identifier of the formal part.

Side conditions

- $\mathrm{id}_f \in Ins(\Delta_c)$ checks that the identifier id_f is an input port identifier of Δ_c.

- $\mathrm{id}_f \notin \mathcal{L}$ checks that the port identifier id_f is not already mapped, i.e. it is not already referenced in the $\mathcal{L}$ set.

- $\nexists v_i$ s.t. $(\mathrm{id}_f, v_i) \in \mathcal{L}$ checks that a subelement of id_f is not already map, that is, if id_f denotes a signal identifier of the array type.

LISTIPMSIMPLE

$$\frac{\Delta, \sigma \vdash \mathrm{e} \xrightarrow{e} v \quad v \in_c T}{\Delta, \Delta_c, \sigma, \mathcal{L} \vdash (\mathrm{id}_f, \mathrm{e}) \xrightarrow{list_{ipm}} \mathcal{L} \cup \{id_f\}} \quad \begin{array}{l} \mathrm{id}_f \notin \mathcal{L}, \mathrm{id}_f \in Ins(\Delta_c) \\ \nexists v_i \text{ s.t. } (\mathrm{id}_f, v_i) \in \mathcal{L} \\ \Delta_c(\mathrm{id}_f) = T \end{array}$$

Premises

$v_i \in_c nat(n, m)$ checks that the index value stays in the array bounds.

Side conditions

$\mathrm{id}_f \notin \mathcal{L}$ and $(\mathrm{id}_f, v_i) \notin \mathcal{L}$ checks that neither the port identifier id_f nor the couple port identifier id_f and index v_i are already mapped.

LISTIPMPARTIAL

$$\frac{\begin{array}{cc} & \mathrm{e}_i \xrightarrow{e} v_i \quad v_i \in_c nat(n, m) \\ SE_l(\mathrm{e}_i) \quad \Delta, \sigma \vdash \mathrm{e} \xrightarrow{e} v \quad v \in_c T \end{array}}{\Delta, \Delta_c, \sigma, \mathcal{L} \vdash (\mathrm{id}_f(\mathrm{e}_i), \mathrm{e}) \xrightarrow{list_{ipm}} \mathcal{L} \cup \{ (id_f, v_i) \}} \quad \begin{array}{l} \mathrm{id}_f \notin \mathcal{L}, (\mathrm{id}_f, v_i) \notin \mathcal{L} \\ \mathrm{id}_f \in Ins(\Delta_c) \\ \Delta_c(\mathrm{id}_f) = array(T, n, m) \end{array}$$

LISTIPMCONS

$$\frac{\Delta, \Delta_c, \sigma, \mathcal{L} \vdash assoc_{ip} \xrightarrow{list_{ipm}} \mathcal{L}' \quad \Delta, \Delta_c, \sigma, \mathcal{L}' \vdash \mathrm{i} \xrightarrow{list_{ipm}} \mathcal{L}''}{\Delta, \Delta_c, \sigma, \mathcal{L} \vdash assoc_{ip}, \mathrm{i} \xrightarrow{list_{ipm}} \mathcal{L}''}$$

The $\mathrm{check}_{pm}(Ports, \mathcal{L})$ predicate states that all port identifiers referenced in the domain of $Ports \in id \nrightarrow type$ appear in $\mathcal{L}$ as a simple identifier, or if the port identifier is of the *array* type, then all couples (id, i) must belong to $\mathcal{L}$, where i denotes all indexes of the index range and id denotes the port identifier.

$$\mathrm{check}_{pm}(Ports, \mathcal{L}) \equiv \forall \mathrm{id}_f \in \mathrm{dom}(Ports), \, \mathrm{id}_f \in \mathcal{L} \vee (Ports(\mathrm{id}_f) = \mathrm{array}(T, n, m) \wedge$$
$$\forall i \in [n, m], \, (\mathrm{id}_f, i) \in \mathcal{L})$$

Valid output port map

The $\texttt{valid}_{opm}$ predicate states that an *output* port map is valid in the context Δ, Δ_c, where Δ is the embedding design structure and Δ_c denotes the component instance owner of the port map, if:

- An output port identifier appears at most once in the output port map.

- Two different output port identifiers cannot be connected to the same signal.

- For each output port map entry, the formal and the actual part are of the exact same type.

We allow partially connected output port map; i.e, an output port map where all output ports might not be present in the mapping. Such output ports are open by default.

> **Premises**
>
> $list_{opm}$ builds two sets $\mathcal{L}, \mathcal{L}_{ids} \subseteq id \sqcup (id \times \mathbb{N})$ out of the output port map opmap. $\mathcal{L}_{ids}$ is built incrementally to check that there are no multiply-driven signals resulting of the port map connection.

$$\text{VALIDOPM}$$
$$\frac{\Delta, \Delta_c, \mathcal{L}_\varnothing, \mathcal{L}_{ids\varnothing} \vdash o \xrightarrow{list_{opm}} \mathcal{L}, \mathcal{L}_{ids}}{\Delta, \Delta_c \vdash \texttt{valid}_{opm}(o)}$$

> **Side conditions**
>
> - $\text{id}_f \notin \mathcal{L}$ checks that the port identifier id_f is not already mapped (i.e, is not already used in the formal part of a port map entry).
>
> - $\text{id}_a \notin \mathcal{L}_{ids}$ checks that the signal identifier id_a is not already mapped (i.e, is not already used in the actual part of a port map entry).
>
> - $\text{id}_f \in Outs(\Delta_c)$ checks that id_f is an output port identifier of Δ_c.
>
> - $\text{id}_a \in Sigs(\Delta) \cup Outs(\Delta)$ checks that id_a is either an output port or an internal signal identifier of Δ.
>
> - $\Delta_c(\text{id}_f) = \Delta(\text{id}_a) = T$ checks that id_f and id_a are exactly of the same type.

$$\text{LISTOPMSIMPLETOSIMPLE}$$
$$\frac{}{\Delta, \Delta_c, \mathcal{L}, \mathcal{L}_{ids} \vdash (\text{id}_f, \text{id}_a) \xrightarrow{list_{opm}} \mathcal{L} \cup \{id_f\}, \mathcal{L}_{ids} \cup \{id_a\}} \quad \begin{array}{l} \text{id}_f \notin \mathcal{L}, \text{id}_a \notin \mathcal{L}_{ids} \\ \text{id}_f \in Outs(\Delta_c) \\ \text{id}_a \in Sigs(\Delta) \cup Outs(\Delta) \\ \Delta_c(\text{id}_f) = \Delta(\text{id}_a) = T \end{array}$$

> **Side conditions**
>
> $Outs_c(\text{id}_f) = T$ and $Sigs(\text{id}_a) = \text{array}(T,n,m)$ checks that the type of id_f and the type of the elements of id_a are the same. Note that id_a be a signal identifier of the array type as id_f is mapped to one subelement of id_a.

LISTOPMSIMPLETOPARTIAL

$$\frac{SE_l(\text{e}_i) \quad \text{e}_i \xrightarrow{e} v_i \quad v_i \in_c nat(n,m)}{\Delta, \Delta_c, \mathcal{L}, \mathcal{L}_{ids} \vdash (\text{id}_f, \text{id}_a(\text{e}_i)) \xrightarrow{list_{opm}} \mathcal{L} \cup \{id_f\}, \mathcal{L}_{ids} \cup \{(id_a, v_i)\}}$$

$\text{id}_f \notin \mathcal{L}, \text{id}_a, (\text{id}_a, v_i) \notin \mathcal{L}_{ids}$
$\text{id}_f \in Outs(\Delta_c)$
$\text{id}_a \in Sigs(\Delta) \cup Outs(\Delta)$
$\Delta_c(\text{id}_f) = T$
$\Delta(\text{id}_a) = array(T,n,m)$

LISTOPMSIMPLETOOPEN

$$\frac{}{\Delta, \Delta_c, \mathcal{L}, \mathcal{L}_{ids} \vdash (\text{id}_f, \text{open}) \xrightarrow{list_{opm}} \mathcal{L} \cup \{id_f\}, \mathcal{L}_{ids}}$$

$\text{id}_f \notin \mathcal{L}$
$\text{id}_f \in Outs(\Delta_c)$

> **Remark 3** (Unconnected output port.). *We forbid the case where an indexed formal part corresponding to the subelement of a composite output port is unconnected, i.e.* $(\text{id}_f(\text{e}_i), \text{open})$, *as it could lead to the case where some subelements of a composite output port are connected while others are not (error case in [64, p.7]).*

LISTOPMPARTIALTOSIMPLE

$$\frac{SE_l(\text{e}_i) \quad \text{e}_i \xrightarrow{e} v_i \quad v_i \in_c nat(n,m)}{\Delta, \Delta_c, \mathcal{L}, \mathcal{L}_{ids} \vdash (\text{id}_f(\text{e}_i), \text{id}_a) \xrightarrow{list_{opm}} \mathcal{L} \cup \{(id_f, v_i)\}, \mathcal{L}_{ids} \cup \{id_a\}}$$

$\text{id}_f, (\text{id}_f, v_i) \notin \mathcal{L}, \text{id}_a \notin \mathcal{L}_{ids}$
$\text{id}_f \in Outs(\Delta_c)$
$\text{id}_a \in Sigs(\Delta) \cup Outs(\Delta)$
$\Delta_c(\text{id}_f) = array(T,n,m)$
$\Delta(\text{id}_a) = T$

LISTOPMPARTIALTOPARTIAL

$$\frac{\begin{array}{ccc} SE_l(\text{e}'_i) & \text{e}'_i \xrightarrow{e} v'_i & v'_i \in_c nat(n',m') \\ SE_l(\text{e}_i) & \text{e}_i \xrightarrow{e} v_i & v_i \in_c nat(n,m) \end{array}}{\Delta, \Delta_c, \mathcal{L}, \mathcal{L}_{ids} \vdash (\text{id}_f(\text{e}_i), \text{id}_a(\text{e}'_i)) \xrightarrow{list_{opm}} \begin{array}{c} \mathcal{L} \cup \{(id_f, v_i)\}, \\ \mathcal{L}_{ids} \cup \{(id_a, v'_i)\} \end{array}}$$

$\text{id}_f, (\text{id}_f, v_i) \notin \mathcal{L}, \text{id}_a, (\text{id}_a, v'_i) \notin \mathcal{L}_{ids}$
$\text{id}_f \in Outs(\Delta_c)$
$\text{id}_a \in Sigs(\Delta) \cup Outs(\Delta)$
$\Delta_c(\text{id}_f) = array(T,n,m)$
$\Delta(\text{id}_a) = array(T,n',m')$

LISTOPMCONS

$$\frac{\Delta, \Delta_c, \mathcal{L}, \mathcal{L}_{ids} \vdash \text{assoc}_{po} \xrightarrow{list_{opm}} \mathcal{L}', \mathcal{L}'_{ids} \quad \Delta, \Delta_c, \mathcal{L}', \mathcal{L}'_{ids} \vdash \text{opmap} \xrightarrow{list_{opm}} \mathcal{L}'', \mathcal{L}''_{ids}}{\Delta, \Delta_c, \mathcal{L}, \mathcal{L}_{ids} \vdash \text{assoc}_{po}, \text{opmap} \xrightarrow{list_{opm}} \mathcal{L}'', \mathcal{L}''_{ids}}$$

4.5.11 Valid sequential statements

The valid_{ss} predicate states that a sequential statement is well-typed in the context Δ, σ, Λ.

Well-typed signal assignment

> **Premises**
>
> – $\Delta, \sigma, \Lambda \vdash e \xrightarrow{e} v$ evaluates the expression assigned to signal id_s in the context Δ, σ, Λ.
>
> – $v \in_c T$ checks that the value of expression e conforms to the type of signal id_s.

WTSIG

$$\frac{\Delta, \sigma, \Lambda \vdash e \xrightarrow{e} v \quad v \in_c T \quad id_s \in \mathit{Sigs}(\Delta) \cup \mathit{Outs}(\Delta)}{\Delta, \sigma, \Lambda \vdash \mathtt{valid}_{ss}(id_s \Leftarrow e) \quad \Delta(id_s) = T}$$

WTIDXSIG

$$\frac{\begin{array}{c} \Delta, \sigma, \Lambda \vdash e \xrightarrow{e} v \qquad v \in_c T \\ \Delta, \sigma, \Lambda \vdash e_i \xrightarrow{e} v_i \quad v_i \in_c \mathit{nat}(n,m) \quad id_s \in \mathit{Sigs}(\Delta) \cup \mathit{Outs}(\Delta) \end{array}}{\Delta, \sigma, \Lambda \vdash \mathtt{valid}_{ss}(id_s(e_i) \Leftarrow e) \qquad \Delta(id_s) = \mathit{array}(T,n,m)}$$

Well-typed variable assignment

WTVAR

$$\frac{\Delta, \sigma, \Lambda \vdash e \xrightarrow{e} v \quad v \in_c T \quad id_v \in \Lambda}{\Delta, \sigma, \Lambda \vdash \mathtt{valid}_{ss}(id_v := e) \quad \Lambda(id_v) = (T, \mathit{val})}$$

WTIDXVAR

$$\frac{\begin{array}{c} \Delta, \sigma, \Lambda \vdash e \xrightarrow{e} v \qquad v \in_c T \\ \Delta, \sigma, \Lambda \vdash e_i \xrightarrow{e} v_i \quad v_i \in_c \mathit{nat}(n,m) \quad id_v \in \Lambda \end{array}}{\Delta, \Lambda \vdash \mathtt{valid}_{ss}(id_v(e_i) := e) \qquad \Lambda(id_v) = (\mathit{array}(T,n,m), \mathit{val})}$$

Well-typed if statements

WTIF

$$\frac{\Delta, \sigma, \Lambda \vdash e \xrightarrow{e} v \quad v \in_c \mathtt{bool} \quad \Delta, \sigma, \Lambda \vdash \mathtt{valid}_{ss}(ss)}{\Delta, \sigma, \Lambda \vdash \mathtt{valid}_{ss}(\mathtt{if}\ (e)\ ss)}$$

WTIFELSE

$$\frac{\begin{array}{c} \Delta, \sigma, \Lambda \vdash \mathtt{valid}_{ss}(ss) \\ \Delta, \sigma, \Lambda \vdash e \xrightarrow{e} v \quad v \in_c \mathtt{bool} \quad \Delta, \sigma, \Lambda \vdash \mathtt{valid}_{ss}(ss') \end{array}}{\Delta, \sigma, \Lambda \vdash \mathtt{valid}_{ss}(\mathtt{if}\ (e)\ ss\ ss')}$$

Well-typed loop statement

WTLOOP
$$\frac{\begin{array}{cc} \Delta, \sigma, \Lambda \vdash \mathrm{e} \xrightarrow{e} v & v \in_c nat(0, \mathtt{NATMAX}) \\ \Delta, \sigma, \Lambda \vdash \mathrm{e}' \xrightarrow{e} v' & v' \in_c nat(0, \mathtt{NATMAX}) \quad \Delta, \sigma, \Lambda' \vdash \mathtt{valid}_{ss}(\mathrm{ss}) \end{array}}{\Delta, \sigma, \Lambda \vdash \mathtt{valid}_{ss}(\mathtt{for}\,(\mathrm{id}_v, \mathrm{e}, \mathrm{e}')\,\mathrm{ss})} \; \Lambda' = \Lambda \cup (\mathrm{id}_v, (nat(v, v'), v))$$

Well-typed rising and falling edge blocks

WTRISING
$$\frac{\Delta, \sigma, \Lambda \vdash \mathtt{valid}_{ss}(\mathrm{ss})}{\Delta, \sigma, \Lambda \vdash \mathtt{valid}_{ss}(\mathtt{rising}\,\mathrm{ss})}$$

WTFALLING
$$\frac{\Delta, \sigma, \Lambda \vdash \mathtt{valid}_{ss}(\mathrm{ss})}{\Delta, \sigma, \Lambda \vdash \mathtt{valid}_{ss}(\mathtt{falling}\,\mathrm{ss})}$$

Well-typed rst blocks

WTRST
$$\frac{\Delta, \sigma, \Lambda \vdash \mathtt{valid}_{ss}(\mathrm{ss}) \quad \Delta, \sigma, \Lambda \vdash \mathtt{valid}_{ss}(\mathrm{ss}')}{\Delta, \sigma, \Lambda \vdash \mathtt{valid}_{ss}(\mathtt{rst}\,\mathrm{ss}\,\mathrm{ss}')}$$

Well-typed null statement

WTNULL
$$\frac{}{\Delta, \sigma, \Lambda \vdash \mathtt{valid}_{ss}(\mathtt{null})}$$

4.6 Simulation rules

In this section, we formalize a specific simulation algorithm for the *H*-VHDL designs. This algorithm is much simpler than the one presented in the LRM. This is mostly due to the fact that *H*-VHDL is a subset of VHDL that aims at the description of synthesizable and synchronous designs. Synthesizable designs mean that the only kind of signal assignment used to describe the design behaviors are δ-delay signal assignments. Leaving apart the synchronous side, we only need a simulation algorithm that performs *delta cycles* (see Section 4.1.2) to simulate such synthesizable designs. However, *H*-VHDL designs are also synchronous designs. As such, a *H*-VHDL design is equipped with a clock input port. The value of the clock input port changes from 0 to 1 and inversely at a constant rate, i.e. the clock rate. One can see the changing of the value of the clock input port as the result of the execution of a unit-delay signal assignment where the time clause is equal to half the clock period. Listing 4.4 illustrates how a *H*-VHDL design tl can be embedded in another top-level design alongside a process regulating the value of a clock signal by using a unit-delay signal assignment. Listing 4.4 presents the behavioral part of the embedding top-level design.

```
1  architecture toplevel_arch of toplevel is
2  begin
3
4    clkp : process (clock)
5    begin
6      clock ⇐ not clock after τ -- where τ is half a clock period
7    end process clkp;
8
9    id_tl : entity tl
10   generic map (...)
11   port map (clock ⇒ clock, ...);
12
13 end toplevel_arch;
```

LISTING 4.4: An architecture to simulate a synchronous design. The architecture `toplevel_arch` is composed of the `clkp` process, that simulates a clock signal, and of an instance of the design `tl` named id_{tl}, i.e. the design under simulation.

In Listing 4.4, the `clkp` process assigns the clock signal with its inverse value after τ unit of time where τ corresponds to half the clock period. Of course, the clock period is specified by the designer of the circuit. The component instance id_{tl} corresponds to the instantiation of the $\mathcal{H}$-VHDL design `tl`, i.e. the one we want to simulate. The `clock` input port of id_{tl} is connected to the `clock` signal of the embedding design in id_{tl}'s port map. Thus, when the value of the clock signal changes every half clock period, the processes that react to the changes of the clock signal, i.e. the so-called *synchronous* processes, are executed in the internal behavior of the component instance id_{tl}. Then, it is the turn of *combinational* processes to be executed until stabilization of all signal values. Using the terms of the LRM simulation algorithm, what will happen when trying to simulate the design of Listing 4.4 will be an alternation between one time cycle to move to the next clock event and execute synchronous processes, followed by many delta cycles corresponding to the execution of combinational processes until stabilization. Thus, we choose to embed this alternation within the definition of our simulation algorithm.

We must add a last element to the definition of our simulation algorithm. The top-level design generated by the HILECOP transformation interacts with its environment through the input ports. The input ports of a top-level design are called *primary* input ports. In our simulation algorithm, we need to represent the capture and the injection of the values of primary input ports and how this affects the values of the internal signals of the simulated design.

Finally, Algorithm 2 gives an overview of our simulation algorithm in a pseudo-code language. This simulation algorithm is formalized in a small-step semantics style in the following sections. Here, we say small-step semantics because the different intermediary states of the design under simulation are detailed and registered in a simulation trace θ. This simulation trace is built incrementally through the execution of simulation cycles, and is returned at the end of Algorithm 2. However, the execution of sequential statements in the body of processes are expressed with a big-step operational semantics.

Algorithm 2: Simulation(Δ, σ_e, cs, E_p, T_c)

   // Initialization phase.

1 $\sigma'_e \leftarrow$ RunAllOnce(Δ,σ_e,cs)

2 $\sigma \leftarrow$ Stabilize(Δ,σ'_e,cs)

   // Main loop.

3 $\theta \leftarrow [\sigma]$

4 **while** $T_c > 0$ **do**

5 $\sigma_i \leftarrow$ Inject(Δ,σ,E_p,T_c)

6 $\sigma_\uparrow \leftarrow$ RisingEdge(Δ,σ_i,cs)

7 $\sigma' \leftarrow$ Stabilize(Δ,$\sigma_\uparrow$,cs)

8 $\sigma_\downarrow \leftarrow$ FallingEdge(Δ,σ',cs)

9 $\sigma \leftarrow$ Stabilize(Δ,$\sigma_\downarrow$,cs)

10 $\theta \leftarrow \theta + \!\!\!+\, [\sigma',\sigma]$

11 $T_c \leftarrow T_c - 1$

12 **return** θ

Algorithm 2 defines an elaborated design Δ and a default design state σ_e as parameters. We assume that they are the result of the elaboration of the design being simulated. *cs* corresponds to the behavior of the design, i.e. the one that will be executed during the simulation. E_p is the environment that will provide values to the primary input ports; it is a function that maps the set of input port identifiers to values. T_c corresponds to the number of simulation cycles to be performed. Algorithm 2 begins with an initialization phase (following the LRM simulation algorithm); all processes are run exactly once (Line 1) followed by a stabilization phase (Line 2, multiple delta cycles). Line 3 initializes the variable θ with a singleton list holding state σ, i.e. the initial simulation state. Then, the same loop is performed until T_c reaches zero. First, the values of primary input ports are retrieved from the environment E_p at the current time value T_c; this is performed by the Inject function; then, all parts of *cs* that react to the rising edge (resp. falling edge) of the clock signal are executed; finally, the combinational parts of *cs* are executed until stabilization of all signals. At Line 10, the states obtained after the rising edge phase (i.e. σ') and after the falling edge phase (i.e. σ) are appended to the simulation trace θ. Note that we only register stable states in the simulation trace. At the end of the simulation cycle, the parameter T_c is decremented. After the execution of all simulation cycles, Algorithm 2 returns the simulation trace.

4.6.1 Full simulation

The full simulation process is decomposed in two steps. The first step is the elaboration phase that builds an elaborated version of a H-VHDL design along with its default state. Previous to the elaboration phase, the top-level design receives a value for each of its generic constant; we refer to it as the *dimensioning* of the top-level design. The second step is the simulation phase that executes the behavioral part of the top-level design, and yields a simulation/execution trace; this step has been presented through the Algorithm 2.

The *full* simulation relation, defined by the FULLSIM rule, formalizes the full simulation process for a given $\mathcal{H}$-VHDL design. The relation holds eight parameters, namely: a top-level design d, a design store $\mathcal{D} \in id \nrightarrow design$, an elaborated design $\Delta \in ElDesign$, a dimensioning function $\mathcal{M}_g \in Gens(\Delta) \nrightarrow value$, a simulation environment $E_p \in \mathbb{N} \to (Ins(\Delta) \to value)$, a simulation cycle count $\tau \in \mathbb{N}$, an initial state $\sigma_0 \in \Sigma$, and a simulation trace $\theta \in \texttt{list}(\Sigma)$, corresponding to the list of states yielded by the simulation of design d during τ cycles. Note that we use the pointed notation to access the behavioral part of design d, written d.cs. It is this part of the design that is executed during the simulation, and therefore is passed as a parameter of the initialization and simulation relations.

> **Premises**
>
> - $\mathcal{M}_g \in Gens(\Delta) \nrightarrow value$, the function yielding the values of generic constants for a given top-level design, referred to as the *dimensioning* function. Here, $Gens(\Delta)$ is a short-hand notation for the domain of $Gens(\Delta)$, normally written $\texttt{dom}(Gens(\Delta))$, i.e. the set of generic constant identifiers of Δ.
>
> - $E_p \in \mathbb{N} \to (Ins(\Delta) \to value)$, the function yielding a mapping from primary inputs (i.e, input ports of the top-level design) to values at a given simulation cycle count. Here, $Ins(\Delta)$ is a shorthand notation for the domain of $Ins(\Delta)$, normally written $\texttt{dom}(Ins(\Delta))$, i.e. the set of input port identifiers of Δ.
>
> - τ, the number of simulation cycles to execute. The value of τ is decremented at each clock cycle until it reaches zero (see Section 4.6.2).

FULLSIM
$$\frac{\mathcal{D}, \mathcal{M}_g \vdash \text{d} \xrightarrow{elab} \Delta, \sigma \quad \mathcal{D}, \Delta, \sigma \vdash \text{d.cs} \xrightarrow{init} \sigma_0 \quad \mathcal{D}, E_p, \Delta, \tau, \sigma_0 \vdash \text{d.cs} \to \theta}{\mathcal{D}, \Delta, \mathcal{M}_g, E_p, \tau \vdash \text{d} \xrightarrow{full} (\sigma_0 :: \theta)}$$

Our simulation algorithm aims at representing the execution of a hardware system in the presence of an environment. Thus, we need to make some hypotheses regarding the relation between the environment and the clock signal defining the operating frequency of the modeled system:

> **Hypothesis 1** (Stable primary inputs). *The values of primary inputs (i.e, input ports of the top-level design) are captured at the beginning of a clock cycle, and thus remain stable (i.e, their values do not change) during a whole clock cycle.*

Hypothesis 1 arises from the fact that the clock signal sample rate respects the Nyquist-Shannon sampling theorem. Therefore, the sample rate of the design's clock is sufficient to capture all events possibly arising in the environment. We only need to settle the values of the primary inputs at the beginning of a clock cycle.

Also, after each clock event phase follows a signal stabilization phase in the proceedings of a simulation cycle. One more hypothesis is needed here:

Hypothesis 2 (Stabilization). *All signals have enough time to stabilize during the signal stabilization phase that happens between two clock events.*

As a $\mathcal{H}$-VHDL design represents a physical circuit, one can assume that the represented circuit is analyzed former to the simulation. Therefore, the analysis tells us exactly how much time is needed to propagate signal values through the longest physical path; as a consequence, a proper clock frequency is set ensuring signal stabilization between two clock events. Thus, Hypothesis 2 arises from the latter facts.

4.6.2 Simulation loop

The following rules define the $\mathcal{H}$-VHDL simulation relation. The $\mathcal{H}$-VHDL simulation relation associates the execution of a behavior *cs* with a simulation trace θ in a context $\mathcal{D}, E_p, \Delta, \tau, \sigma$. The simulation trace θ is the result of the execution of the design behavior *cs* during τ cycles. In the case where τ is equal to zero (Rule SIMEND), the execution of *cs* returns an empty trace. In the case where τ is greater than zero (Rule SIMLOOP), one simulation cycle is performed from the starting state σ and returns the two states: σ', the state in the middle of the clock cycle (i.e. after a rising edge phase), and σ'', the state at the end of the clock cycle (i.e. after a falling edge phase). Then, the $\mathcal{H}$-VHDL simulation relation calls itself recursively with a decremented cycle count. The recursive call yields a trace θ which is then appended to the states σ' and σ'' to form the final simulation trace.

$$
\text{SIMEND} \atop \overline{\mathcal{D}, E_p, \Delta, 0, \sigma \vdash cs \rightarrow [\,]}
$$

$$
\text{SIMLOOP} \atop \frac{\mathcal{D}, E_p, \Delta, \tau, \sigma \vdash cs \xrightarrow{\uparrow, \downarrow} \sigma', \sigma'' \qquad \mathcal{D}, E_p, \Delta, \tau - 1, \sigma'' \vdash cs \rightarrow \theta}{\mathcal{D}, E_p, \Delta, \tau, \sigma \vdash cs \rightarrow (\sigma' :: \sigma'' :: \theta)} \; \tau > 0
$$

4.6.3 Simulation cycle

To ease the reading of forward simulation rules, we need to introduce two notations.

Notation 7 (Overriding union). *For all partial function $f, f' \in X \nrightarrow Y$, $f \overleftarrow{\cup} f'$ denotes the overriding union of f and f' such that $f \overleftarrow{\cup} f'(x) = \begin{cases} f'(x) & \text{if } x \in \text{dom}(f') \\ f(x) & \text{otherwise} \end{cases}$*

Notation 8 (Differentiated intersection domain). *For all partial function $f, f' \in X \nrightarrow Y$, $f \overset{\neq}{\cap} f'$ denotes the intersection of the domain of f and f' for which f and f' yields different values. That is, $f \overset{\neq}{\cap} f' = \{\, x \in \text{dom}(f) \cap \text{dom}(f') \mid f(x) \neq f'(x) \,\}$.*

> **Definition 34** (Input port values update). *Given an elaborated design Δ and a simulation environment $E_p \in \mathbb{N} \to (Ins(\Delta) \to value)$, let us define the relation expressing the update of the value of input ports at a given design state $\sigma \in \Sigma$ and clock cycle count $\tau \in \mathbb{N}$, thus resulting in a new state $\sigma_i \in \Sigma$. The relation is written $\texttt{Inject}(\sigma, E_p, \tau, \sigma_i)$ and verifies that:*
>
> $\sigma = <\mathcal{S}, \mathcal{C}, \mathcal{E}>$ and $\sigma_i = <\mathcal{S} \overset{\leftarrow}{\cup} E_p(\tau), \mathcal{C}, \mathcal{E}>$.

The $\mathcal{H}$-VHDL simulation cycle relation, written $\xrightarrow{\uparrow,\downarrow}$, is defined through the only Rule SIM-CYC. It states that the design states σ' and σ'' are the result of the execution of the design behavior *cs* over one simulation cycle, this starting from state σ. Here, σ' is the state obtained in the middle of the clock cycle, i.e. after the rising edge phase and the first stabilization phase, and σ'' is the state obtained at the end of the clock cycle, i.e. after the falling edge phase and the second stabilization phase. As mentioned in Hypothesis 1, the update of the value of input ports is performed at each clock event. New input port values are coming from the environment E_p. The update is made through the definitions of state σ_i which is qualified in the side condition by the $\texttt{Inject}$ relation.

SIMCYC

$$\frac{\mathcal{D}, \Delta, \sigma_i \vdash cs \xrightarrow{\uparrow} \sigma_\uparrow \quad \mathcal{D}, \Delta, \sigma_\uparrow \vdash cs \xrightarrow{\leadsto} \sigma' \quad \mathcal{D}, \Delta, \sigma' \vdash cs \xrightarrow{\downarrow} \sigma_\downarrow \quad \mathcal{D}, \Delta, \sigma_\downarrow \vdash cs \xrightarrow{\leadsto} \sigma''}{\mathcal{D}, E_p, \Delta, \tau, \sigma \vdash cs \xrightarrow{\uparrow,\downarrow} \sigma', \sigma''} \quad \texttt{Inject}(\sigma, E_p, \tau, \sigma_i)$$

4.6.4 Initialization rules

The *init* relation, defined through the Rule INIT, describes the initialization phase of the $\mathcal{H}$-VHDL simulation algorithm. It produces an initial simulation state σ_0 by executing the design behavior cs in the context $\mathcal{D}, \Delta, \sigma$.

INIT

$$\frac{\mathcal{D}, \Delta, \sigma \vdash cs \xrightarrow{runinit} \sigma' \quad \mathcal{D}, \Delta, \sigma' \vdash cs \xrightarrow{\leadsto} \sigma_0}{\mathcal{D}, \Delta, \sigma \vdash cs \xrightarrow{init} \sigma_0}$$

During the initialization phase, each process is executed exactly once. This is formalized by the *runinit* relation. Then, a stabilization phase follows, formalized by the *stabilize* relation, written $\xrightarrow{\leadsto}$. The initialization phase triggers the execution of the first part of reset blocks. A reset block (`rst ss ss'`) is equivalent to (`if rst = '0'then ss else ss' end if;`). Therefore, when considering a (`rst ss ss'`) block, the *runinit* relation always executes the ss block; at every other moment of the simulation, the ss' block is executed. This corresponds to the conventional execution of a hardware system where a *reset* signal set to false triggers the initialization of the system, and then is set to true for the rest of the execution.

The *runinit* relation is defined by the Rules PSRUNINIT, COMPRUNINIT, PARRUNINIT and NULLRUNINIT which are detailed right below. The *stabilize* relation is defined in Section 4.6.6.

Evaluation of a process statement

The PSRUNINIT rule describes the execution of a process statement during the initialization phase. The execution of a process statement comes down to the execution of the process statement body. The result of the execution is a new state σ'.

Premises

– The i flag of the ss_i relation indicates that all sequential statements responding to the initialization phase (i.e, reset blocks) will be executed.

– The ss_i relation takes two states in its context, i.e. two σ. The first σ is the state used to evaluate expressions appearing in the process statement body; the second σ is the state that will be modified by the execution of signal assignment statements.

Side conditions

The local environment Λ used to execute the body of the process id_p is retrieved from the Ps sub-environment of the elaborated design Δ.

PSRUNINIT

$$\frac{\Delta, \sigma, \sigma, \Lambda \vdash \mathrm{ss} \xrightarrow{ss_i} \sigma', \Lambda'}{\mathcal{D}, \Delta, \sigma \vdash \mathtt{process}\ (\mathrm{id}_p, \mathrm{sl}, \mathrm{vars}, \mathrm{ss}) \xrightarrow{runinit} \sigma'} \quad \Delta(\mathrm{id}_p) = \Lambda$$

Evaluation of a component instantiation statement

Rule COMPRUNINIT describes the execution of a component instantiation statement during the initialization phase. The execution of a component instantiation statement is divided in three phases. First, the input ports of the component instance receive new values through the evaluation of the component instance's input port map. Second, the internal behavior of the component instance is evaluated; this evaluation possibly modifies the value of the internal signals and the output ports of the component instance. Finally, through the evaluation of its output port map, the component instance propagates the value of its output ports to the signals of the embedding design.

Premises

– The *mapip* relation evaluates the input port map i of id_c, thus modifying the internal state σ_c of id_c. The result is a new internal state σ_c'.

– The expression $\mathcal{D}(\mathrm{id}_e).\mathrm{cs}$ refers to the internal behavior of the component instance id_c.

– State σ_c'' is the new internal state of component instance id_c resulting from the execution of its internal behavior.

> – The *mapop* relation evaluates the output port map o of id_c, thus modifying the state σ of the embedding design. The result is a new embedding design state σ'.

Side conditions

– Δ_c is the elaborated version of the component instance id_c referenced in the *Comps* sub-environment of the embedding design Δ, i.e. $\Delta(id_c) = \Delta_c$.

– σ_c is the internal design state of the component instance id_c referenced in the component store of state σ, i.e. $\sigma(id_c) = \sigma_c$.

– The component store $\mathcal{C}''$ of state σ'' is equal to the component store $\mathcal{C}'$ of state σ' where the component instance id_c is assigned to its new internal state σ_c''.

– The expression $\mathcal{C} \stackrel{\neq}{\cap} \mathcal{C}''$ equals $\{\text{id}_c\}$ if the internal state of the component instance id_c has changed after the evaluation of its input port map and its internal behavior. In other words, we register the component instance id_c as an eventful component instance if $\sigma_c \neq \sigma_c''$.

COMPRUNINIT

$$
\frac{
\begin{array}{c}
\Delta, \Delta_c, \sigma, \sigma_c \vdash \text{i} \xrightarrow{\textit{mapip}} \sigma_c' \\
\mathcal{D}, \Delta_c, \sigma_c' \vdash \mathcal{D}(\text{id}_e).\text{cs} \xrightarrow{\textit{runinit}} \sigma_c'' \\
\Delta, \Delta_c, \sigma, \sigma_c'' \vdash \text{o} \xrightarrow{\textit{mapop}} \sigma'
\end{array}
}{
\mathcal{D}, \Delta, \sigma \vdash \text{comp} \ (\text{id}_c, \text{id}_e, \text{g}, \text{i}, \text{o}) \xrightarrow{\textit{runinit}} \sigma''
}
\quad
\begin{array}{l}
\text{id}_e \in \mathcal{D} \\
\Delta(\text{id}_c) = \Delta_c, \sigma(\text{id}_c) = \sigma_c \\
\sigma'' = \,<\mathcal{S}', \mathcal{C}'', \mathcal{E}' \cup (\mathcal{C} \stackrel{\neq}{\cap} \mathcal{C}'')> \\
\mathcal{C}'' = \mathcal{C}'(\text{id}_c) \leftarrow \sigma_c''
\end{array}
$$

Evaluation of the composition of concurrent statements

Rule PARRUNINIT describes the evaluation of the parallel composition of two concurrent statements cs and cs'. The two concurrent statements are evaluated starting from the same state σ, and they generate two different state σ' and σ''. The state resulting from the concurrent execution of cs and cs' is the result of a merging between the starting state σ, and the two states σ' and σ''.

PARRUNINIT

$$
\frac{
\mathcal{D}, \Delta, \sigma \vdash \text{cs} \xrightarrow{\textit{runinit}} \sigma' \quad \mathcal{D}, \Delta, \sigma \vdash \text{cs}' \xrightarrow{\textit{runinit}} \sigma''
}{
\mathcal{D}, \Delta, \sigma \vdash \text{cs} \, || \, \text{cs}' \xrightarrow{\textit{runinit}} \text{merge}(\sigma, \sigma', \sigma'')
}
\quad \mathcal{E}' \cap \mathcal{E}'' = \varnothing
$$

The merge function, defined in Listing 4.5 in pseudo-Coq language, computes a new state based on the original state σ, and the states σ' and σ'' yielded by the computation of two concurrent statements. In the resulting state, the signal value store $\mathcal{S}_m$ is a function merging

together the signal stores of states σ, σ' and σ''. S_m yields values from the signal store $\mathcal{S}'$ (resp. $\mathcal{S}''$) for all signal that belongs to the set of events at state σ' (resp. σ''), and yields values from the original signal store $\mathcal{S}$ for all eventless signals. The same goes for the merged component store C_m. The new set of events $\mathcal{E}_m$ is the union between the set of events at state σ' and state σ''. The merge function correctly merges the state σ, σ' and σ'' only if the set of events of σ' and σ'' are disjoint. The PARRUNINIT rule, which appeals to the merge function, defines the condition of disjoint set of events as a side condition.

```
1  Definition merge(σ, σ′, σ″) :=
2      let σ = <S,C,E> in
3      let σ′ = <S′,C′,E′> in
4      let σ″ = <S″,C″,E″> in
```
$$5 \quad \text{let } \mathcal{S}_m(\text{id}) = \begin{cases} \mathcal{S}'(\text{id}) & \text{if id} \in \mathcal{E}' \\ \mathcal{S}''(\text{id}) & \text{if id} \in \mathcal{E}'' \text{ in} \\ \mathcal{S}(\text{id}) & \textit{otherwise} \end{cases}$$
$$6 \quad \text{let } \mathcal{C}_m(\text{id}) = \begin{cases} \mathcal{C}'(\text{id}) & \text{if id} \in \mathcal{E}' \\ \mathcal{C}''(\text{id}) & \text{if id} \in \mathcal{E}'' \text{ in} \\ \mathcal{C}(\text{id}) & \textit{otherwise} \end{cases}$$
```
7      let Eₘ = E′ ∪ E″ in <Sₘ,Cₘ,Eₘ>.
```

LISTING 4.5: The merge function that fuses together an origin state σ, with two states σ' and σ'' generated by the execution of two $\mathcal{H}$-VHDL concurrent statements.

Remark 4 (No multiply-driven signals). *For all states $\sigma' = <\mathcal{S}',\mathcal{C}',\mathcal{E}'>$ and $\sigma'' = <\mathcal{S}'',\mathcal{C}'',\mathcal{E}''>$ resulting from the execution of two $\mathcal{H}$-VHDL concurrent statements, $\mathcal{E}' \cap \mathcal{E}'' = \varnothing$ must be enforced. Otherwise, there are some multiply-driven signals, which are forbidden in our semantics.*

Rule NULLRUNINIT evaluates a null statement during the initialization phase. The evaluation of a null statement yields a state similar to the starting state.

NULLRUNINIT

$$\frac{}{\Delta, \sigma \vdash \texttt{null} \xrightarrow{runinit} \sigma}$$

4.6.5   Clock phases rules

The following rules express the evaluation of concurrent statements at clock phases, i.e. the rising edge ($\uparrow$) and the falling edge ($\downarrow$) phases. The clock signal, which triggers the evaluation of synchronous process statements, is represented by the reserved signal identifier `clk`. Thus, synchronous processes are processes containing the `clk` signal in their sensitivity list.

Evaluation of a process statement

The following rules describe the evaluation of a process statement at the occurrence of the rising or the falling edge of the clock signal. In the case where a process does not contain the `clk` identifier in its sensitivity list, then its statement body is not executed during the clock phases (see Rules PSRENOCLK and PSFENOCLK). Otherwise, its statement body is executed. Depending on the considered clock event, falling blocks or rising blocks are executed when encountered in the body of a process (see Rules PSRECLK and PSFECLK).

PSRENOCLK

$$\frac{}{\mathcal{D}, \Delta, \sigma \vdash \texttt{process} \ (\text{id}_p, \ \text{sl}, \ \text{vars}, \ \text{ss}) \xrightarrow{\uparrow} \sigma} \quad \text{clk} \notin \text{sl}$$

> **Premises**
>
> The ↑ flag in the $ss_\uparrow$ relation indicates that `rising` blocks will be executed.

PSRECLK

$$\frac{\Delta, \sigma, \sigma, \Lambda \vdash \text{ss} \xrightarrow{ss_\uparrow} \sigma', \Lambda'}{\mathcal{D}, \Delta, \sigma \vdash \texttt{process} \ (\text{id}_p, \ \text{sl}, \ \text{vars}, \ \text{ss}) \xrightarrow{\uparrow} \sigma'} \quad \begin{array}{l} \text{clk} \in \text{sl} \\ \Delta(\text{id}_p) = \Lambda \end{array}$$

PSFENOCLK

$$\frac{}{\mathcal{D}, \Delta, \sigma \vdash \texttt{process} \ (\text{id}_p, \ \text{sl}, \ \text{vars}, \ \text{ss}) \xrightarrow{\downarrow} \sigma} \quad \text{clk} \notin \text{sl}$$

> **Premises**
>
> The ↓ flag in the $ss_\downarrow$ relation indicates that `falling` blocks will be executed.

PSFECLK

$$\frac{\Delta, \sigma, \sigma, \Lambda \vdash \text{ss} \xrightarrow{ss_\downarrow} \sigma', \Lambda'}{\mathcal{D}, \Delta, \sigma \vdash \texttt{process} \ (\text{id}_p, \ \text{sl}, \ \text{vars}, \ \text{ss}) \xrightarrow{\downarrow} \sigma'} \quad \begin{array}{l} \text{clk} \in \text{sl} \\ \Delta(\text{id}_p) = \Lambda \end{array}$$

Evaluation of a component instantiation statement

The following rules describe the evaluation of a component instantiation statement during clock phases. These rules are similar in every point to Rule COMPRUNINIT that describes the evaluation of a component instantiation statement during the initialization phase. The only difference lies in the execution of the internal behavior of the component instance. During the clock phases, the falling relation, written $\xrightarrow{\downarrow}$, or the rising relation, written $\xrightarrow{\uparrow}$, evaluate the internal behavior of component instances.

COMPRE

$$\Delta, \Delta_c, \sigma, \sigma_c \vdash \mathrm{i} \xrightarrow{mapip} \sigma_c'$$

$$\mathcal{D}, \Delta_c, \sigma_c' \vdash \mathcal{D}(id_e).\mathrm{cs} \xrightarrow{\uparrow} \sigma_c'' \qquad id_e \in \mathcal{D}$$

$$\underline{\Delta, \Delta_c, \sigma, \sigma_c'' \vdash \mathrm{o} \xrightarrow{mapop} \sigma'} \qquad \Delta(id_c) = \Delta_c, \sigma(id_c) = \sigma_c$$

$$\mathcal{D}, \Delta, \sigma \vdash \mathtt{comp}\ (id_c, id_e, \mathrm{g}, \mathrm{i}, \mathrm{o}) \xrightarrow{\uparrow} \sigma'' \qquad \sigma'' = <\mathcal{S}', \mathcal{C}'', \mathcal{E}' \cup (\mathcal{C} \overset{\neq}{\cap} \mathcal{C}'')>$$

$$\mathcal{C}'' = \mathcal{C}'(id_c) \leftarrow \sigma_c''$$

COMPFE

$$\Delta, \Delta_c, \sigma, \sigma_c \vdash \mathrm{i} \xrightarrow{mapip} \sigma_c'$$

$$\mathcal{D}, \Delta_c, \sigma_c' \vdash \mathcal{D}(id_e).\mathrm{cs} \xrightarrow{\downarrow} \sigma_c'' \qquad id_e \in \mathcal{D}$$

$$\underline{\Delta, \Delta_c, \sigma, \sigma_c'' \vdash \mathrm{o} \xrightarrow{mapop} \sigma'} \qquad \Delta(id_c) = \Delta_c, \sigma(id_c) = \sigma_c$$

$$\mathcal{D}, \Delta, \sigma \vdash \mathtt{comp}\ (id_c, id_e, \mathrm{g}, \mathrm{i}, \mathrm{o}) \xrightarrow{\downarrow} \sigma'' \qquad \sigma'' = <\mathcal{S}', \mathcal{C}'', \mathcal{E}' \cup (\mathcal{C} \overset{\neq}{\cap} \mathcal{C}'')>$$

$$\mathcal{C}'' = \mathcal{C}'(id_c) \leftarrow \sigma_c''$$

Evaluation of the composition of concurrent statements

The following rules describe the evaluation of the composition of concurrent statements and the evaluation of null statements during the clock phases. These rules are similar to the ones described for the initialization phase. Thus, the reader can refer to Section 4.6.4 for more details.

PARFE

$$\underline{\mathcal{D}, \Delta, \sigma \vdash \mathrm{cs} \xrightarrow{\downarrow} \sigma' \qquad \mathcal{D}, \Delta, \sigma \vdash \mathrm{cs}' \xrightarrow{\downarrow} \sigma''} \qquad \mathcal{E}' \cap \mathcal{E}'' = \varnothing$$

$$\mathcal{D}, \Delta, \sigma \vdash \mathrm{cs}\ ||\ \mathrm{cs}' \xrightarrow{\downarrow} \mathtt{merge}(\sigma, \sigma', \sigma'')$$

NULLFE

$$\overline{\Delta, \sigma \vdash \mathtt{null} \xrightarrow{\downarrow} \sigma}$$

PARRE

$$\underline{\mathcal{D}, \Delta, \sigma \vdash \mathrm{cs} \xrightarrow{\uparrow} \sigma' \qquad \mathcal{D}, \Delta, \sigma \vdash \mathrm{cs}' \xrightarrow{\uparrow} \sigma''} \qquad \mathcal{E}' \cap \mathcal{E}'' = \varnothing$$

$$\mathcal{D}, \Delta, \sigma \vdash \mathrm{cs}\ ||\ \mathrm{cs}' \xrightarrow{\uparrow} \mathtt{merge}(\sigma, \sigma', \sigma'')$$

NULLRE

$$\overline{\Delta, \sigma \vdash \mathtt{null} \xrightarrow{\uparrow} \sigma}$$

4.6.6 Stabilization rules

The following rules describe the evaluation of concurrent statements, representing a design's behavior, during a stabilization phase. The stabilization phase triggers the execution of the combinational parts of the behavior by appealing to the *comb* relation. When the execution of the combinational parts of the behavior does not change the design state anymore, then we have reached a stable state and the stabilization phase ends (Rule STABILIZEEND). When the execution of the combinational parts produces some events, i.e. it changes the value of signals or the internal state of component instances, then the stabilization phase must continue until a stable state is reached (Rule STABILIZELOOP). In the formalization of the H-VHDL simulation

algorithm, the set of events of a design state is useful to merge the states resulting from the execution of multiple concurrent statements (see Definition 4.5), and to determine if a stable state has been reached. In the LRM simulation algorithm, the kernel process uses the set of events to resume the activity of processes. If one of the signal declared in a process' sensitivity list is registered in the current set of events, then the process body must be executed. We choose to disregard this aspect of the execution of processes in the formalization of our simulation algorithm. Thus, all combinational processes are executed when a delta cycle is performed, regardless of the intersection between the set of events and the sensitivity lists.

> **Side conditions**
>
> – In Rule STABILIZEEND, state σ is an eventless state, i.e. its event set $\mathcal{E}$ is empty.
>
> – In Rule STABILIZELOOP, state σ' is an eventful state and state σ'' is eventless.

STABILIZEEND

$$\frac{\mathcal{D}, \Delta, \sigma \vdash \text{cs} \xrightarrow{comb} \sigma}{\mathcal{D}, \Delta, \sigma \vdash \text{cs} \xrightarrow{\leadsto} \sigma} \quad \mathcal{E} = \varnothing$$

STABILIZELOOP

$$\frac{\mathcal{D}, \Delta, \sigma \vdash \text{cs} \xrightarrow{comb} \sigma' \quad \mathcal{D}, \Delta, \sigma' \vdash \text{cs} \xrightarrow{\leadsto} \sigma''}{\mathcal{D}, \Delta, \sigma \vdash \text{cs} \xrightarrow{\leadsto} \sigma''} \quad \begin{array}{l} \mathcal{E} \neq \varnothing \\ \mathcal{E}'' = \varnothing \end{array}$$

Evaluation of a process statement

Rule PSCOMB describes the execution of a process statement during a stabilization phase. Even synchronous processes can be executed during a stabilization phase, however, the falling and rising blocks are not interpreted. Thus, the evaluation of a *purely* synchronous process, defined only with falling or rising blocks and no combinational parts, does not change the design state during a stabilization phase.

> **Premises**
>
> – The c flag (for *combinational*) on the ss_c relation indicates that statements responding to clock events (i.e. `falling` and `rising` blocks) and statements executed during the initialization phase only (i.e. `rst` blocks) will not be considered.
>
> – The set of events of state σ is emptied ($NoEv(\sigma)$, see Notation 6) before the evaluation of the process statement body. It corresponds to the consumption of the information brought by the event set. Once the information has been consumed, new events can be generated by executing the process body. Otherwise, if the set of events is never emptied, then a stable state might never be reached.

PSCOMB

$$\frac{\Delta, \sigma, NoEv(\sigma), \Lambda \vdash \text{ss} \xrightarrow{ss_c} \sigma', \Lambda'}{\mathcal{D}, \Delta, \sigma \vdash \texttt{process (id}_p\texttt{, sl, vars, ss)} \xrightarrow{comb} \sigma'} \quad \Delta(\text{id}_p) = \Lambda$$

Evaluation of a component instantiation statement

Rule COMPCOMB describes the evaluation of a component instantiation statement during a stabilization phase. This rule is similar in every point to Rule COMPRUNINIT, and Rules COMPRE and COMPFE, that describe the evaluation of a component instantiation statement during the initialization phase, and the clock phases. The only difference lies in the execution of the internal behavior of the component instance. During a stabilization, the *comb* relation evaluates the internal behavior of component instances. Otherwise, see Section 4.6.4 for more details about the premises and side conditions of Rule COMPCOMB.

COMPCOMB

$$\Delta, \Delta_c, \sigma, \sigma_c \vdash \text{i} \xrightarrow{mapip} \sigma_c'$$

$$\mathcal{D}, \Delta_c, \sigma_c' \vdash \mathcal{D}(id_e).\text{cs} \xrightarrow{comb} \sigma_c'' \qquad id_e \in \mathcal{D}$$

$$\Delta, \Delta_c, NoEv(\sigma), \sigma_c'' \vdash \text{o} \xrightarrow{mapop} \sigma' \qquad \Delta(id_c) = \Delta_c, \sigma(id_c) = \sigma_c$$

$$\frac{}{\mathcal{D}, \Delta, \sigma \vdash \text{comp } (id_c, id_e, \text{g}, \text{i}, \text{o}) \xrightarrow{comb} \sigma''} \qquad \sigma'' = <\mathcal{S}', \mathcal{C}'', \mathcal{E}' \cup (\mathcal{C} \overset{\neq}{\cap} \mathcal{C}'')>$$

$$\mathcal{C}'' = \mathcal{C}'(id_c) \leftarrow \sigma_c''$$

Evaluation of the composition of concurrent statements

The following rules describe the evaluation of the composition of concurrent statements and the evaluation of null statements during a stabilization phase. These rules are similar to the ones describe for the initialization phase. Thus, the reader can refer to Section 4.6.4 for more details.

PARCOMB

$$\frac{\mathcal{D}, \Delta, \sigma \vdash \text{cs} \xrightarrow{comb} \sigma' \qquad \mathcal{D}, \Delta, \sigma \vdash \text{cs}' \xrightarrow{comb} \sigma''}{\mathcal{D}, \Delta, \sigma \vdash \text{cs} \| \text{cs}' \xrightarrow{comb} \text{merge}(\sigma, \sigma', \sigma'')} \qquad \mathcal{E}' \cap \mathcal{E}'' = \varnothing$$

NULLCOMB

$$\frac{}{\Delta, \sigma \vdash \text{null} \xrightarrow{comb} NoEv(\sigma)}$$

4.6.7   Evaluation of input and output port maps

Evaluation of an input port map

Here, we define the *mapip* relation that evaluates the input port map of a component instance. For each association of the input port map, the actual part is evaluated and the result is assigned to the formal part of the association, i.e. an input port (Rule MAPIPSIMPLE) or an indexed input port (Rule MAPIPPARTIAL) identifier. The following rules define the *mapip* relation.

MAPIPSIMPLE

$$\frac{\Delta, \sigma \vdash \text{e} \xrightarrow{e} v \qquad v \in_c T}{\Delta, \Delta_c, \sigma, \sigma_c \vdash (id_s, \text{e}) \xrightarrow{mapip} <\mathcal{S}', \mathcal{C}, \mathcal{E}>} \qquad \begin{aligned} &\Delta_c(id_s) = T \\ &\sigma_c = <\mathcal{S}, \mathcal{C}, \mathcal{E}> \\ &\mathcal{S}' = \mathcal{S}(id_s) \leftarrow v \end{aligned}$$

MAPIPPARTIAL

$$\frac{\begin{array}{cc} \Delta, \sigma \vdash e \xrightarrow{e} v & v \in_c T \\ e_i \xrightarrow{e} v_i & v_i \in_c nat(n, m) \end{array}}{\Delta, \Delta_c, \sigma, \sigma_c \vdash (\text{id}_s(e_i), e) \xrightarrow{mapip} <\mathcal{S}', \mathcal{C}, \mathcal{E}>} \quad \begin{array}{l} \Delta_c(\text{id}_s) = array(T, n, m) \\ \sigma_c = <\mathcal{S}, \mathcal{C}, \mathcal{E}> \\ \mathcal{S}' = \mathcal{S}(id_s) \leftarrow \text{set\_at}(v, v_i, \mathcal{S}(id_s)) \end{array}$$

MAPIPCOMP

$$\frac{\Delta, \Delta_c, \sigma, \sigma_c \vdash \text{assoc}_{ip} \xrightarrow{mapip} \sigma'_c \quad \Delta, \Delta_c, \sigma, \sigma'_c \vdash \text{ipmap} \xrightarrow{mapip} \sigma''_c}{\Delta, \Delta_c, \sigma, \sigma_c \vdash \text{assoc}_{ip}, \text{ipmap} \xrightarrow{mapip} \sigma''_c}$$

Evaluation of an output port map

Here, we define the *mapop* relation that evaluates the output port map of a component instance. For each association of the output port map, the formal part is evaluated and the result is assigned to the actual part of the association. There can be five kinds of associations in an output port map:

– an output port identifier (of the component instance) is associated with the open keyword, thus denoting an unconnected output port in the output interface of the component instance

– an output port identifier is associated with an internal signal or an output port identifier of the embedding design (Rule MAPOPSIMPLETOSIMPLE)

– an output port identifier is associated with an indexed internal signal or an indexed output port identifier of the embedding design (Rule MAPOPSIMPLETOPARTIAL)

– an indexed output port identifier is associated with an internal signal or an output port identifier of the embedding design (Rule MAPOPPARTIALTOSIMPLE)

– an indexed output port identifier is associated with an indexed internal signal or an indexed output port identifier of the embedding design (Rule MAPOPPARTIALTOPARTIAL)

> **Remark 5** (Out ports and *e*)**.** *We can not use the e relation to interpret the values of output ports, because output ports are write-only constructs. We append the flag o to the e relation (i.e, e_o) to enable the evaluation of output port identifiers as regular signal identifier expressions.*

The e_o relation is only defined to retrieve the value of output ports from a store signal $\mathcal{S}$ under a design state $\sigma = <\mathcal{S}, \mathcal{C}, \mathcal{E}>$.

$$\frac{\text{OUTO}}{\Delta, \sigma \vdash \text{id}_s \xrightarrow{e_o} \sigma(\text{id}_s)} \quad \begin{array}{l} \text{id}_s \in Outs(\Delta) \\ \text{id}_s \in \sigma \end{array}$$

$$\frac{\text{IDXOUTO}}{\text{e}_i \xrightarrow{e} v_i \quad v_i \in_c nat(n,m)} \quad \begin{array}{l} \text{id}_s \in Outs(\Delta) \\ \text{id}_s \in \sigma \\ \Delta, \sigma \vdash \text{id}_s(\text{e}_i) \xrightarrow{e_o} \texttt{get\_at}(v_i, \sigma(\text{id}_s)) \end{array} \quad \begin{array}{l} \text{id}_s \in \sigma \\ \Delta(\text{id}_s) = array(T,n,m) \end{array}$$

The following rules define the *mapop* relation.

MAPOPOPEN

$$\overline{\Delta, \Delta_c, \sigma, \sigma_c \vdash (\text{id}_f, \texttt{open}) \xrightarrow{mapop} \sigma}$$

> **Side conditions**
>
> In the signal store $\mathcal{S}'$, value v is assigned to the signal identifier id_a. If this assignment changes the value of id_a, then an event on signal id_a must be registered. The expression $\mathcal{E} \cup \mathcal{S} \stackrel{\neq}{\cap} \mathcal{S}'$ represents the set of signals that have a different value in signal store $\mathcal{S}$ and $\mathcal{S}'$.

MAPOPSIMPLETOSIMPLE

$$\frac{\Delta_c, \sigma_c \vdash \text{id}_f \xrightarrow{e_o} v \quad v \in_c T}{\Delta, \Delta_c, \sigma, \sigma_c \vdash (\text{id}_f, \text{id}_a) \xrightarrow{mapop} <\mathcal{S}', \mathcal{C}, \mathcal{E}'>} \quad \begin{array}{l} \text{id}_a \in Sigs(\Delta) \cup Outs(\Delta) \\ \Delta(\text{id}_a) = T \\ \sigma = <\mathcal{S}, \mathcal{C}, \mathcal{E}> \\ \mathcal{S}' = \mathcal{S}(id_a) \leftarrow v, \mathcal{E}' = \mathcal{E} \cup (\mathcal{S} \stackrel{\neq}{\cap} \mathcal{S}') \end{array}$$

MAPOPSIMPLETOPARTIAL

$$\frac{\begin{array}{cc} \text{e}_i \xrightarrow{e} v_i & \\ \Delta_c, \sigma_c \vdash \text{id}_f \xrightarrow{e_o} v & \begin{array}{c} v \in_c T \\ v_i \in_c nat(n,m) \end{array} \end{array}}{\Delta, \Delta_c, \sigma, \sigma_c \vdash (\text{id}_f, \text{id}_a(\text{e}_i)) \xrightarrow{mapop} <\mathcal{S}', \mathcal{C}, \mathcal{E}'>} \quad \begin{array}{l} \text{id}_a \in Sigs(\Delta) \cup Outs(\Delta) \\ \Delta(\text{id}_a) = array(T,n,m) \\ \sigma = <\mathcal{S}, \mathcal{C}, \mathcal{E}> \\ \mathcal{S}' = \mathcal{S}(id_a) \leftarrow \texttt{set\_at}(v, v_i, \mathcal{S}(id_a)) \\ \mathcal{E}' = \mathcal{E} \cup (\mathcal{S} \stackrel{\neq}{\cap} \mathcal{S}') \end{array}$$

MAPOPPARTIALTOSIMPLE

$$\frac{\Delta_c, \sigma_c \vdash \text{id}_f(\text{e}_i') \xrightarrow{e_o} v \quad v \in_c T}{\Delta, \Delta_c, \sigma, \sigma_c \vdash (\text{id}_f(\text{e}_i'), \text{id}_a) \xrightarrow{mapop} <\mathcal{S}', \mathcal{C}, \mathcal{E}'>} \quad \begin{array}{l} \text{id}_a \in Sigs(\Delta) \cup Outs(\Delta) \\ \Delta(\text{id}_a) = T \\ \sigma = <\mathcal{S}, \mathcal{C}, \mathcal{E}> \\ \mathcal{S}' = \mathcal{S}(id_a) \leftarrow v, \mathcal{E}' = \mathcal{E} \cup (\mathcal{S} \stackrel{\neq}{\cap} \mathcal{S}') \end{array}$$

MAPOPPARTIALTOPARTIAL

$$\frac{\begin{array}{cc} \text{e}_i \xrightarrow{e} v_i & \\ \Delta_c, \sigma_c \vdash \text{id}_f(\text{e}_i') \xrightarrow{e_o} v & \begin{array}{c} v \in_c T \\ v_i \in_c nat(n,m) \end{array} \end{array}}{\Delta, \Delta_c, \sigma, \sigma_c \vdash (\text{id}_f(\text{e}_i'), \text{id}_a(\text{e}_i)) \xrightarrow{mapop} <\mathcal{S}', \mathcal{C}, \mathcal{E}'>} \quad \begin{array}{l} \text{id}_a \in Sigs(\Delta) \cup Outs(\Delta) \\ \Delta(\text{id}_a) = array(T,n,m) \\ \sigma = <\mathcal{S}, \mathcal{C}, \mathcal{E}> \\ \mathcal{S}' = \mathcal{S}(id_a) \leftarrow \texttt{set\_at}(v, v_i, \mathcal{S}(id_a)) \\ \mathcal{E}' = \mathcal{E} \cup (\mathcal{S} \stackrel{\neq}{\cap} \mathcal{S}') \end{array}$$

MAPOPCOMP

$$\frac{\Delta, \Delta_c, \sigma, \sigma_c \vdash \text{assoc}_{po} \xrightarrow{mapop} \sigma' \qquad \Delta, \Delta_c, \sigma', \sigma_c \vdash \text{opmap} \xrightarrow{mapop} \sigma''}{\Delta, \Delta_c, \sigma, \sigma_c \vdash \text{assoc}_{po}, \text{opmap} \xrightarrow{mapop} \sigma''}$$

4.6.8 Evaluation of sequential statements

Here, we define the *ss* relation that evaluates the sequential statements composing the body of processes. The phases of a simulation cycle affect the evaluation of sequential statements. For instance, reset blocks are only evaluated during an initialization phase, falling blocks during a falling edge phase... Thus, we append a specific flag to the *ss* relation to enable the evaluation of specific sequential statements at particular phases of the simulation cycle. There are four different flags, the *c* flag to denote the execution of combinational statements only, the *i* flag to enable the execution of reset blocks, the $\uparrow$ (resp. $\downarrow$) flag to enable the execution of rising (resp. falling) blocks. Writing the *ss* relation with no flag indicates that the evaluation of a given sequential statement is the same for every phase of the simulation cycle. A flag is passed from the conclusion to the premises when a sequential statement is composed of inner sequential blocks.

Signal assignment statement

A signal assignment generates a new design state with a modified signal store and a new set of events. Note that there are two states on the left side of the thesis symbol. State σ represents the state holding the current values of signals (i.e. the *reading* state), and state σ_w holds the new values of signals (i.e. the *written* state).

> **Side conditions**
>
> The expression $\mathcal{S} \stackrel{\neq}{\cap} \mathcal{S}'_w$ registers signal id_s as an eventful signal if its value after assignment, i.e. in the signal store $\mathcal{S}'_w$, is different from its current value at state σ, i.e. in the signal store $\mathcal{S}$.

SIGASSIGN

$$\frac{\Delta, \sigma, \Lambda \vdash e \xrightarrow{e} v \qquad v \in_c T}{\Delta, \sigma, \sigma_w, \Lambda \vdash id_s \Leftarrow e \xrightarrow{ss} <\mathcal{S}'_w, \mathcal{C}_w, \mathcal{E}'_w>, \Lambda} \quad \begin{array}{l} id_s \in Sigs(\Delta) \cup Outs(\Delta) \\ \Delta(id_s) = T \\ \mathcal{S}'_w = \mathcal{S}_w(id_s) \leftarrow v \\ \mathcal{E}'_w = \mathcal{E}_w \cup (\mathcal{S} \stackrel{\neq}{\cap} \mathcal{S}'_w) \end{array}$$

IDXSIGASSIGN

$$\frac{\begin{array}{ll} \Delta, \sigma, \Lambda \vdash e_i \xrightarrow{e} v_i & v \in_c T \\ \Delta, \sigma, \Lambda \vdash e \xrightarrow{e} v & v_i \in_c nat(n, m) \end{array}}{\Delta, \sigma, \sigma_w, \Lambda \vdash id_s(e_i) \Leftarrow e \xrightarrow{ss} <\mathcal{S}'_w, \mathcal{C}_w, \mathcal{E}'_w>, \Lambda} \quad \begin{array}{l} id_s \in Sigs(\Delta) \cup Outs(\Delta) \\ \Delta(id_s) = array(T, n, m) \\ \mathcal{S}'_w = \mathcal{S}_w(id_s) \leftarrow \texttt{set\_at}(v, v_i, \mathcal{S}_w(id_s)) \\ \mathcal{E}'_w = \mathcal{E}_w \cup (\mathcal{S} \stackrel{\neq}{\cap} \mathcal{S}'_w) \end{array}$$

Remark 6 (Signal assignments). *Let us take the example of a synchronous process that performs a swap between the value of two signals s_1 and s_2 at the falling edge of the clock signal* `clk`. *This process is implemented by the following listing written in VHDL concrete syntax:*

```
swap : process(clk)
begin
  if falling_edge(clk) then
    s₁ ⇐ s₂;
    s₂ ⇐ s₁;
  end if;
end process swap;
```

In the above listing, the two signal assignment statements must be considered as being executed in parallel, even though they are written as a sequence. Thus, when the statement $s_2 \Leftarrow s_1$ *is evaluated, the value to consider for signal* s_1 *is not the one resulting from the execution of statement* $s_1 \Leftarrow s_2$ *but the one at the beginning of the process evaluation. For these reasons, we must include a reading state and a written state in the environment of the ss relation.*

Variable assignment statement

A variable assignment statement modifies the value of a variable defined in a local environment Λ. Contrary to the case of signal assignments, a sequence of variable assignment statements are to be considered as a real sequence, and not as being executed in parallel.

VARASSIGN

$$\frac{\Delta, \sigma, \Lambda \vdash \text{e} \xrightarrow{e} v \quad v \in_c T}{\Delta, \sigma, \sigma_w, \Lambda \vdash \text{id}_v := \text{e} \xrightarrow{ss} \sigma_w, \Lambda(id_v) \leftarrow (T, v)} \quad \begin{array}{l} \text{id}_v \in \Lambda \\ \Lambda(\text{id}_v) = (T, val) \end{array}$$

IDXVARASSIGN

$$\frac{\begin{array}{cc} \Delta, \sigma, \Lambda \vdash \text{e}_i \xrightarrow{e} v_i & v_i \in_c nat(n, m) \\ \Delta, \sigma, \Lambda \vdash \text{e} \xrightarrow{e} v & v \in_c T \end{array}}{\Delta, \sigma, \sigma_w, \Lambda \vdash \text{id}_v(\text{e}_i) := \text{e} \xrightarrow{ss} \sigma_w, \Lambda(id_v) \leftarrow (T, \texttt{set\_at}(v, v_i, val))} \quad \begin{array}{l} \text{id}_v \in \Lambda \\ \Lambda(\text{id}_v) = (array(T, n, m), val) \end{array}$$

Remark 7 (Local variables and persistent values). *In the LRM, the value of local variables is persistent through the multiple execution of a process. However, in the definition of the* `place` *and* `transition` *designs, and in the VHDL programs generated by HILECOP, all local variables are initialized by an assignment statement at the beginning of the body of processes. Thus, to simplify the $\mathcal{H}$-VHDL semantics, we choose not to consider local variables as persistent memory as their values are renewed at each execution of a process.*

If statement

Here, we present the classical evaluation of if and if-else statements.

IF⊤

$$\dfrac{\Delta,\sigma,\Lambda \vdash e \xrightarrow{e} \top \quad \Delta,\sigma,\sigma_w,\Lambda \vdash ss \xrightarrow{ss} \sigma'_w,\Lambda'}{\Delta,\sigma,\sigma_w,\Lambda \vdash \texttt{if } (e) \text{ ss} \xrightarrow{ss} \sigma'_w,\Lambda'}$$

IF⊥

$$\dfrac{\Delta,\sigma,\Lambda \vdash e \xrightarrow{e} \bot}{\Delta,\sigma,\sigma_w,\Lambda \vdash \texttt{if } (e) \text{ ss} \xrightarrow{ss} \sigma_w,\Lambda}$$

IFELSE⊤

$$\dfrac{\Delta,\sigma,\Lambda \vdash e \xrightarrow{e} \top \quad \Delta,\sigma,\sigma_w,\Lambda \vdash ss \xrightarrow{ss} \sigma'_w,\Lambda'}{\Delta,\sigma,\sigma_w,\Lambda \vdash \texttt{if } (e) \text{ ss ss}' \xrightarrow{ss} \sigma'_w,\Lambda'}$$

IFELSE⊥

$$\dfrac{\Delta,\sigma,\Lambda \vdash e \xrightarrow{e} \bot \quad \Delta,\sigma,\sigma_w,\Lambda \vdash ss' \xrightarrow{ss} \sigma'_w,\Lambda'}{\Delta,\sigma,\sigma_w,\Lambda \vdash \texttt{if } (e) \text{ ss ss}' \xrightarrow{ss} \sigma'_w,\Lambda'}$$

Loop statement

Here, we present the classical evaluation of for-loop statements. Rule LOOPINIT corresponds to the evaluation of a for loop in the case where the range variable id_v is not already defined in the local environment; in that case, the loop range bounds are evaluated and binding between the variable id_v and its type and initial value is added to the local environment; finally the loop statement is re-evaluated with the updated local environment. Rule LOOP⊥ denotes the evaluation of a loop statement in the case where the range variable id_v has not yet reached the upper bound of the loop range. Rule LOOP⊤ denotes the evaluation of a loop statement in the opposite case.

LOOP⊥

$$\dfrac{\Delta,\sigma,\Lambda_i \vdash id_v = e' \xrightarrow{e} \bot \quad \dfrac{\Delta,\sigma,\sigma_w,\Lambda_i \vdash ss \xrightarrow{ss} \sigma'_w,\Lambda'}{\Delta,\sigma,\sigma'_w,\Lambda' \vdash \texttt{for } (id_v,e,e') \text{ ss} \xrightarrow{ss} \sigma''_w,\Lambda''}}{\Delta,\sigma,\sigma_w,\Lambda \vdash \texttt{for } (id_v,e,e') \text{ ss} \xrightarrow{ss} \sigma''_w,\Lambda''} \begin{array}{l} id_v \in \Lambda \\ \Lambda(id_v) = (T,val) \\ \Lambda_i = \Lambda(id_v) \leftarrow (T,val+1) \end{array}$$

LOOP⊤

$$\dfrac{\Delta,\sigma,\Lambda_i \vdash id_v = e' \xrightarrow{e} \top}{\Delta,\sigma,\sigma_w,\Lambda \vdash \texttt{for } (id_v,e,e') \text{ ss} \xrightarrow{ss} \sigma_w,\Lambda \setminus (id_v,\Lambda(id_v))} \begin{array}{l} id_v \in \Lambda \\ \Lambda(id_v) = (T,val) \\ \Lambda_i = \Lambda(id_v) \leftarrow (T,val+1) \end{array}$$

LOOPINIT

$$\dfrac{\begin{array}{l} \Delta,\sigma,\Lambda \vdash e \xrightarrow{e} v \\ \Delta,\sigma,\Lambda \vdash e' \xrightarrow{e} v' \quad \Delta,\sigma,\sigma_w,\Lambda_i \vdash \texttt{for } (id_v,e,e') \text{ ss} \xrightarrow{ss} \sigma'_w,\Lambda' \end{array}}{\Delta,\sigma,\sigma_w,\Lambda \vdash \texttt{for } (id_v,e,e') \text{ ss} \xrightarrow{ss} \sigma'_w,\Lambda'} \begin{array}{l} id_v \notin \Lambda \\ \Lambda_i = \Lambda \cup (id_v,(nat(v,v'),v)) \end{array}$$

Rising and falling edge block statements

Here, we define the execution of rising and falling blocks. Rising (resp. Falling) blocks are executed only during a rising (resp. falling) edge phase of a simulation cycle, i.e. when the flag $\uparrow$ (resp. $\downarrow$) is raised (Rule RISINGEGDEEXEC and FALLINGEDGEEXEC). Otherwise, the evaluation of these blocks is without effect on state σ_w and on the local environment Λ (Rules RISINGEDGEDEFAULT and FALLINGEDGEDEFAULT).

RISINGEDGEDEFAULT
$$\frac{}{\Delta, \sigma, \sigma_w, \Lambda \vdash \texttt{rising ss} \xrightarrow{ss_f} \sigma_w, \Lambda} \quad \begin{array}{l} f \neq \uparrow \\ f \in \{\downarrow, i, o\} \end{array}$$

FALLINGEDGEDEFAULT
$$\frac{}{\Delta, \sigma, \sigma_w, \Lambda \vdash \texttt{falling ss} \xrightarrow{ss_f} \sigma_w, \Lambda} \quad \begin{array}{l} f \neq \downarrow \\ f \in \{\uparrow, i, o\} \end{array}$$

RISINGEDGEEXEC
$$\frac{\Delta, \sigma, \sigma_w, \Lambda \vdash ss \xrightarrow{ss_\uparrow} \sigma'_w, \Lambda'}{\Delta, \sigma, \sigma_w, \Lambda \vdash \texttt{rising ss} \xrightarrow{ss_\uparrow} \sigma'_w, \Lambda'}$$

FALLINGEDGEEXEC
$$\frac{\Delta, \sigma, \sigma_w, \Lambda \vdash ss \xrightarrow{ss_\downarrow} \sigma'_w, \Lambda'}{\Delta, \sigma, \sigma_w, \Lambda \vdash \texttt{falling ss} \xrightarrow{ss_\downarrow} \sigma'_w, \Lambda'}$$

Rst block statement

Here, we define the evaluation of reset blocks. The first part of reset blocks is only evaluated during the initialization phase of a simulation, i.e. when the i flag is raised (Rule RSTEXEC). Otherwise, it is the second part of the reset block that is evaluated (Rule RSTDEFAULT). Remember that a reset block is the transcription of an if-else statement specifically devised for the H-VHDL abstract syntax.

RSTDEFAULT
$$\frac{\Delta, \sigma, \sigma_w, \Lambda \vdash ss' \xrightarrow{ss_f} \sigma'_w, \Lambda'}{\Delta, \sigma, \sigma_w, \Lambda \vdash \texttt{rst ss ss'} \xrightarrow{ss_f} \sigma'_w, \Lambda'} \quad \begin{array}{l} f \neq i \\ f \in \{\uparrow, \downarrow, o\} \end{array}$$

RSTEXEC
$$\frac{\Delta, \sigma, \sigma_w, \Lambda \vdash ss \xrightarrow{ss_i} \sigma'_w, \Lambda'}{\Delta, \sigma, \sigma_w, \Lambda \vdash \texttt{rst ss ss'} \xrightarrow{ss_i} \sigma'_w, \Lambda'}$$

Composition of sequential statements and null statement

Here, we present the evaluation of the composition of sequential statements (Rule SEQSTMT) and of the null sequential statement (Rule NULLSTMT). When evaluating a sequence of statements, the same state σ holding the current value of signals is used to execute both part of the sequence. The written state σ_w is modified by the first part of the sequence, thus resulting in a state σ'_w. Then, σ'_w is used to evaluate the second part of the sequence.

SEQSTMT
$$\frac{\Delta, \sigma, \sigma_w, \Lambda \vdash ss \xrightarrow{ss} \sigma'_w, \Lambda' \quad \Delta, \sigma, \sigma'_w, \Lambda' \vdash ss' \xrightarrow{ss} \sigma''_w, \Lambda''}{\Delta, \sigma, \sigma_w, \Lambda \vdash \texttt{ss; ss'} \xrightarrow{ss} \sigma''_w, \Lambda''}$$

NULLSTMT
$$\frac{}{\Delta, \sigma, \sigma_w, \Lambda \vdash \texttt{null} \xrightarrow{ss} \sigma_w, \Lambda}$$

4.6.9   Evaluation of expressions

Here, we present the evaluation of expressions used throughout the definition of $\mathcal{H}$-VHDL designs. Rules NAT, FALSE and TRUE describe the evaluation of natural number and boolean constants. Rule AGGREG describes the evaluation of an aggregate expression. Rule GEN presents the evaluation of a generic constant identifier. Rules SIG and VAR describe the evaluation of signal and variable identifiers. Rules IDXSIG and IDXVAR correspond to the evaluation of indexed signal and indexed variable identifiers. In Rules IDXSIG and IDXVAR, $\mathtt{get\_at}(i, a)$ is a function returning the i-th element of array a.

$$\frac{\text{NAT}}{\Delta, \sigma, \Lambda \vdash \mathtt{n} \xrightarrow{e} n} \quad \begin{array}{l} n \in \mathbb{N} \\ n \leq \mathtt{NATMAX} \end{array} \qquad \frac{\text{FALSE}}{\Delta, \sigma, \Lambda \vdash \mathtt{false} \xrightarrow{e} \bot} \qquad \frac{\text{TRUE}}{\Delta, \sigma, \Lambda \vdash \mathtt{true} \xrightarrow{e} \top}$$

$$\frac{\text{AGGREG}}{\Delta, \sigma, \Lambda \vdash \mathtt{e}_i \xrightarrow{e} v_i} \quad i = 1, \dots, n$$
$$\frac{}{\Delta, \sigma, \Lambda \vdash (\mathtt{e}_1, \dots, \mathtt{e}_n) \xrightarrow{e} (v_1, \dots, v_n)}$$

$$\frac{\text{SIG}}{\Delta, \sigma, \Lambda \vdash \mathrm{id}_s \xrightarrow{e} \sigma(\mathrm{id}_s)} \quad \mathrm{id}_s \in Sigs(\Delta) \cup Ins(\Delta) \qquad \frac{\text{VAR}}{\Delta, \sigma, \Lambda \vdash \mathrm{id}_v \xrightarrow{e} v} \quad \begin{array}{l} \mathrm{id}_v \in \Lambda \\ \Lambda(\mathrm{id}_v) = (T, v) \end{array}$$

$$\frac{\text{GEN}}{\Delta, \sigma, \Lambda \vdash \mathrm{id}_g \xrightarrow{e} v} \quad \begin{array}{l} \mathrm{id}_g \in Gens(\Delta) \\ \Delta(\mathrm{id}_g) = (T, v) \end{array}$$

$$\frac{\text{IDXSIG}}{\Delta, \sigma, \Lambda \vdash \mathtt{e}_i \xrightarrow{e} v_i \quad v_i \in_c nat(n, m)}{\Delta, \sigma, \Lambda \vdash \mathrm{id}_s(\mathtt{e}_i) \xrightarrow{e} \mathtt{get\_at}(i, \sigma(\mathrm{id}_s))} \quad \begin{array}{l} \mathrm{id}_s \in Sigs(\Delta) \cup Ins(\Delta) \\ \Delta(\mathrm{id}_s) = array(T, n, m) \\ i = v_i \bmod n \end{array}$$

$$\frac{\text{IDXVAR}}{\Delta, \sigma, \Lambda \vdash \mathtt{e}_i \xrightarrow{e} v_i \quad v_i \in_c nat(n, m)}{\Delta, \sigma, \Lambda \vdash \mathrm{id}_v(\mathtt{e}_i) \xrightarrow{e} \mathtt{get\_at}(i, v)} \quad \begin{array}{l} \mathrm{id}_v \in \Lambda \\ \Lambda(\mathrm{id}_v) = (array(T, n, m), v) \\ i = v_i \bmod n \end{array}$$

Rule NATADD describe the evaluation of the addition between two expressions of the natural type. The operator $+_{\mathbb{N}}$ denotes the addition operator of natural numbers in the semantic world. We add as a side condition that the result of the addition between two natural numbers must not exceed the value of the $\mathtt{NATMAX}$ number (the greatest natural number representable in $\mathcal{H}$-VHDL). Rule NATSUB describes the evaluation of the substraction between two expressions of the natural type. Rule ORDOP describes the evaluation of the comparison between two

expressions of the natural type. The result of the comparison is a Boolean value. Rules BOOL-BINOP and NOTOP describes the evaluation of Boolean expressions. Rules EQOP and DIFFOP define the evaluation of the equality and difference between two expressions of the same type; the result is a Boolean value. Rule PARENTH describes the evaluation of a parenthesized expression.

NATADD

$$\frac{\Delta, \sigma, \Lambda \vdash \mathrm{e} \xrightarrow{e} v \quad \Delta, \sigma, \Lambda \vdash \mathrm{e'} \xrightarrow{e} v'}{\Delta, \sigma, \Lambda \vdash \mathrm{e} + \mathrm{e'} \xrightarrow{e} v +_{\mathbb{N}} v'} \; v +_{\mathbb{N}} v' \leq \mathtt{NATMAX}$$

NATSUB

$$\frac{\Delta, \sigma, \Lambda \vdash \mathrm{e} \xrightarrow{e} v \quad \Delta, \sigma, \Lambda \vdash \mathrm{e'} \xrightarrow{e} v'}{\Delta, \sigma, \Lambda \vdash \mathrm{e} - \mathrm{e'} \xrightarrow{e} v -_{\mathbb{N}} v'} \; v \geq v'$$

ORDOP

$$\frac{\Delta, \sigma, \Lambda \vdash \mathrm{e} \xrightarrow{e} v \quad \Delta, \sigma, \Lambda \vdash \mathrm{e'} \xrightarrow{e} v'}{\Delta, \sigma, \Lambda \vdash \mathrm{e} \; \mathrm{op}_{ordn} \; \mathrm{e'} \xrightarrow{e} v \; op_{ord\mathbb{N}} \; v'} \; \mathrm{op}_{ordn} \in \{<, \leq, >, \geq\}$$

BOOLBINOP

$$\frac{\Delta, \sigma, \Lambda \vdash \mathrm{e} \xrightarrow{e} v \quad \Delta, \sigma, \Lambda \vdash \mathrm{e'} \xrightarrow{e} v'}{\Delta, \sigma, \Lambda \vdash \mathrm{e} \; \mathrm{op}_{bool} \; \mathrm{e'} \xrightarrow{e} v \; op_{\mathbb{B}} \; v'} \; \mathrm{op}_{bool} \in \{\mathtt{and}, \mathtt{or}\}$$

NOTOP

$$\frac{\Delta, \sigma, \Lambda \vdash \mathrm{e} \xrightarrow{e} v}{\Delta, \sigma, \Lambda \vdash \mathtt{not} \; \mathrm{e} \xrightarrow{e} \neg v}$$

EQOP

$$\frac{\Delta, \sigma, \Lambda \vdash \mathrm{e} \xrightarrow{e} v \quad \Delta, \sigma, \Lambda \vdash \mathrm{e'} \xrightarrow{e} v'}{\Delta, \sigma, \Lambda \vdash \mathrm{e} = \mathrm{e'} \xrightarrow{e} eq(v, v')}$$

DIFFOP

$$\frac{\Delta, \sigma, \Lambda \vdash \mathrm{e} = \mathrm{e'} \xrightarrow{e} v}{\Delta, \sigma, \Lambda \vdash \mathrm{e} \neq \mathrm{e'} \xrightarrow{e} \neg v}$$

PARENTH

$$\frac{\Delta, \sigma, \Lambda \vdash \mathrm{e} \xrightarrow{e} v}{\Delta, \sigma, \Lambda \vdash (\mathrm{e}) \xrightarrow{e} v}$$

In Rule EQOP, *eq* is the equality operator established for all types defined in the semantics. In the definition of *eq*, two natural numbers and two Booleans are compared with the Leibniz equality. Two values of an array type are equal if the sub-elements sharing the same index are equal w.r.t. the definition of the *eq* relation. Thus, to be equal, the two arrays must be of the same size.

4.7   An example of full simulation

In this section, we will illustrate the full simulation of a H-VHDL top-level design on the example of Listing 4.6. The aim here is to give some derivations of the formal rules composing the

$\mathcal{H}$-VHDL semantics. Listing 4.6 is the result of the transformation of the SITPN model presented in Figure 4.6 into a $\mathcal{H}$-VHDL design.

To keep the examples within a reasonable size, Listing 4.6, and the other listings and derivation trees used in this section, refer to the generic constants, the ports and the internal signals of the `transition` and `place` designs by their short names. See Table D.1 for a correspondence between the short names and the full names of constants and signals of the `place` and `transition` designs. In Listing, the generated $\mathcal{H}$-VHDL design, named `tl`, declares its input and output ports at Line 7, and its internal signals at Line 10. The behavior of `tl` is defined by a `place` component instance id_p (Lines 15 to 23), a `transition` component instance id_t (Lines 28 to 36), and two processes, namely: the `actions` process (Lines 41 to 43), and the `functions` process (Lines 48 to 50).

```
1   design tl tla
2
3   -- Generic constants
4   ∅
5
6   -- Ports (ports_tl)
7   ((in, id_c0, boolean), (out, id_f0, boolean), (out, id_a0, boolean))
8
9   -- Declared (internal) signals (sigs_tl)
10  ((id_ft, boolean), (id_av, boolean), (id_rt, boolean),(id_m, boolean))
11
12  -- Behavior (cs_tl)
13
14  -- Place component instance id_p
15  comp (id_p, place,
16  -- Generic map
17  ((ian, 1), (oan, 1), (mm, 1)),
18
19  -- Input port map
20  ((im, 1), (iaw(0), 1), (oat(0), 0), (oaw(0), 1), (itf(0), id_ft), (otf(0), id_ft))
21
22  -- Output port map
23  ((oav(0), id_av), (pauths, open), (rtt(0), id_rt), (marked, id_m)))
24
25  ||
26
27  -- Transition component instance id_t
28  comp (id_t, transition,
29  -- Generic map
30  ((tt, 0), (ian, 1), (cn, 1)),
31
32  -- Input port map
33  ((ic(0), id_c0), (A, 0), (B, 0), (iav(0), id_av), (rt(0), id_rt), (pauths(0), true)),
34
35  -- Output port map
```

```
36   ((fired, id_ft)))

37

38   ||

39

40   -- The actions process
41   process (actions, {clk}, ∅,
42   (rst (id_a0 ⇐ false)
43        (falling (id_a0 ⇐ id_m or false))))

44

45   ||

46

47   -- The functions process
48   process (functions, {clk}, ∅,
49   (rst (id_f0 ⇐ false)
50        (rising (id_f0 ⇐ id_ft or false))))
```

LISTING 4.6: An example of $\mathcal{H}$-VHDL top-level design generated by the HILECOP transformation.
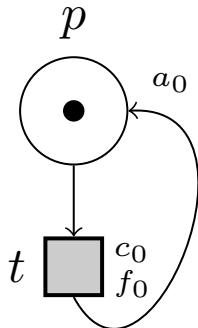
FIGURE 4.6: The SITPN model at the base of the generation of the top-level design presented in Listing 4.6.

Figure 4.7 is a graphical transcription of the top-level design of Listing 4.6. In Figure 4.7, note that we are representing the clk and rst in the interface of the tl design and also in the interfaces of tl's subcomponents, even though these two ports do not appear in the port clause of the tl design. These ports are considered as natively included in the port interface of all $\mathcal{H}$-VHDL designs.
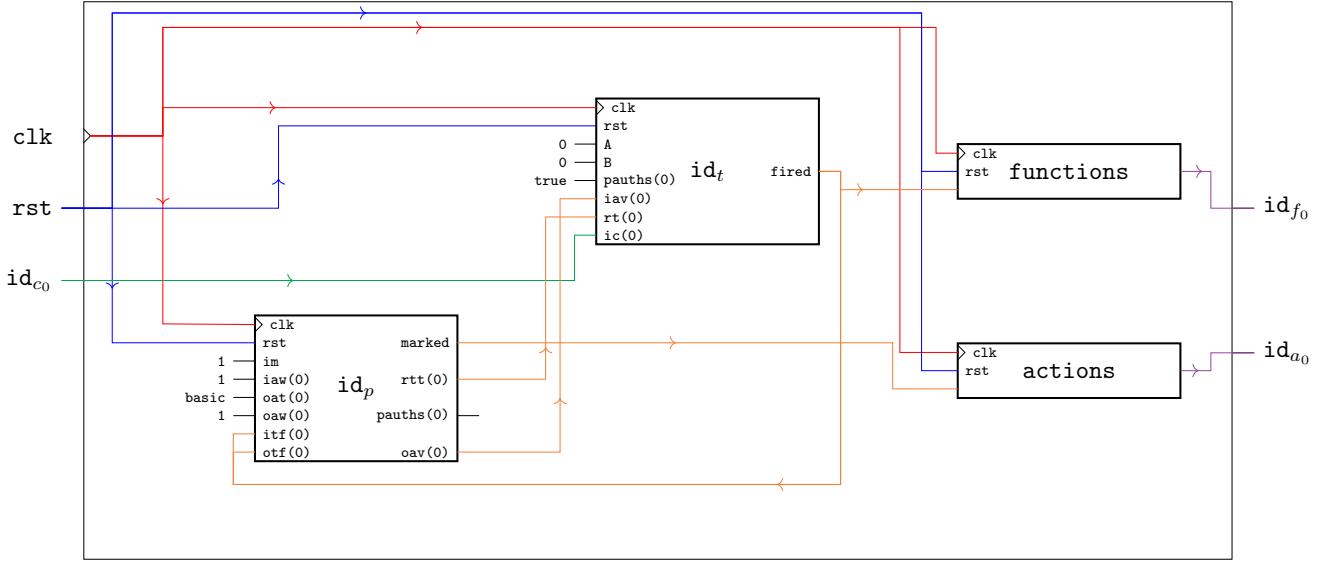
FIGURE 4.7: A graphic representation of the `tl` $\mathcal{H}$-VHDL top-level design presented in abstract syntax in Listing 4.6.

The rule of Figure 4.8 states that the full simulation of the `tl` design (presented in Listing 4.6) over 1 clock cycle yields the simulation trace ($\sigma_0 :: \sigma_1 :: \sigma_2$). The simulation over one clock cycle (the rightmost premise) yields a trace composed of two states: the state σ_1 at half the clock period, and the state σ_2 at the end of the first cycle. The full simulation happens in the context of the HILECOP's design store $\mathcal{D}_{\mathcal{H}}$, the elaborated design Δ, an empty dimensioning function and a simulation environment E_p. Here, $ports_{tl}$ is an alias for the list of ports of `tl`, $sigs_{tl}$ for the list of internal signals of `tl`, and cs_{tl} for the behavior of `tl`. In what follows, we will detail the premises of the FULLSIM rule.

$$
\cfrac{
\cfrac{\vdots}{\mathcal{D}_{\mathcal{H}}, \varnothing \vdash \texttt{design tl} \ldots \xrightarrow{elab} \Delta, \sigma_e}
\quad
\cfrac{\vdots}{\mathcal{D}_{\mathcal{H}}, \Delta, \sigma_e \vdash cs_{tl} \xrightarrow{init} \sigma_0}
\quad
\cfrac{\vdots}{\mathcal{D}_{\mathcal{H}}, E_p, \Delta, 1, \sigma_0 \vdash cs_{tl} \to (\sigma_1 :: \sigma_2)}
}{
\mathcal{D}_{\mathcal{H}}, \Delta, \varnothing, E_p, 1 \vdash \texttt{design tl tla } ports_{tl} \; sigs_{tl} \; cs_{tl} \xrightarrow{full} (\sigma_0 :: \sigma_1 :: \sigma_2)
} \; \text{FULLSIM}
$$

FIGURE 4.8: The FULLSIM rule applied to the `tl` design.

4.7.1 Elaboration of the `tl` design

The rule of Figure 4.9 states that the elaborated design Δ and the default design state σ_e are the result of the elaboration of the `tl` design. From left to right in the premises of the rule, the three first premises pertain to the elaboration of the declarative parts of the `tl` design, i.e. the generic constant declaration list, the port declaration list and the internal signal declaration list. The rightmost premise pertains to the elaboration of the behavior of the `tl` design.

$$\frac{\vdots}{\varnothing, \varnothing \vdash gens_{tl} \xrightarrow{egens} \Delta_0} \quad \frac{\vdots}{\Delta_0, \varnothing \vdash ports_{tl} \xrightarrow{eports} \Delta_1, \sigma_{e1}} \quad \frac{\vdots}{\Delta_1, \sigma_{e1} \vdash sigs_{tl} \xrightarrow{esigs} \Delta_2, \sigma_{e2}} \quad \frac{\vdots}{\mathcal{D}_{\mathcal{H}}, \Delta_2, \sigma_{e2} \vdash cs_{tl} \xrightarrow{ebeh} \Delta, \sigma_e}$$

$$\mathcal{D}_{\mathcal{H}}, \varnothing \vdash \texttt{design tl tla } gens_{tl} \ ports_{tl} \ sigs_{tl} \ cs_{tl} \xrightarrow{elab} \Delta, \sigma_e$$

FIGURE 4.9: The DESIGNELAB rule applied to the `tl` design.

Elaboration of the declarative parts

The elaboration of the declarative parts populates the *Gens*, *Ins*, *Outs* and *Sigs* sub-environments of the elaborated design Δ. Here is the content of the *Gens*, *Ins*, *Outs* and *Sigs* sub-environments of Δ_2, where Δ_2 is the partial elaborated design after the elaboration of the declarative parts of the `tl` design (passed as a parameter of third and the fourth premises of the rule in Figure 4.9).

– $Gens(\Delta_2) := \varnothing$

– $Ins(\Delta_2) := \{(id_{c0}, bool)\}$

– $Outs(\Delta_2) := \{(id_{f0}, bool), (id_{a0}, bool)\}$

– $Sigs(\Delta_2) := \{(id_{ft}, bool), (id_{av}, bool), (id_{rt}, bool), (id_m, bool)\}$

The top-level design generated by the HILECOP transformation all have an empty list of generic constants (see Chapter 5 for more details about the transformation). Also, all ports and internal signals are of the Boolean type. Thus, there are no range constraint or index constraint to solve here. The `boolean` type indication is simply transformed into the *bool* semantic type.

The elaboration of the declarative parts also gives a default value to the signals in the signal store of the default design state σ_{e2}, where σ_{e2} is the partial default design state at the end of the elaboration of the declarative parts of the `tl` design (passed as a parameter of the third and the fourth premises of the rule in Figure 4.9). Here is the content of the signal store $\mathcal{S}$ of σ_{e2}.

– $\mathcal{S}(\sigma_{e2}) := \{(id_{c0}, \bot), (id_{f0}, \bot), (id_{a0}, \bot), (id_{ft}, \bot), (id_{av}, \bot), (id_{rt}, \bot), (id_m, \bot)\}$

The default value associated to the *bool* type is $\bot$, thus, all signals of the `tl design` are initialized to $\bot$ in the signal store of σ_{e2}.

Elaboration of the behavioral part

The behavior of the `tl` design contains two component instantiation statements and two process statements. Each one of these statements will be elaborated in sequence. First, we present the elaboration of the `actions` process to illustrate the elaboration of a process statement; then, we present the elaboration of the place component instance id_p to illustrate the elaboration of a component instantiation statement.

Elaboration of a process statement

The rule of Figure 4.10 presents the elaboration of the `actions` process defined in the behavior of the `tl` design.

$$\vdots$$

$$\cfrac{\cfrac{}{\Delta_2, \varnothing \vdash \varnothing \xrightarrow{evars} \varnothing} \qquad \cfrac{\Delta_2, \sigma_{e2}, \varnothing \vdash \text{valid}_{ss}\left(\begin{array}{l}\texttt{rst}\,(id_{a0} \Leftarrow \texttt{false}) \\ (\texttt{falling}(id_{a0} \Leftarrow id_m \text{ or false}))\end{array}\right)}{} \text{WTR\scriptsize ST}}{\mathcal{D}_{\mathcal{H}}, \Delta_2, \sigma_{e2} \vdash \texttt{process}(\texttt{actions}, \texttt{clk}, \varnothing, \dots) \xrightarrow{ebeh} \Delta_2 \cup (\texttt{actions}, \varnothing), \sigma_{e2}} \text{P\scriptsize SELAB}$$

FIGURE 4.10: The elaboration of the `actions` process defined in the behavior of the `tl` design.

The `actions` process is elaborated in the context $\mathcal{D}_{\mathcal{H}}, \Delta_2, \sigma_{e2}$ where Δ_2 and σ_{e2} are the partially-built elaborated design and default design state at a certain point of the elaboration of the behavioral part of the `tl` design. The elaboration of a process statement associates the process identifier to a local variable environment in the Ps sub-environment of the being-built elaborated design. The local variable environment is built out of the variable declaration list of the process. Here, the `actions` process has an empty variable declaration list. Thus, the binding (`actions`,$\varnothing$) is added in the Ps sub-environment of Δ_2.

The elaboration of process statement also performs static type-checking on the process statement body leveraging the valid_{ss} relation. The rule of Figure 4.11 details the static type-checking of the statement body of the `actions` process (rightmost premise of the rule presented in Figure 4.10). To keep the example within a reasonable size, we discard the context of some rules with it is not relevant. We annotate the rule names to describe the side conditions associated to a derivation.

$$\cfrac{\cfrac{\cfrac{}{\texttt{false} \xrightarrow{e} \bot} \text{F\scriptsize ALSE} \quad \cfrac{}{\bot \in_c bool} \text{I\scriptsize SBOOL}}{\vdash \text{valid}_{ss}(id_{a0} \Leftarrow \texttt{false})} \text{WTS\scriptsize IG}^1 \quad \cfrac{\cfrac{\cfrac{\cfrac{}{\sigma_{e2} \vdash id_m \xrightarrow{e} \bot} \text{S\scriptsize IG}^2 \quad \cfrac{}{\texttt{false} \xrightarrow{e} \bot} \text{F\scriptsize ALSE}}{\sigma_{e2} \vdash id_m \text{ or false} \xrightarrow{e} \bot} \text{B\scriptsize OOLBINOP} \quad \cfrac{}{\bot \in_c bool} \text{I\scriptsize SBOOL}}{\sigma_{e2} \vdash \text{valid}_{ss}(id_{a0} \Leftarrow id_m \text{ or false})} \text{WTS\scriptsize IG}^1}{\sigma_{e2} \vdash \text{valid}_{ss}(\texttt{falling}(id_{a0} \Leftarrow id_m \text{ or false}))} \text{WTF\scriptsize ALLING}}{\Delta_2, \sigma_{e2}, \varnothing \vdash \text{valid}_{ss}\left(\begin{array}{l}\texttt{rst}\,(id_{a0} \Leftarrow \texttt{false}) \\ (\texttt{falling}(id_{a0} \Leftarrow id_m \text{ or false}))\end{array}\right)} \text{WTR\scriptsize ST}$$

(1) $\Delta_2(id_{a0}) = bool$ (2) $\sigma_{e2}(id_m) = \bot$

FIGURE 4.11: Static type-checking of the `actions` process statement body.

At the end of the elaboration of the `tl` design's behavior, the Ps sub-environment of Δ is as follows: $Ps(\Delta) := \{(\texttt{actions}, \varnothing), (\texttt{functions}, \varnothing)\}$

Elaboration of a component instantiation statement

The rule of Figure 4.12 presents the elaboration of the place component instance id_p belonging to the behavior of the tl design.

$$\frac{\varnothing \vdash g_p \xrightarrow{emapg} \mathcal{M} \quad \mathcal{D}_{\mathcal{H}}, \mathcal{M} \vdash \texttt{design place}\ldots \xrightarrow{elab} \Delta_p, \sigma_p \quad \Delta_2, \Delta_p, \sigma_{e2} \vdash \texttt{valid}_{ipm}(i_p) \quad \Delta_2, \Delta_p \vdash \texttt{valid}_{opm}(o_p)}{\mathcal{D}_{\mathcal{H}}, \Delta_2, \sigma_{e2} \vdash \texttt{comp}\ (id_p, \texttt{place}, g_p, i_p, o_p) \xrightarrow{ebeh} \Delta_2 \cup (id_p, \Delta_p), \sigma_{e2} \cup (id_p, \sigma_p)} \text{COMPELAB}^1$$

$$(1) \quad \begin{array}{l} id_p \notin \Delta_2 \\ id_p \notin \sigma_{e2} \\ \texttt{place} \in \mathcal{D}_{\mathcal{H}} \\ \mathcal{M} \subseteq Gens(\Delta_p) \end{array}$$

FIGURE 4.12: The elaboration of the id_p component instance defined in the behavior of the tl design.

The elaboration of a component instantiation statement is divided in three parts. First, a dimensioning function is built out of the generic map of the component instance. Figure 4.13 shows a part of the creation of the dimensioning functioning $\mathcal{M}$ from the generic map of the component instance id_p. Basically, the elaboration of a generic map is a transformation from a syntactic construct, i.e. the generic map, into a function, i.e. the dimensioning function $\mathcal{M}$. For each association of the generic map, the elaboration checks that the actual part of the association is a locally static expression (see Section 4.5.9).

$$\frac{\dfrac{\overline{SE_l(1)} \text{ LSENAT} \quad \overline{1 \xrightarrow{e} 1} \text{ NAT}}{\varnothing \vdash (\texttt{ian}, 1) \xrightarrow{emapg} \{(\texttt{ian}, 1)\}} \text{ASSOCGELAB}^1 \quad \dfrac{\vdots}{\{(\texttt{ian}, 1)\} \vdash (\texttt{oan}, 1), (\texttt{mm}, 1) \xrightarrow{emapg} \{(\texttt{ian}, 1), (\texttt{oan}, 1), (\texttt{mm}, 1)\}} \text{GMELAB}}{\varnothing \vdash (\texttt{ian}, 1), (\texttt{oan}, 1), (\texttt{mm}, 1) \xrightarrow{emapg} \{(\texttt{ian}, 1), (\texttt{oan}, 1), (\texttt{mm}, 1)\}} \text{GMELAB}$$

(1) $\texttt{ian} \notin \varnothing$

FIGURE 4.13: The elaboration of the generic map of the id_p component instance defined in the behavior of the tl design.

The second step of the elaboration of a component instance is to retrieve from the design store the design associated with the component instance, and to elaborate this design. Here, the design store is the HILECOP design store $\mathcal{D}_{\mathcal{H}}$, and the design associated with id_p is the place design. The dimensioning function $\mathcal{M}$ sets the value of the generic constants declared in the place design. The full code of place design is available in Appendix A. In Figures 4.14

and 4.15, we give the elaborated design Δ_p and the default design state σ_p resulting of the elaboration of the `place` design given the dimensioning function $\mathcal{M}$.

$\Delta_p := \{$

$\quad Gens := \quad \{(\texttt{ian}, (nat(0, \texttt{NATMAX}, 1)),$
$\qquad\qquad\qquad (\texttt{oan}, (nat(0, \texttt{NATMAX}), 1))$
$\qquad\qquad\qquad (\texttt{mm}, (nat(0, \texttt{NATMAX}), 1))\}$

$\quad Ins := \quad \{(\texttt{im}, nat(0, 1)),$
$\qquad\qquad\quad (\texttt{iaw}, array(nat(0, 255), 0, 0)),$
$\qquad\qquad\quad (\texttt{oat}, array(nat(0, 2), 0, 0)),$
$\qquad\qquad\quad (\texttt{oaw}, array(nat(0, 255), 0, 0)),$
$\qquad\qquad\quad (\texttt{itf}, array(bool, 0, 0)),$
$\qquad\qquad\quad (\texttt{otf}, array(bool, 0, 0))\},$

$\quad Outs := \quad \{(\texttt{oav}, array(bool, 0, 0)),$
$\qquad\qquad\qquad (\texttt{pauths}, array(bool, 0, 0)),$
$\qquad\qquad\qquad (\texttt{rtt}, array(bool, 0, 0))\}$

$\quad Sigs := \quad \{(\texttt{sits}, nat(0, 1)),$
$\qquad\qquad\qquad (\texttt{sm}, nat(0, 1)),$
$\qquad\qquad\qquad (\texttt{sots}, nat(0, 1))\},$

$\quad Ps := \quad \{(\texttt{input\_tokens\_sum}, \{(\texttt{v\_internal\_its}, (nat(0, 1), 0))\}),$
$\qquad\qquad\quad (\texttt{output\_tokens\_sum}, \{(\texttt{v\_internal\_ots}, (nat(0, 1), 0))\})\}$
$\qquad\qquad\quad (\texttt{priority\_evaluation}, \{(\texttt{v\_saved\_ots}, (nat(0, 1), 0))\})\}$

$\quad Comps := \varnothing$

$\}$

FIGURE 4.14: An elaborated version of the place design built with the dimensioning function deduced from the generic map of the component instance id_p.

In Δ_p, all the types associated with ports and internal signals of the `place` design have been *resolved*; i.e. the expressions qualifying the bounds of the range and index constraints in type indications have been evaluated. For example, `array(boolean, 0, input_arcs_number-1)` is the type indication associated with the `input_transitions_fired` input port (i.e. `itf`) defined in the port clause of the `place` design. The dimensioning function $\mathcal{M}$ sets the value of the `input_arcs_number` (i.e. `ian`) generic constant to 1. After the elaboration, the type indication `array(boolean, 0, input_arcs_number-1)` is thus transformed into the semantic type $array(bool, 0, 0)$. Thus, we have $\Delta_p(\texttt{itf}) = array(bool, 0, 0)$ in the resulting Δ_p.

Figure 4.15 shows the default design state σ_p of Δ_p.

$$\sigma_p := \{$$

$$\begin{aligned}
\mathcal{S} := \quad &\{(\texttt{im},(0)), (\texttt{iaw},(0)), (\texttt{oat},(0)), \\
&(\texttt{oaw},(0)), (\texttt{itf},(\bot)), (\texttt{otf},(\bot)), \\
&(\texttt{oav},(\bot)), (\texttt{pauths},(\bot)), (\texttt{rtt},(\bot)) \\
&(\texttt{sits},0), (\texttt{sm},0), (\texttt{sots},0)\}, \\[6pt]
\mathcal{C} := \; &\varnothing \\
\mathcal{E} := \; &\varnothing
\end{aligned}$$

$$\}$$

FIGURE 4.15: The default design state σ_p of the elaborated design Δ_p.

The component store of design state σ_p is empty as there are no component instantiation statements in the behavior of the place design. The same stands for the *Comps* subenvironment of Δ_p. Also, the set of events of a default design state is always empty.

The final step in the elaboration of a component instantiation statement is to check the wellformedness and the well-typedness of the input and output port maps. The $\texttt{valid}_{ipm}$ and $\texttt{valid}_{opm}$ relations, defined in Section 4.5.10, state the validity of the port maps. The rule of Figure 4.16 presents a part of the construction of the $valid_{opm}$ relation applied to the output port map of the place component instance id_p. Note that Δ_p is necessary to check the validity of the output port map of id_p, as it holds the correspondence between port identifiers and port types.

$$\dfrac{\dfrac{}{SE_l(0)} \text{\small LSENat} \qquad \dfrac{}{0 \xrightarrow{e} 0} \text{\small Nat} \qquad \dfrac{}{0 \in_c nat(0,0)} \text{\small IscNat}}{\Delta_2, \Delta_p, \varnothing, \varnothing \vdash (\texttt{oav}(0), id_{av}) \xrightarrow{list_{opm}} \{(\texttt{oav}, 0)\}, \{id_{av}\}} \; 1 \qquad \vdots \; \text{\small ListOPMCons}_B$$

$$\dfrac{\Delta_2, \Delta_p, \varnothing, \varnothing \vdash \begin{array}{l}(\texttt{oav}(0), id_{av}), (\texttt{pauths}, \texttt{open}) \\ (\texttt{rtt}(0), id_{rt}), (marked, id_m)\end{array} \xrightarrow{list_{opm}} \begin{array}{l}\{(\texttt{oav}, 0), \texttt{pauths}, \\ (\texttt{rtt}, 0), \texttt{marked}\}\end{array}, \{id_{av}, id_{rt}, id_m\}}{\quad} \text{\small ListOPMCons}_A$$

$$\dfrac{}{\Delta_2, \Delta_p \vdash \texttt{valid}_{opm}\left(\begin{array}{l}(\texttt{oav}(0), id_{av}), (\texttt{pauths}, \texttt{open}) \\ (\texttt{rtt}(0), id_{rt}), (marked, id_m)\end{array}\right)} \text{\small ValidOPM}$$

$$\vdots$$

$$\dfrac{}{\Delta_2, \Delta_p, \{(\texttt{oav}, 0)\}, \{id_{av}\} \vdash \begin{array}{l}(\texttt{pauths}, \texttt{open}), \\ (\texttt{rtt}(0), id_{rt}), \\ (marked, id_m)\end{array} \xrightarrow{list_{opm}} \begin{array}{l}\{(\texttt{oav}, 0), \\ \texttt{pauths}, \\ (\texttt{rtt}, 0), \\ \texttt{marked}\}\end{array}, \{id_{av}, id_{rt}, id_m\}} \text{\small ListOPMCons}_B$$

$$(1) \quad \begin{array}{l} \Delta_p(\texttt{oav}) = array(bool, 0, 0) \\ \Delta_2(id_{av}) = bool \\ \texttt{oav} \notin \varnothing \text{ and } (\texttt{oav}, 0) \notin \varnothing \\ id_{av} \notin \varnothing \end{array}$$

FIGURE 4.16: An example of validity checking performed on the output port map of the place component instance id_p. The bottom proof tree represents the top-right premise of the top proof tree.

At the end of the elaboration of the `tl` design's behavior, the *Comps* sub-environment of Δ is as follows: $Comps(\Delta) := \{(id_p, \Delta_p), (id_t, \Delta_t)\}$. Here, Δ_t represents the elaborated version of the `transition` design obtained from the elaboration of the transition component instance id_t.

Also, at the end of the elaboration, the component store of σ_e is as follows: $\mathcal{C}(\sigma_e) := \{(id_p, \sigma_p), (id_t, \sigma_t)\}$. Here, σ_t is the default design state of the transition component instance id_t.

4.7.2   Simulation of the `tl` design

Let us now present the rules pertaining to the simulation of the `tl` design, that is, pertaining to the execution of the `tl` design's behavior with respect to our formal simulation algorithm.

Initialization

The rule of Figure 4.17 presents the initialization phase in the proceeding of the simulation of the `tl` design. The initialization phase builds the initial state of the simulation. The first step

of the initialization, formalized by the *runinit* relation, runs the processes and the internal behavior of component instances exactly once (with the execution of the first part of reset blocks). Then, a stabilization phase follows.

$$\frac{\overset{\vdots}{\mathcal{D}_{\mathcal{H}},\Delta,\sigma_e \vdash cs_{tl} \xrightarrow{runinit} \sigma'} \qquad \overset{\vdots}{\mathcal{D}_{\mathcal{H}},\Delta,\sigma_e \vdash cs_{tl} \overset{\rightsquigarrow}{\longrightarrow} \sigma_0}}{\mathcal{D}_{\mathcal{H}},\Delta,\sigma_e \vdash cs_{tl} \xrightarrow{init} \sigma_0} \text{ INIT}$$

FIGURE 4.17: The initialization phase, first step of the simulation of the `tl` design.

The rule in Figure 4.18 presents the execution of the `tl` design's behavior during the *runinit* phase. The `tl` design's behavior is defined by the composition of concurrent statements. Here, the `actions` process is at the head of the behavior, whereas it is not the case in Listing 4.6. We formally proved, with the Coq proof assistant, that the $\parallel$ composition operator for concurrent statements is commutative and associative with respect to the *runinit* relation. In Figure, the `actions` process is executed and yields the state σ'_e. Then, the rest of the `tl` design's behavior is executed and yields the state σ''_e. Finally, the starting state σ_e and the two states σ'_e and σ''_e are merged into one by the merge function.

$$\frac{\overset{\vdots}{\mathcal{D}_{\mathcal{H}},\Delta,\sigma_e \vdash \texttt{process(actions,}\dots\texttt{)} \xrightarrow{runinit} \sigma'_e} \qquad \overset{\vdots}{\mathcal{D}_{\mathcal{H}},\Delta,\sigma_e \vdash cs'_{tl} \xrightarrow{runinit} \sigma''_e}}{\mathcal{D}_{\mathcal{H}},\Delta,\sigma_e \vdash \texttt{process(actions,}\dots\texttt{)} \parallel cs'_{tl} \xrightarrow{runinit} \texttt{merge}(\sigma_e,\sigma'_e,\sigma''_e)} \text{ COMPRUNINIT}^1$$

(1) $\mathcal{E}'_e \cap \mathcal{E}''_e$

FIGURE 4.18: The *runinit* phase applied to the concurrent statements composing the behavior of the `tl` design.

In what follows, we detail the execution of a process statement and of a component instantiation statement during the first part of the initialization, i.e. the *runinit* phase.

Execution of a process statement with the *runinit* relation

The rule in Figure 4.19 shows the execution of the `actions` process during the *runinit* phase. The first part of the reset block defining the statement body of the `actions` process is executed. This first part assigns the expression `false` to signal id_{a0}.

$$\frac{\overline{\vdash \texttt{false} \xrightarrow{e} \bot}\ \text{FALSE} \qquad \overline{\bot \in_c bool}\ \text{IsBool}}{\mathcal{D}_{\mathcal{H}}, \Delta, \sigma_e, \sigma_e \vdash id_{a0} \Leftarrow \texttt{false} \xrightarrow{ss_i} \sigma_e', \emptyset}\ \text{SigAssign}^2$$

$$\frac{}{\mathcal{D}_{\mathcal{H}}, \Delta, \sigma_e \vdash \texttt{rst}(id_{a0} \Leftarrow \texttt{false})(\dots) \xrightarrow{ss_i} \sigma_e', \emptyset}\ \text{RstExec}$$

$$\frac{}{\mathcal{D}_{\mathcal{H}}, \Delta, \sigma_e \vdash \texttt{process}(\texttt{actions}, \dots) \xrightarrow{runinit} \sigma_e'}\ \text{PsRunInit}^1$$

$$(1)\ \Delta(\texttt{actions}) = \emptyset \qquad (2)\ \begin{aligned} \Delta(id_{a0}) &= bool \\ \sigma_e' &= <\mathcal{S}_e', \mathcal{C}_e', \mathcal{E}_e'> \\ \mathcal{S}_e' &= \mathcal{S}_e(id_{a0}) \leftarrow \bot \\ \mathcal{E}_e' &= \mathcal{E}_e \cup (\mathcal{S}_e \overset{\neq}{\cap} \mathcal{S}_e') \end{aligned}$$

FIGURE 4.19: The *runinit* phase applied to the concurrent statements composing the behavior of the `tl` design.

In the side conditions of the SigAssign rule, a new event set $\mathcal{E}_e'$ is computed based on the event set $\mathcal{E}_e$ joined to the expression $\mathcal{S}_e \overset{\neq}{\cap} \mathcal{S}_e'$. This expression returns the set of signals with a different value between signal store $\mathcal{S}_e$ and signal store $\mathcal{S}_e'$. The only signal that possibly has a different value from $\mathcal{S}_e$ to $\mathcal{S}_e'$ is the assigned signal id_{a0}. Thus, this expression is a shorthand to test if the value of signal id_{a0} has changed after the execution of the signal assignment statement. If it is the case, then the event set receives the signal identifier id_{a0}; id_{a0} is then an eventful signal. In the present case, the value of signal id_{a0} was $\bot$ at state σ_e and is still $\bot$ after the execution of the signal assignment statement. Therefore, no event is registered on signal id_{a0}. When states σ_e, σ_e' and σ_e'' will be merged (cf. Figure 4.18), if id_{a0} is part of the event set of state σ_e'', then, the merged state will return the value associated to id_{a0} in state σ_e''. We would have $\texttt{merge}(\sigma_e, \sigma_e', \sigma_e'')(id_{a0}) = \sigma_e''(id_{a0})$. However, signal id_{a0} would be a potentially multiply-driven signal because both the `actions` process and the concurrent statement cs_{tl}' (cf. Figure 4.18) both assign the signal value.

Execution of a component instantiation statement with the *runinit* relation

The rule of Figure 4.20 presents the execution of the `place` component instance id_p during the *runinit* phase. The execution of a component instantiation statement is pretty much the same in all the phases of the simulation algorithm. The difference lies in the choice of the relation used to execute of the internal behavior of the component instance. During the initialization phase, it is the *runinit* relation that executes the internal behavior of component instances; during the rising edge phase, it is the $\uparrow$ relation that executes the internal behaviors, etc.

$$\cfrac{\cfrac{\vdots}{\Delta,\Delta_p,\sigma_e,\sigma_p \vdash i_p \xrightarrow{\;mapip\;} \sigma'_p} \quad \cfrac{\vdots}{\mathcal{D_H},\Delta_p,\sigma'_p \vdash cs_p \xrightarrow{\;runinit\;} \sigma''_p} \quad \cfrac{\vdots}{\Delta,\Delta_p,\sigma_e,\sigma''_p \vdash o_p \xrightarrow{\;mapop\;} \sigma'_e}}{\mathcal{D_H},\Delta,\sigma_e \vdash \text{comp}(id_p,\text{place},g_p,i_p,o_p) \xrightarrow{\;runinit\;} \sigma''_e} \; \text{\scriptsize COMPRUNINIT}^1$$

$$(1)\;\begin{aligned} \sigma'_e &= <\mathcal{S}'_e,\mathcal{C}'_e,\mathcal{E}'_e> \\ \sigma''_e &= <\mathcal{S}'_e,\mathcal{C}''_e,\mathcal{E}'_e \cup (\mathcal{C}_e \overset{\neq}{\cap} \mathcal{C}''_e)> \\ \mathcal{C}''_e &= \mathcal{C}'_e(id_p) \leftarrow \sigma''_p \end{aligned}$$

FIGURE 4.20: The execution of the `place` component instance id_p during the *runinit* phase (first part of the initialization).

The execution of a component instantiation statement is decomposed in four parts. First, the input ports of the component instance receive new values through the evaluation of the input port map. Second, the internal behavior of the component instance is executed. Thirdly, the evaluation of the output port map propagates the values coming from the output interface of the component to the signals of the embedding design. Finally, the component instance is assigned to its new internal state in the component store of the embedding design; here, σ''_p is assigned to id_p in the component store $\mathcal{C}''_e$. Moreover, if the new internal state of the component instance is different from its older internal state, then the component instance identifier is added to the event set of the embedding design. Here, the expression $\mathcal{C}_e \overset{\neq}{\cap} \mathcal{C}''_e$ performs the state comparison; we have:

$$\begin{aligned} \mathcal{C}_e \overset{\neq}{\cap} \mathcal{C}''_e \;&= \mathcal{C}_e \overset{\neq}{\cap} (\mathcal{C}'_e \leftarrow \sigma''_p) \\ &= \mathcal{C}_e \overset{\neq}{\cap} (\mathcal{C}_e \leftarrow \sigma''_p) \\ &= \begin{cases} \{id_p\} \text{ if } \sigma_p \neq \sigma''_p \\ \varnothing \text{ otherwise} \end{cases} \end{aligned}$$

In the second line, we have $\mathcal{C}_e = \mathcal{C}'_e$ because the evaluation of the output port map (performed by the *mapop* relation) does not affect the component store.

The rule of Figure 4.21 gives a part of the evaluation of the input port map of id_p.

$$\dfrac{\dfrac{}{\Delta, \sigma_e \vdash 1 \xrightarrow{e} 1} \text{N}\textsc{at} \quad \dfrac{}{1 \in_c nat(0,1)} \text{I}\textsc{sCN}\textsc{at}}{\Delta, \Delta_p, \sigma_e, \sigma_p \vdash (\mathtt{im}, 1) \xrightarrow{mapip} \sigma'_{p0}} \text{M}\textsc{apip}\textsc{Simple}^1 \qquad \dfrac{\vdots}{\Delta, \Delta_p, \sigma_e, \sigma'_{p0} \vdash \begin{array}{l}(\mathtt{iaw}(0),1) \\ (\mathtt{oat}(0),0),\, (\mathtt{oaw}(0),1), \\ (\mathtt{itf}(0), id_{ft}),\, (\mathtt{otf}(0), id_{ft})\end{array} \xrightarrow{mapip} \sigma'_p}$$

$$\dfrac{}{\Delta, \Delta_p, \sigma_e, \sigma_p \vdash \begin{array}{l}(\mathtt{im}, 1),\, (\mathtt{iaw}(0),1) \\ (\mathtt{oat}(0),0),\, (\mathtt{oaw}(0),1), \\ (\mathtt{itf}(0), id_{ft}),\, (\mathtt{otf}(0), id_{ft})\end{array} \xrightarrow{mapip} \sigma'_p} \text{M}\textsc{apip}\textsc{Comp}$$

$$(1) \quad \begin{aligned} & \Delta_p(\mathtt{im}) = nat(0,1) \\ & \sigma_p = {<}\mathcal{S}, \mathcal{C}, \mathcal{E}{>} \\ & \mathcal{S}' = \mathcal{S}(\mathtt{im}) \leftarrow 1 \\ & \sigma'_{p0} = {<}\mathcal{S}', \mathcal{C}, \mathcal{E}{>} \end{aligned}$$

FIGURE 4.21: The evaluation of the input port map of the place component instance id_p.

The evaluation of the input port map of id_p changes the value of the `initial_marking` input port (i.e. `im`). We have $\sigma_p(\mathtt{im}) = 0$ and $\sigma'_p(\mathtt{im}) = 1$. As the value of one of its input port has changed, the `place` component instance id_p will be registered as an eventful component instance.

The rule of Figure 4.22 gives a part of the evaluation of the output port map of id_p.

$$\dfrac{\dfrac{\dfrac{\vdash 0 \xrightarrow{e} 0 \quad \overline{0 \in_c nat(0,0)}}{\Delta_p, \sigma''_p \vdash \mathtt{oav}(0) \xrightarrow{e_0} \top} \text{I}\textsc{dxS}\textsc{ig}^2 \quad \dfrac{}{\top \in_c bool} \text{I}\textsc{sB}\textsc{ool}}{\Delta, \Delta_p, \sigma_e, \sigma''_p \vdash (\mathtt{oav}(0), id_{av}) \xrightarrow{mapop} \sigma'_{e0}} 1 \qquad \dfrac{\vdots}{\Delta, \Delta_p, \sigma'_{e0}, \sigma''_p \vdash \begin{array}{l}(\mathtt{pauths}, \mathtt{open}) \\ (\mathtt{rtt}(0), id_{rt}),\, (\mathtt{marked}, id_m)\end{array} \xrightarrow{mapop} \sigma'_e}}{\Delta, \Delta_p, \sigma_e, \sigma''_p \vdash \begin{array}{l}(\mathtt{oav}(0), id_{av}),\, (\mathtt{pauths}, \mathtt{open}) \\ (\mathtt{rtt}(0), id_{rt}),\, (\mathtt{marked}, id_m)\end{array} \xrightarrow{mapop} \sigma'_e} \text{M}\textsc{apop}\textsc{Comp}$$

$$(1) \quad \begin{aligned} & \Delta(id_{av}) = bool \\ & \sigma_e = {<}\mathcal{S}, \mathcal{C}, \mathcal{E}{>} \\ & \mathcal{S}' = \mathcal{S}(id_{av}) \leftarrow \top \\ & \mathcal{E}' = \mathcal{E} \cup (\mathcal{S} \overset{\neq}{\cap} \mathcal{S}') \\ & \sigma'_{e0} = {<}\mathcal{S}', \mathcal{C}, \mathcal{E}'{>} \end{aligned} \qquad (2) \quad \begin{aligned} & \Delta_p(\mathtt{oav}) = array(bool, 0, 0) \\ & \sigma''_p(\mathtt{oav}) = (\top) \\ & \mathtt{get\_at}(0, (\top)) = \top \end{aligned}$$

FIGURE 4.22: The evaluation of the output port map of the place component instance id_p.

Stabilization

A stabilization phase happens after the *runinit* phase during the initialization phase, but also after the rising edge phase and the falling edge phase in the course of a simulation cycle. The stabilization phase executes the combinational parts of the design's behavior. The `tl` design holds no combinational processes in its behavior. The `actions` and `functions` processes are both synchronous. To illustrate the execution of a combinational process during a stabilization phase, let us consider the `fired_evaluation` process defined in the behavior of the `transition` design. The `fired_evaluation` process will be executed with the internal behavior of the `transition` component instance id_t during the stabilization phase. The rule of Figure 4.23 presents the execution of the internal behavior of the `transition` component instance id_t. As shown, the internal behavior cs_t is executed three times before reaching a stable state. Here, the number of execution before stabilization is arbitrary. In Figure 4.23, σ_{t0} corresponds to the state of id_t after the *runinit* phase and after the evaluation of its input port map. Remember that the evaluation of the input port map of a component instance always precedes the execution of the internal behavior of the component. Since σ_{t0} and σ_{t1} are not stable states, it means that their event set is not empty. Thus, we have $\mathcal{E}(\sigma_{t0}) \neq \varnothing$ and $\mathcal{E}(\sigma_{t1}) \neq \varnothing$. On the contrary, σ_{t2} is a stable state, and thus, $\mathcal{E}(\sigma_{t2}) = \varnothing$.

$$
\cfrac{
  \cfrac{\vdots}{\mathcal{D}_{\mathcal{H}},\Delta_t,\sigma_{t0} \vdash cs_t \xrightarrow{comb} \sigma_{t1}}
  \quad
  \cfrac{
    \cfrac{\vdots}{\mathcal{D}_{\mathcal{H}},\Delta_t,\sigma_{t1} \vdash cs_t \xrightarrow{comb} \sigma_{t2}}
    \quad
    \cfrac{
      \cfrac{\vdots}{\mathcal{D}_{\mathcal{H}},\Delta_t,\sigma_{t2} \vdash cs_t \xrightarrow{comb} \sigma_{t2}}
      \quad
      \mathcal{D}_{\mathcal{H}},\Delta_t,\sigma_{t2} \vdash cs_t \xrightarrow{\rightsquigarrow} \sigma_{t2}
    }{\mathcal{D}_{\mathcal{H}},\Delta_t,\sigma_{t1} \vdash cs_t \xrightarrow{\rightsquigarrow} \sigma_{t2}} \text{ StabilizeLoop}
  }{\mathcal{D}_{\mathcal{H}},\Delta_t,\sigma_{t0} \vdash cs_t \xrightarrow{\rightsquigarrow} \sigma_{t2}} \text{ StabilizeLoop}
$$

FIGURE 4.23: Three rounds of execution of the combinational parts of the `transition` component instance id_t during a stabilization phase.

Now, let us illustrate the execution of the `fired_evaluation` process, which is a part of the `transition` design behavior cs_t, happening during the stabilization phase. The rule of Figure 4.24 details the execution of the body of process `fired_evaluation` performed by the *comb* relation.

$$\dfrac{\overline{\Delta_t,\sigma_{t0} \vdash \texttt{sfa} \xrightarrow{e} \bot}\ \text{\scriptsize SIG} \qquad \overline{\Delta_t,\sigma_{t0} \vdash \texttt{spc} \xrightarrow{e} \top}\ \text{\scriptsize SIG}}{\Delta_t,\sigma_{t0} \vdash \texttt{sfa and spc} \xrightarrow{e} \bot}\ \text{\scriptsize BOOLBINOP}$$

$$\dfrac{\Delta_t,\sigma_{t0} \vdash \texttt{sfa and spc} \xrightarrow{e} \bot \qquad \overline{\bot \in_c bool}\ \text{\scriptsize ISBOOL}}{\mathcal{D}_{\mathcal{H}},\Delta_t,\sigma_{t0},NoEv(\sigma_{t0}),\varnothing \vdash \texttt{fired} \Leftarrow \texttt{sfa and spc} \xrightarrow{ss_c} \sigma'_{t0},\varnothing}\ \text{\scriptsize SIGASSIGN}^2$$

$$\dfrac{\mathcal{D}_{\mathcal{H}},\Delta_t,\sigma_{t0},NoEv(\sigma_{t0}),\varnothing \vdash \texttt{fired} \Leftarrow \texttt{sfa and spc} \xrightarrow{ss_c} \sigma'_{t0},\varnothing}{\mathcal{D}_{\mathcal{H}},\Delta_t,\sigma_{t0} \vdash \begin{array}{l}\texttt{process(fired\_evaluation, \{sfa, spc\}, } \varnothing, \xrightarrow{\rightsquigarrow} \sigma'_{t0}\\ \texttt{fired} \Leftarrow \texttt{sfa and spc)}\end{array}}\ \text{\scriptsize PSCOMB}^1$$

$$(1)\ \Delta_t(\texttt{fired\_evaluation}) = \varnothing \qquad (2)\ \begin{array}{l}\Delta_t(\texttt{fired}) = bool\\ NoEv(\sigma_{t0}) = <\mathcal{S},\mathcal{C},\varnothing>\\ \mathcal{S}' = \mathcal{S}(\texttt{fired}) \leftarrow \bot\\ \sigma'_{t0} = <\mathcal{S}',\mathcal{C},\mathcal{S} \stackrel{\neq}{\cap} \mathcal{S}'>\end{array}$$

FIGURE 4.24: The execution of the `fired_evaluation` process during a stabilization phase. The `fired_evaluation` process is defined in the `transition` design's behavior.

Let us consider that the value of the `fired` signal was $\top$ at state σ_{t0}, i.e. $\sigma_{t0}(\texttt{fired}) = \top$. Then, since $\sigma'_{t0}(\texttt{fired}) = \bot$, we have $\mathcal{S} \stackrel{\neq}{\cap} \mathcal{S}' = \{\texttt{fired}\}$. When state σ'_{t0} will be merged with the other states generated by the concurrent execution of processes defining the `transition` design's behavior, the resulting merged state will have a non-empty set of events. Thus, another round of execution will be triggered. A stable state has been reached when the execution of the combinational parts of the behavior does not generate any event anymore.

Simulation cycle and clock phases

We describe here the execution of the `tl` design over one clock cycle. After the initialization phase, the design under simulation will execute τ simulation cycles, where τ is a natural numbers passed as a parameter. In the rule of Figure 4.25, τ equals 1. Thus, the behavior of the `tl` design is executed during one clock cycle and then the simulation ends. In Figure 4.25, σ_0 is the initial simulation state, i.e. the one at the end of the initialization phase. One simulation cycle yields two states σ_1 and σ_2, where σ_1 is the state after the rising edge and the stabilization phases (i.e. in the middle of the clock cycle), and σ_2 is the state after the falling edge and the stabilization phases (i.e. at the end of the first cycle). The resulting simulation trace is only composed of states σ_1 and σ_2.

$$\frac{\vdots}{\mathcal{D}_{\mathcal{H}}, E_p, \Delta, 1, \sigma_0 \vdash cs_{tl} \xrightarrow{\uparrow, \downarrow} \sigma_1, \sigma_2} \text{SimCyc} \qquad \frac{}{\mathcal{D}_{\mathcal{H}}, E_p, \Delta, 0, \sigma_2 \vdash cs_{tl} \rightarrow []} \text{SimEnd}$$

$$\frac{}{\mathcal{D}_{\mathcal{H}}, E_p, \Delta, 1, \sigma_0 \vdash cs_{tl} \rightarrow (\sigma_1 :: \sigma_2 :: [])} \text{SimLoop}^1$$

(1) $1 > 0$

FIGURE 4.25: The execution of the `tl` design's behavior during one clock cycle.

The rule of Figure 4.26 zooms in the first simulation cycle. The state σ_1 is produced by a rising edge phase followed by a stabilization phase. The state σ_2 is produced by a falling edge phase followed by a stabilization phase. The value of the primary input ports of the `tl` design are updated before the rising edge event. States σ_{0i} is the new state obtained after the update of the primary input port values. The update corresponds to the capture of values yielded by the given simulation environment E_p. The `tl` design has only one primary input port, i.e. the input port id_{c0}. The value of id_{c0} at state σ_{0i} is equal to the value yielded by the environment E_p during the first clock cycle. Thus, we have $\sigma_{0i}(id_{c0}) = E_p(1)(id_{c0})$.

$$\frac{\vdots}{\mathcal{D}_{\mathcal{H}}, \Delta, \sigma_{0i} \vdash cs_{tl} \xrightarrow{\uparrow} \sigma_{\uparrow}} \quad \frac{\vdots}{\mathcal{D}_{\mathcal{H}}, \Delta, \sigma_{\uparrow} \vdash cs_{tl} \xrightarrow{\rightsquigarrow} \sigma_1} \quad \frac{\vdots}{\mathcal{D}_{\mathcal{H}}, \Delta, \sigma_1 \vdash cs_{tl} \xrightarrow{\downarrow} \sigma_{\downarrow}} \quad \frac{\vdots}{\mathcal{D}_{\mathcal{H}}, \Delta, \sigma_{\downarrow} \vdash \xrightarrow{\rightsquigarrow} \sigma_2}$$

$$\frac{}{\mathcal{D}_{\mathcal{H}}, E_p, \Delta, 1, \sigma_0 \vdash \xrightarrow{\uparrow, \downarrow} \sigma_1, \sigma_2} \text{SimCyc}^1$$

(1) $\texttt{Inject}(\sigma_0, E_p, 1, \sigma_{0i})$

FIGURE 4.26: Details of the execution of the `tl` design's behavior during the first clock cycle.

The rule of Figure 4.27 describes the execution of the `functions` process, defined in the `tl` design's behavior, during the rising edge phase of the first simulation cycle. During the rising edge phase, rising blocks are executed. Thus, the $\uparrow$ relation triggers the execution of the rising block defined in the body of the `functions` process.

$$\frac{}{\Delta, \sigma_{0i} \vdash id_{ft} \xrightarrow{e} \top} \text{ SIG}^3 \qquad \frac{}{\texttt{false} \xrightarrow{e} \bot} \text{ FALSE}$$

$$\frac{}{\Delta, \sigma_{0i} \vdash id_{ft} \text{ or } \texttt{false} \xrightarrow{e} \top} \text{ BOOLBINOP} \qquad \frac{}{\top \in_c bool} \text{ ISBOOL}$$

$$\frac{}{\Delta, \sigma_{0i}, \sigma_{0i}, \varnothing \vdash id_{f0} \Leftarrow id_{ft} \text{ or } \texttt{false} \xrightarrow{ss_\uparrow} \sigma'_{0i}, \varnothing} \text{ SIGASSIGN}^2$$

$$\frac{}{\Delta, \sigma_{0i}, \sigma_{0i}, \varnothing \vdash \texttt{rising}\,(id_{f0} \Leftarrow id_{ft} \text{ or } \texttt{false}) \xrightarrow{ss_\uparrow} \sigma'_{0i}, \varnothing} \text{ RISINGEDGEEXEC}$$

$$\frac{}{\Delta, \sigma_{0i}, \sigma_{0i}, \varnothing \vdash (\texttt{rst}\,(id_{f0} \Leftarrow \texttt{false})(\texttt{rising}\,(id_{f0} \Leftarrow id_{ft} \text{ or } \texttt{false}))) \xrightarrow{ss_\uparrow} \sigma'_{0i}, \varnothing} \text{ RSTDEFAULT}$$

$$\frac{}{\mathcal{D}_{\mathcal{H}}, \Delta, \sigma_{0i} \vdash \begin{matrix}\texttt{process}\,(\texttt{functions}, \{\texttt{clk}\}, \varnothing, \\ (\texttt{rst}\,(id_{f0} \Leftarrow \texttt{false})(\texttt{rising}\,(id_{f0} \Leftarrow id_{ft} \text{ or } \texttt{false})))) \end{matrix} \xrightarrow{\uparrow} \sigma'_e} \text{ PSRECLK}^1$$

$$(1) \begin{array}{l} \texttt{clk} \in \{\texttt{clk}\} \\ \Delta(\texttt{functions}) = \varnothing \end{array} \qquad (2) \begin{array}{l} \Delta(id_{f0}) = bool \\ \sigma_{0i} = <\mathcal{S}, \mathcal{C}, \mathcal{E}> \\ \mathcal{S}' = \mathcal{S}(id_{f0}) \leftarrow \top \\ \mathcal{E}' = \mathcal{E} \cup (\mathcal{S} \stackrel{\neq}{\cap} \mathcal{S}') \\ \sigma'_{0i} = <\mathcal{S}', \mathcal{C}, \mathcal{E}'> \end{array} \qquad (3) \; \sigma_{0i}(id_{ft}) = \top$$

FIGURE 4.27: The execution of the `functions` process during a rising edge phase. The `functions` process is a part of the `tl` design's behavior.

4.8 Implementation of the H-VHDL syntax and semantics

This section presents the implementation of the H-VHDL abstract syntax, and also of the elaboration and the simulation semantics of H-VHDL designs with the Coq proof assistant. The full code is available under the hvhdl folder of the following repository: https://github.com/viampietro/ver-hilecop.

4.8.1 Implementation of the H-VHDL abstract syntax, elaborated design and design state

H-VHDL abstract syntax

The implementation of the H-VHDL abstract syntax is naturally done leveraging the Inductive construct of the Coq proof assistant. The result is strictly similar to the formal definition of the abstract syntax given in Section 4.3. The reader can refer to the AbstractSyntax.v under the hvhdl folder for the details of the implementation.

Elaborated design

Listing 4.7 presents the implementation of the elaborated design structure (cf. Definition 31). Two definitions are involved in the implementation of the elaborated design structure. The first one defines the SemanticObject inductive type. Each constructor of this type corresponds

to a sub-environment of the elaborated design. For instance, the `Generic` constructor corresponds to the couple (*type* × *value*) associated with a generic constant identifier in the *Gens* sub-environment of Definition 31. The `Process` constructor corresponds to the local variable environment associated with the process identifiers in the *Ps* sub-environment. A local variable environment is implemented by the `LEnv` type. The `LEnv` type is a map between identifiers and couples (*type* × *value*). Identifiers are implemented by the `ident` type, an alias of the `nat` type. The `type` and `value` types are the implementation of the semantic *type* and *value* presented in Table 4.2. The `ElDesign` type implements the elaborated design structure. It is an alias to the `IdMap SemanticObject` type. The `IdMap` is the type of maps from identifiers (i.e. belonging to the `ident` type) to instances of the type passed as an input. Here, the input is the `SemanticObject` type. Thus, an elaborated design is implemented as a map between identifiers and terms of the `SemanticObject` type. We leverage the `FMaps` module defined in the Coq standard library to implement the `IdMap` type. The `IdMap` type ensures that an identifier is only mapped once. Thus, the implementation of the elaborated design structure verifies that there are no intersection between the domains of sub-environments. For instance, a generic constant identifier can not be an input port identifier, and, as it is implemented, an identifier `id` can not be mapped to a `Generic` object and to an `Input` object in the same instance of `ElDesign`.

```
1  Inductive SemanticObject : Type :=
2  | Generic (t : type) (v : value)
3  | Input (t : type)
4  | Output (t : type)
5  | Declared (t : type)
6  | Process (lenv : LEnv)
7  | Component (Δ_c : IdMap SemanticObject).
8
9  Definition ElDesign := IdMap SemanticObject.
```

LISTING 4.7: The implementation of the elaborated design structure with the Coq proof assistant.

Design state

Listing 4.8 gives the implementation of the design state structure through the definition of the `DState` inductive type. The constructor of the `DState` type defines three fields: `sigstore`, implementing the signal store $\mathcal{S}$ of the design state, `compstore`, implementing the component store C, and `events`, implementing the set of events $\mathcal{E}$ of the design state. The `sigstore` field is a map from identifiers to values. The `compstore` field is a map from identifiers to design states, justifying the inductive definition of the `DState` type. The `events` field is an instance of the `IdSet` type. The `IdSet` is the type of sets of identifiers (i.e. sets of natural numbers). The `IdSet` type is defined leveraging the `MSets` module of the Coq standard library.

```
1  Inductive DState : Type := MkDState {
2    sigstore :  IdMap value;
3    compstore : IdMap DState;
4    events   :  IdSet;
```

```
5  }.
```

LISTING 4.8: The implementation of the design state structure with the Coq proof assistant.

4.8.2 Implementation of the elaboration phase

The design elaboration relation, as presented in Section 4.5.1, is implemented in Coq by the `edesign` relation. Listing 4.9 presents the definition of the `edesign` relation as an inductive type. As usual, a n-ary relation is implemented in Coq by a type defined with *n* parameters and projecting to the `Prop` type. The `edesign` relation has five parameters. The first parameter is the design store $\mathcal{D}$ of type `IdMap design`, i.e. a map from identifiers to $\mathcal{H}$-VHDL designs as defined by the abstract syntax. The second parameter is the dimensioning function $\mathcal{M}$ of type `IdMap value`, i.e. a map from identifiers to values. The third parameter is the design being elaborated, of type `design`. The fifth and sixth parameters are the elaborated design (of type `ElDesign` and the default design state (of type `DState`) resulting from the elaboration. In Listing 4.9, the `EDesign` constructor implements the DESIGNELAB rule presented in Section 4.5.1. From Line 7 to Line 10, the constructor defines the premises of Rule DESIGNELAB. The empty elaborated design structure, denoted $\Delta_{\varnothing}$, is implemented by the `EmptyElDesign` definition, and the empty design state structure, denoted by $\sigma_{\varnothing}$, is implemented by the `EmptyDState` definition. Line 13 implements the conclusion of Rule DESIGNELAB.

```
1   Inductive edesign (𝒟 : IdMap design) : IdMap value → design → ElDesign → DState → Prop :=
2   | EDesign :
3       forall ℳ id_e id_a gens ports sigs behavior
4               Δ Δ′ Δ″ Δ‴ σ σ′ σ″,
5
6       (* Premises *)
7       egens EmptyElDesign ℳ gens Δ →
8       eports Δ EmptyDState ports Δ′ σ →
9       edecls Δ′ σ sigs Δ″ σ′ →
10      ebeh 𝒟 Δ″ σ′ behavior Δ‴ σ″ →
11
12      (* Conclusion *)
13      edesign 𝒟 ℳ (design_ id_e id_a gens ports sigs behavior) Δ‴ σ″
14
15  with ebeh (𝒟 : IdMap design) : ElDesign → DState → cs → ElDesign → DState → Prop :=
16  ...
```

LISTING 4.9: The implementation of the design elaboration relation with the Coq proof assistant.

The `edesign` relation requires a mutually recursive definition with the `ebeh` relation. The mutually recursive definition is performed leveraging the `with` clause at the end of Listing 4.9. The `ebeh` relation needs the `edesign` relation to elaborate the component instances found in the behavior of a design. Listing 4.10 gives the details of the `with` clause defining the `ebeh` relation.

At Line 2, the `EBehPs` constructor implements the Rule PSELAB defining the elaboration of a process statement (cf. Section 4.5.6). Lines 6 and 7 implement the premises of the rule; the `evars` relation implements the elaboration of the local variable declaration list of the process; the `validss` relation implements the relation that type-checks the statement body of the process. Lines 10 to 14 implement the side conditions of the rule. The term $\sim$`NatMap.In` id_p Δ implements the side condition $id_p \notin \Delta$. The `NatMap.In id m` relation states that a given identifier `id` is a key of the `m` map. At Line 13, the `NatSet.In` id_s `sl` term states that id_s belongs to the identifier set `sl`. At Line 14, the term `MapsTo` id_s `(Input t)` Δ states that the identifier id_s is mapped to `Input t` in the elaborated design Δ, i.e. $Ins(\Delta)(id_s) = $ `t`. More generally, `MapsTo` is a ternary relation stating that a given key `k` of type `nat`, is mapped to a value `v` of a type `A`, in a given map `m`, i.e. `Mapsto k v m`. Line 17 implements the conclusion of Rule PSELAB. The `NatMap.add` function binds the process identifier id_p to the term `Process` Λ in the elaborated design Δ, i.e. $\Delta \cup (id_p, \Lambda)$.

At Line 19, the `EBehComp` constructor implements the Rule COMPELAB (cf. Section 4.5.6). This rule describes the elaboration of a component instantiation statement. Lines 24 to 27 implement the premises of the rule. Line 25 appeals to the `edesign` relation to elaborate the `cdesign` design associated with the component instance id_c; thence, the mutually recursive definition with the `ebeh` relation. As it is stated at Line 32, the `cdesign` design is associated to identifier id_e, i.e. the entity identifier of component instance id_c, in the design store $\mathcal{D}$. Lines 30 to 33 implement the side conditions of the rule. Line 30 checks that the identifier id_c is not already bound to a semantic object in the elaborated design Δ. Line 31 checks that the identifier id_c is not already bound in the component store of σ. Line 33 checks that all identifiers defined in the domain of map $\mathcal{M}$, i.e. the dimensioning function, are bound to generic constants in the elaborated design Δ_c (i.e. $\mathcal{M} \subseteq Gens(\Delta_c)$). Lines 36 to 38 implement the conclusion of Rule COMPELAB. At Line 39, the `cstore_add` function binds id_c to design state σ_c in the component store of state σ and returns the resulting state.

At Line 41, the `EBehNull` constructor implements Rule CSNULLELAB. At Line 43, the `EBeh-Par` constructor implements Rule CSPARELAB.

```
1  with ebeh (𝒟 : IdMap design) : ElDesign → DState → cs → ElDesign → DState → Prop :=
2  | EBehPs :
3      forall id_p sl vars stmt Λ Δ σ,
4
5      (* Premises *)
6      evars Δ EmptyLEnv vars Λ →
7      validss Δ σ Λ stmt →
8
9      (* Side conditions *)
10     ∼NatMap.In id_p Δ →
11
12     (forall id_s,
13         NatSet.In id_s sl →
14         exists t, MapsTo id_s (Declared t) Δ ∨ MapsTo id_s (Input t) Δ) →
15
16     (* Conclusion *)
17     ebeh 𝒟 Δ σ (cs_ps id_p sl vars stmt) (NatMap.add id_p (Process Λ) Δ) σ
```

```
18
19   | EBehComp :
20       forall Δ σ idc ide gmap ipmap opmap
21              M Δc σc formals actuals cdesign,
22
23       (* Premises *)
24       emapg (NatMap.empty value) gmap M →
25       edesign D M cdesign Δc σc →
26       validipm Δ Δc σ ipmap formals →
27       validopm Δ Δc opmap formals actuals →
28
29       (* Side conditions *)
30       ∼NatMap.In idc Δ →
31       ∼NatMap.In idc (compstore σ) →
32       MapsTo ide cdesign D →
33       (forall g, NatMap.In g M → exists t v, MapsTo g (Generic t v) Δc) →
34
35       (* Conclusion *)
36       ebeh D Δ σ (cs_comp idc ide gmap ipmap opmap)
37               (NatMap.add idc (Component Δc) Δ)
38               (cstore_add idc σc σ)
39
40   | EBehNull: forall Δ σ, ebeh D Δ σ cs_null Δ σ
41
42   | EBehPar:
43       forall Δ Δ' Δ'' σ σ' σ'' cstmt cstmt',
44         ebeh D Δ σ cstmt Δ' σ' →
45         ebeh D Δ' σ' cstmt' Δ'' σ'' →
46         ebeh D Δ σ (cs_par cstmt cstmt') Δ'' σ''.
```

LISTING 4.10: The implementation of the `ebeh` behavior elaboration relation with the Coq proof assistant.

4.8.3 Implementation of the simulation algorithm

The full simulation relation (cf. Section 4.6.1) formalizes the H-VHDL simulation algorithm. The Coq implementation of the full simulation relation, presented in Listing 4.11, is a strict translation of Rule FULLSIM. At Lines 14 and 15, the term (`behavior d`) represents the concurrent statements defining the behavior of the H-VHDL design `d` (i.e. d.cs in the formal rule). Line 13 corresponds to the elaboration phase, Line 14 to the initialization phase, and Line 15 to the main simulation loop.

```
1   Inductive fullsim
2           (D : IdMap design)
3           (M : IdMap value)
4           (Ep : nat → Clk → IdMap value)
5           (τ : nat)
```

```
 6            (Δ : ElDesign)
 7            (d :  design): list DState → Prop :=
 8
 9  | FullSim :
10      forall σ_e σ_0 θ,
11
12        (* * Premises * *)
13        edesign 𝒟 ℳ d Δ σ_e →
14        init 𝒟 Δ σ_e (behavior d) σ_0 →
15        simloop 𝒟 E_p Δ σ_0 (behavior d) τ θ →
16
17        (* * Conclusion * *)
18        fullsim 𝒟 ℳ E_p τ Δ d (σ_0 :: θ).
```

LISTING 4.11: The implementation of the full simulation relation with the Coq
proof assistant.

The `simloop` relation appeals to the `simcycle` that implements the simulation cycle relation defined in Section 4.6.3. Listing 4.12 presents the implementation of the `simcycle` relation. The `simcycle` relation is a strict transcription of the SIMCYC rule. At Line 13, the `vrising` relation implements the ↑ relation, i.e. the rising edge phase of the cycle. At Line 15, the `vfalling` relation implements the ↓ relation, i.e. the falling edge phase of the cycle. At Lines 14 and 16, the `stabilize` relation implements the ⤳ relation, i.e. the stabilization phases of the simulation cycle. At Lines 18 and 19, the `IsInjectedDState` relation implements the `Inject` relation. Line 18 states that the σ_i state is the result of the injection of the map ($E_p\ \tau$) in the signal store of state σ.

```
 1  Inductive simcycle (𝒟 : IdMap design) (E_p : nat → IdMap value)
 2      (Δ : ElDesign) (τ : nat) (σ : DState) (behavior : cs)
 3      (σ' σ'' : DState): Prop :=
 4  | SimCycle : forall σ_i σ_↑ σ_↓,
 5
 6        (* * Premises * *)
 7        vrising 𝒟 Δ σ_i behavior σ_↑ →
 8        stabilize 𝒟 Δ σ_↑ behavior σ' →
 9        vfalling 𝒟 Δ σ' behavior σ_↓ →
10        stabilize 𝒟 Δ σ_↓ behavior σ'' →
11
12        (* * Side conditions * *)
13        IsInjectedDState σ (E_p τ) σ_i →
14
15        (* * Conclusion * *)
16        simcycle 𝒟 E_p Δ τ σ behavior σ' σ''.
```

LISTING 4.12: The implementation of the simulation cycle relation with the Coq
proof assistant.

4.9 Conclusion

In this chapter, we gave an overview of the VHDL language and its informal simulation semantics. Then, considering our needs, that is considering the content of the VHDL programs generated by the HILECOP model-to-text transformation, we defined a synthesizable and synchronous subset of the VHDL language called $\mathcal{H}$-VHDL. We gave a small-step semantics to $\mathcal{H}$-VHDL by formalizing a simplified simulation algorithm. The simulation algorithm yields a simulation trace, i.e. time-ordered list of states, corresponding to the execution of the behavior of a $\mathcal{H}$-VHDL design over multiple clock cycles. The formalization of the $\mathcal{H}$-VHDL semantics also includes the formalization of the design elaboration. The elaboration, prior to the simulation, ensures the well-formedness and the well-typedness of a $\mathcal{H}$-VHDL design. Moreover, we have implemented the $\mathcal{H}$-VHDL syntax and semantics with the Coq proof assistant.

Ever since the mechanization of the proof of behavior preservation has begun, the semantics of $\mathcal{H}$-VHDL has been evolving. Section 4.3, 4.4, 4.5 and 4.6 present the most recent version of the semantics. We realized that keeping an operational semantics as close as possible to the VHDL simulation algorithm added complexity to the proof process. For instance, in the VHDL simulation algorithm, the body of a process is executed during the stabilization phase only if one signal of its sensitivity list is part of the current state's event set. However, it is through the execution of the body of a process with the rules of the $\mathcal{H}$-VHDL semantics that we can determine the *combinational* equation associated with the value of a signal. In the proceeding of the proof of semantic preservation, we must often describe the value of a signal based on the value of its input, or *source*, signals (cf. Section 6.4). Due to the event-based system of resuming a process activity, a combinational process can sometimes never be executed during a stabilization phase. Say that process p assigns signal s with the expression a and b, where a and b are two signals. If the process p is not executed, then we will not be able to state that $s = a$ and b, even though this equation always holds. We had to carry extra hypotheses in the definition of our lemmas to deal with this problem. Finally, our current semantics always executes the body of combinational processes during a stabilization phase, thus permitting us to easily determine the combinational equation tied to a signal. By doing this kind of simplification, we realized that we were heading toward a semantics that was closer to the "synthesis" semantics we talked about at the beginning of the chapter. This semantics tends to get closer to the rules of the combinational logic and the synchronous logic. These rules that a hardware system designer has in mind when devising a model with a hardware description language.

Chapter 5

The HILECOP model-to-text transformation

The aim of this chapter is to present the details of the HILECOP model-to-text transformation that we propose to verify as semantic preserving. The chapter is structured as follows. First, we make an overall description of the HILECOP transformation. Then, we present, in Section 5.2, a literature review of the works pertaining to transformation functions in the context of formal verification. The literature review focuses on the expression of transformation functions and on their implementation. In Section 5.3, we thoroughly present the HILECOP transformation function in the form of a pseudo-code algorithm. Finally, in Section 5.4, we describe the Coq implementation of the algorithm. Note that, in the following chapter, we refer to the generic constant, internal signal and port identifiers defined in the `place` and `transition` designs through their abbreviated names (see Table D.1).

5.1 Informal presentation of the HILECOP model-to-text transformation

This section outlines the main phases of the HILECOP model-to-text transformation function. The goal is to give to the reader the means to appreciate the differences and the similarities between the HILECOP transformation and the other transformations presented in the literature review of Section 5.2. Then, Section 5.3 will enter the details of the transformation by presenting the transformation algorithm.

The HILECOP model-to-text transformation function takes an SITPN model as input; then, it generates a top-level $\mathcal{H}$-VHDL design out of the input model. We will illustrate each step of the HILECOP model-to-text transformation through the transformation of the input SITPN model presented in Figure 5.1.
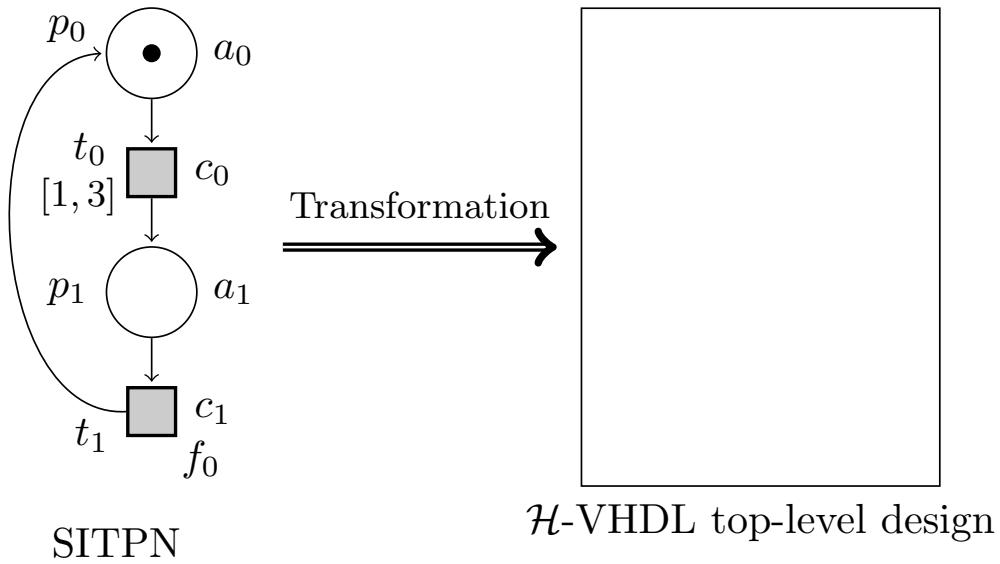
FIGURE 5.1: Transformation of an input SITPN model into a top-level $\mathcal{H}$-VHDL design. The input model is composed of two places, p_0 and p_1, and two transitions, t_0 and t_1. The transition t_0 is associated with the time interval $[1,3]$ and the condition c_0. The transition t_1 is associated with the condition c_1, and its firing triggers the execution of the function f_0. The action a_0 is activated when the place p_0 is marked, and the action a_1 is activated when the place p_1 is marked.

The generated top-level design implements the structure of the input SITPN. As a first step, the transformation generates, for each place of the input SITPN, a component instance of the `place` design, and, for each transition of the input SITPN, a component instance of the `transition` design. These subcomponents constitute the main part of the $\mathcal{H}$-VHDL top-level design's architecture (i.e. its internal behavior). Figure 5.2 shows a graphical representation of the input and output port interfaces of the `place` and `transition` designs. All PCIs (Place Component Instances) and TCIs (Transition Component Instances) generated during the first step of the HILECOP transformation inherit the interface presented in Figure 5.2.

place design interface

transition design interface
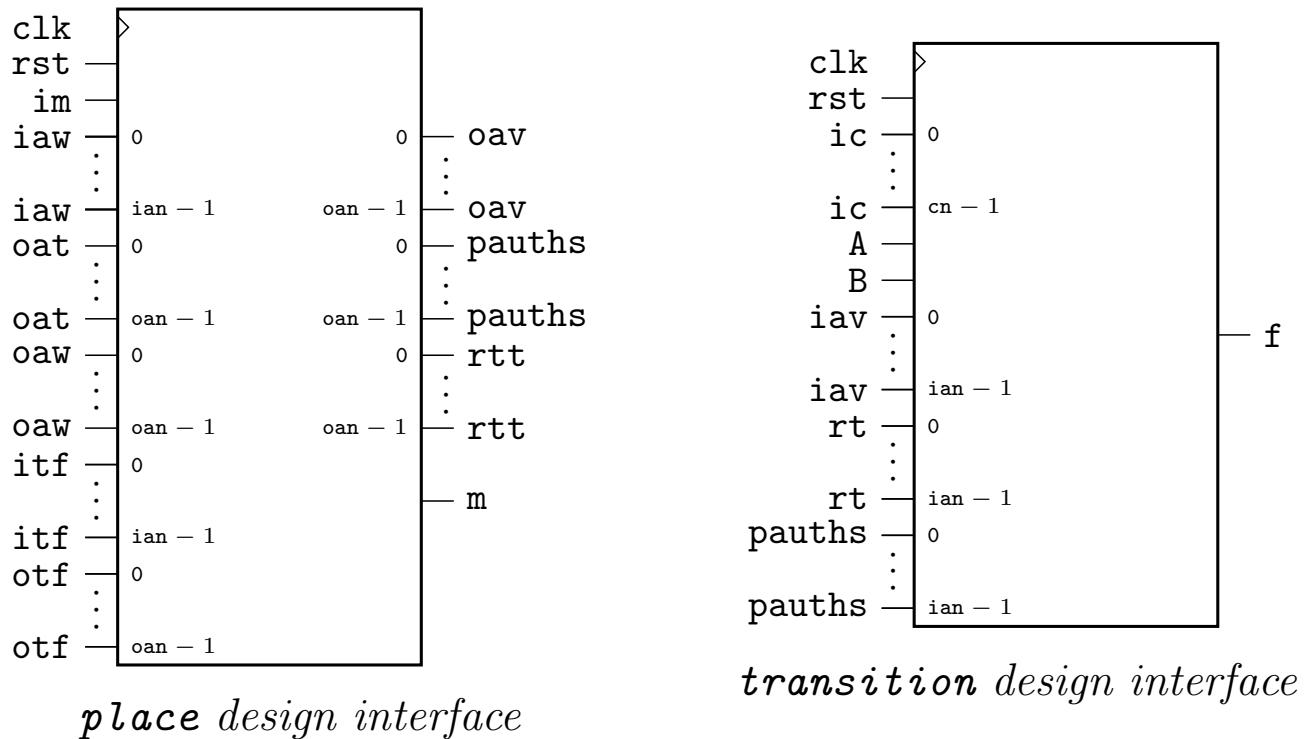
FIGURE 5.2: On the left, the `place` design interface and on the right the `transition` design interface. The indexes of composite ports are expressed at the inner extremity of the pins, while the name (abbreviated) of ports are expressed at the outer extremity.

During the first generation step of the HILECOP transformation, each PCI and TCI receive a value for each of their generic constants through the creation of generic maps. In the generic map of a TCI id_t (implementing a transition t), the `ian` constant is associated with the number of input arcs of t, the `cn` constant with the number of conditions attached to t, etc. In the generic map of a PCI id_p, the `ian` constant is associated with the number of input arcs of p, the `oan` constant with the number of output arcs of p, and the `mm` constant with the maximal marking value of p. The maximal marking value associated with a given place p of the input SITPN is an information passed as a parameter to the transformation function. This information comes from the analysis of the input SITPN pertaining to the *boundedness* of the input model. In the definition of the HILECOP methodology, this analysis takes place before the transformation of the input SITPN into a $\mathcal{H}$-VHDL design. The generic constants do not appear as pins in the interfaces of the `place` and `transition` designs presented in Figure 5.2. The generic constants have an impact of the structure of the interface of each component instance. For example, Figure 5.2 shows the dependency between the size (i.e. the number of pins) of composite ports and the value of generic constants, e.g. the size of `iaw` input port of the `place` design depends on the `ian` generic constant. Thus, the generation of generic maps during this first generation step corresponds to the *dimensioning* of the PCIs and TCIs; this is when the number of pins of composite ports are determined.

Figure 5.3 shows the architecture of the top-level design resulting of the first generation step of the HILECOP transformation.
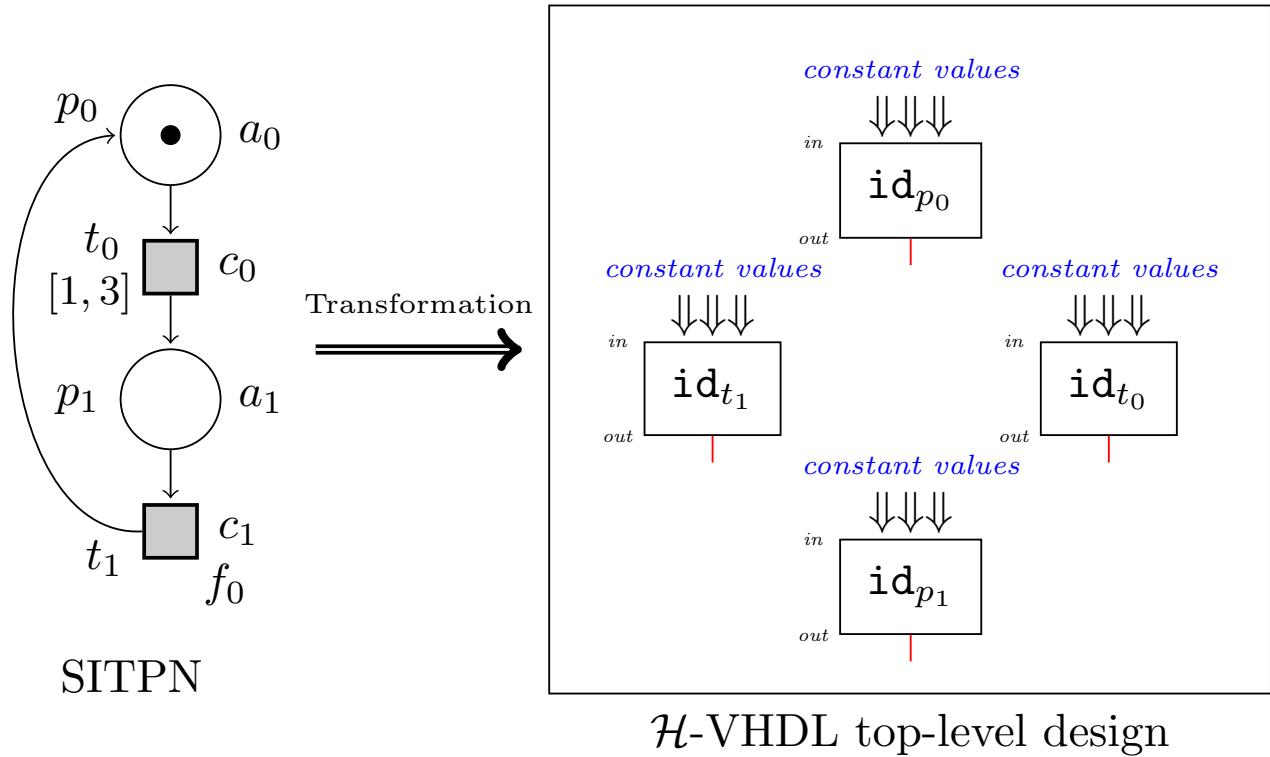
FIGURE 5.3: Generation of the `place` and `transition` component instances based on the set of places and transitions of the input SITPN. The PCI id_{p_0} implements the place p_0, TCI id_{t_0} the transition t_0... In red, the internal signals connected to the `marked` port of PCIs and to the `fired` port of TCIs.

During the first transformation step, illustrated in Figure 5.3, the input and output port maps of PCIs and TCIs are also partly generated. In the manner of the generic constants in generic maps, some input ports are associated with constant values in the input port maps of PCIs and TCIs. All these associations are generated during this first step. Also, the `marked` output port of every PCI is associated with an internal signal in the output port map of the PCI. The internal signal will be connected later in the course of the transformation. The same holds for the `fired` output port of every TCI.

After the first transformation step, the component instances are interconnected through their port interfaces. Figure 5.4 illustrates the behavior of the top-level design after the interconnection of PCIs and TCIs.
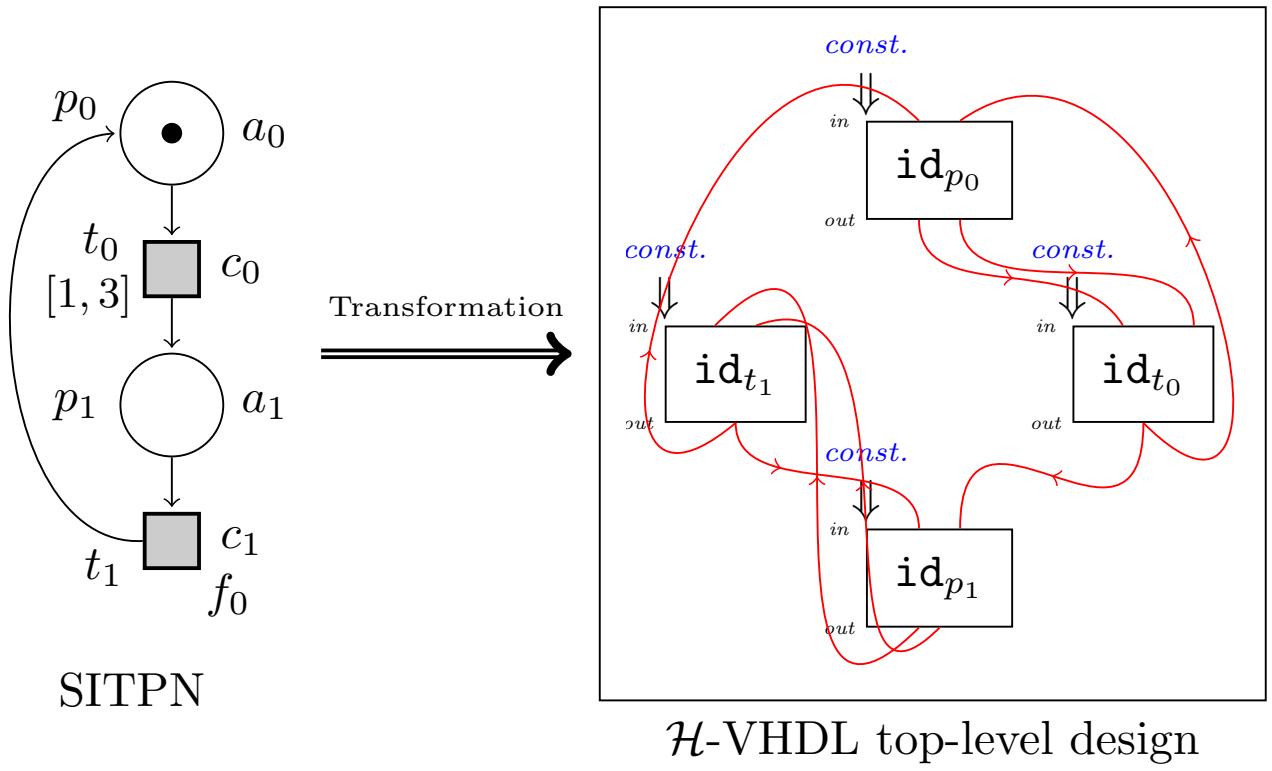
FIGURE 5.4: Generation of the interconnections between the `place` and `transition` component instances. In red, the internal signals interconnecting the PCIs and the TCIs. These signals are generated by the transformation. The arrows indicate the sense of propagation of the information. In blue, the constant associations (i.e. the generic maps and a part of the input port maps) produced during the previous transformation step.

The PCIs and TCIs interact through their interfaces to exchange informations. For instance, a PCI id_p, implementing a given place p, separately informs its output TCIs (i.e. the TCIs implementing the output transitions of p) that its current marking enables them. The marking of a PCI is represented by the value of its internal signal `s_marking`. A PCI is the only one to have access to the current value of its internal signals. Thus, a PCI must communicate to its output TCIs their sensitization status. To perform this exchange of information, the transformation generates an internal signal to connect a specific output port of a PCI (the `oav` port) to a specific input port of the output TCIs (the `iav` port). Likewise, a TCI informs its input and output PCIs about its firing status. The transformation generates an internal signal to connect the `fired` output port of a TCI to the `itf` and `otf` input ports of the input and output PCIs. These interconnections are performed by adding new associations in the input port map and output port map of PCIs and TCIs. Through the execution of the internal behavior of each PCI and TCI, and, through the interconnection of component instances, the transformation aims at generating a design's behavior that, by its inherent structure, carries the rules of the SITPN semantics and conforms to the execution of the input SITPN model.

To reduce the size of circuits after the synthesis on an FPGA or ASIC, PCIs and TCIs only communicate with Boolean signals through their interfaces. To restrict the interconnections to

Boolean signals, the `place` design, which is the mold of all PCIs, carries the arc information (i.e. the weight and type of its input and output arcs) in its interface; this approach of encoding the arc information is called the *place-pivot* approach. Figure 5.5 points out where the arc information is encoded in the interface of the `place` design. Thus, a PCI has all the needed information to compute the sensitization of its output TCIs by comparing the weight of its output arcs to its current marking value. A PCI can simply communicate through a Boolean signal that it is currently enabling its output TCIs. In the other approach, the *transition-pivot* approach, the `transition` design carries the arc information. In that case, the TCIs compute their own sensitization status. To be able to do so, the PCIs must communicate their current marking value to the TCIs. As a marking value is a natural number, the number of interconnecting signals between PCIs and TCIs greatly increases in the *transition-pivot* approach. Eventually, the *place-pivot* approach has been retained in the current version of HILECOP.
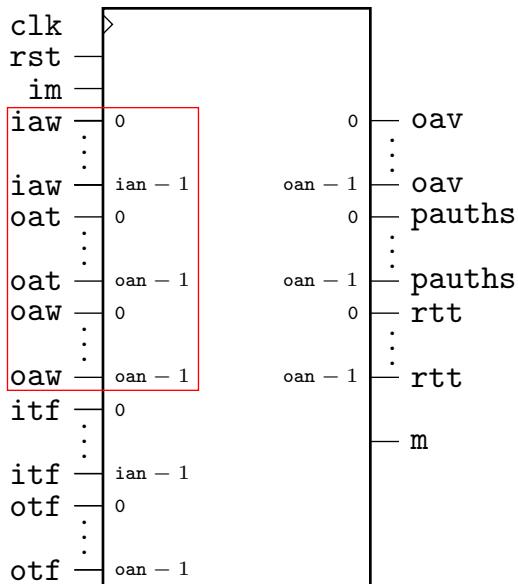
FIGURE 5.5: Inside the red frame, the arc information encoded through the `iaw`, `oat` and `oaw` input ports in the interface of the `place` design.

The last part of the transformation deals with the interpretation elements of the input SITPN, i.e. the conditions, the actions and the functions. Each condition of the input SITPN leads to the declaration of a Boolean input port in the port clause of the top-level design. As it was pointed out in Chapter 3 (cf. Section 3.1.1), the interpretation aspect has been greatly simplified in the SITPN structure, and the generation and the association of an input port to each condition of the input SITPN is a consequence of the simplification. In the *full* version of the SITPN structure, a condition depends on a Boolean expression that involves both the value of internal signals and input ports of the top-level design. In our simplified version of the SITPN structure, a condition value depends on the execution environment, i.e. a function that updates the value of conditions at each falling edge of the clock signal. Thus, we find it natural to transform each condition into an input port of the top-level design, as the value of both depends on the execution/simulation environment. Then, each input port representing

a condition is connected to the ic input port of TCIs. The interconnection of an input port of the top-level design to the ic input port of a TCI reflects an existing association between a transition and a condition of the input SITPN model.

For each action and function of the input SITPN, the transformation generates a Boolean output port, a.k.a. an *action* or a *function* port. At runtime, the value of these output ports represent the activation or execution status of the corresponding actions and functions. To determine the value of the action and function ports, the transformation generates two processes: the action process and the function process. The action process is a synchronous process responding to the falling edge of the clock signal. At the occurrence of the falling edge of the clock signal, the action process sets the value of the *action* ports computed from the values of the multiple marked output ports[1]. The marked port is an output port of the place design. Through the marked port, the PCIs inform the outside about their marking status, i.e. if they possess at least one token or not. Remember that the transformation generated an association between the marked output port and an internal signal in the output port map of PCIs during the first transformation step. These internal signals are read by the action process to assign a value to the *action* ports of the top-level design. The function process is a synchronous process responding to the rising edge of the clock signal. At the occurrence of the rising edge of the clock signal, the function process sets the value of the *function* ports computed from the values of the fired output ports. The fired port is an output port of the transition design. Through the fired port, the TCIs inform the outside about their firing status, i.e. if they are fired or not. Remember that, during the first transformation step, the transformation generated an association between the fired output port and an internal signal in the output port map of TCIs. These internal signals are read by the function process to assign a value to the *function* ports of the top-level design. Figure 5.6 presents the top-level $\mathcal{H}$-VHDL design at the end of the transformation.

---

[1]As one action can be associated to multiple places, one action port can depend on the value of multiple marked output port.
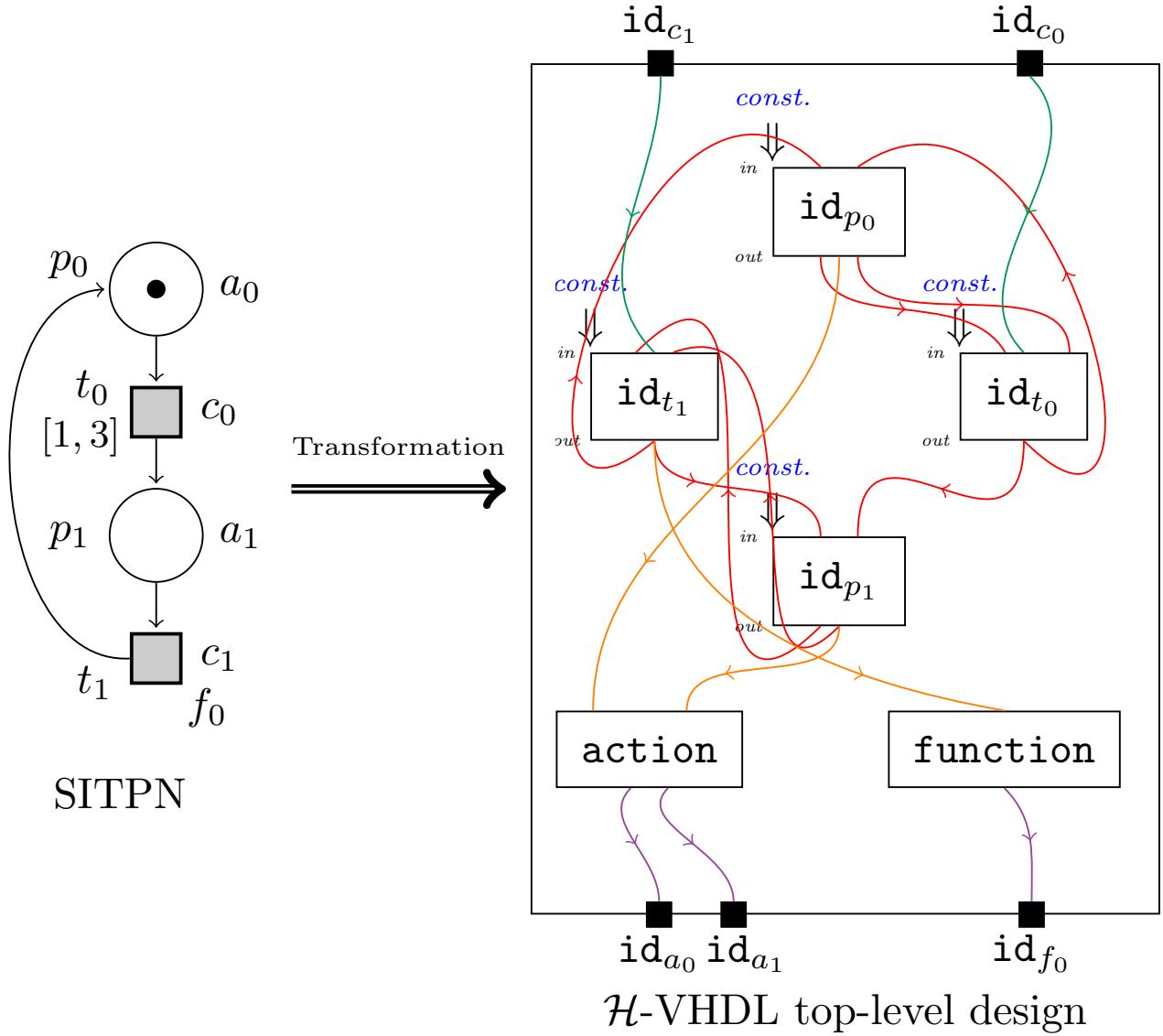
FIGURE 5.6: Generation of the input and output ports, and of the `action` and the `function` processes in the $\mathcal{H}$-VHDL top-level design. The *primary* input port id_{c_1} (resp. id_{c_0}) implements the condition c_1 (resp. c_0). In green, the internal signals, generated by the transformation, connecting the input ports of the top-level design to the `input_conditions` input port of TCIs. The id_{a_0} and id_{a_1} output ports reflect the activation status of the actions a_0 and a_1. The id_{f_0} output port reflects the activation status of the function f_0. In orange, the internal signals, generated by the transformation, connecting the `marked` and `fired` output ports of PCIs and TCIs to the `action` and `function` processes. In purple, the representation of the assignments performed by the `action` and `function` processes and that set the value of the action and function ports of the top-level design.

5.2 Expressing transformation functions

In this section, we present our literature review pertaining to transformation functions in the context of formal verification. Here, a transformation function is understood as any kind of mapping from a source representation to a target representation, where the source and target representations possess a behavior of their own (i.e. they are executable). We use the same articles to perform our literature review in this chapter and in the following chapter, i.e. Chapter 6. However, our research questions, i.e. the questions we try to give an answer to while reading the articles, and our presentation axis differ from one chapter to the other. Here, the following questions guide our reading:

– Is there a proper way to build a transformation function? Are there standards depending on the application domain?

– How can we build a modular, extensible transformation function?

– How can we build a transformation function that will ease the proof of semantic preservation?

The goal is to inspire ourselves with the works of the literature, and to see how far the correspondence holds between our specific case of transformation, and other cases of transformations. The material we used for the literature review is divided in three categories. Each category covers a specific case of transformation function. The three categories are:

– Compilers for generic programming languages

– Compilers for hardware description languages

– Model-to-model and model-to-text transformations

Note that, in the case of compilers for programming languages, the term *translation* is preferred over transformation to talk about the generation of a target program from a source program.

5.2.1 Building transformation functions

As the authors state in [104], "Although theoretically possible, verifying a compiler that is not designed for verification would be a prohibitive amount of work in practice." The question is to know how to design such a compiler? How to anticipate the fact that we will have to prove that the compiler is semantic preserving? Now, let us consider these questions in the more general context of transformation functions that map a source representation to a target one.

Compilers for generic programming languages

In the context of formally verified compilers for generic programming languages, the translation from a source program to a target program is straight forward. While descending recursively through the AST of the input program, each construct of the source language is mapped to one or many constructs of the target language. Figure 5.7 gives an example of the translation from Java program expressions to Java bytecode expressions, set in the context of a compiler for Java programs written within the Isabelle/HOL theorem prover [103]. Here, the mapping between source and target constructs is clearly defined.

```
mkExpr jmb (NewC c) = [New c]
mkExpr jmb (Cast c e) = mkExpr jmb e @ [Checkcast c]
mkExpr jmb (Lit val) = [LitPush val]
mkExpr jmb (BinOp bo e1 e2) = mkExpr jmb e1 @ mkExpr jmb e2 @
   (case bo of
       Eq => [Ifcmpeq 3,LitPush(Bool False),Goto 2,LitPush(Bool True)]
     | Add => [IAdd])
mkExpr jmb (LAcc vn) = [Load (index jmb vn)]
mkExpr jmb (vn::=e) = mkExpr jmb e @ [Dup , Store (index jmb vn)]
mkExpr jmb ( cne..fn ) = mkExpr jmb e @ [Getfield fn cn]
mkExpr jmb (FAss cn e1 fn e2 ) =
   mkExpr jmb e1 @ mkExpr jmb e2 @ [Dup_x1 , Putfield fn cn]
mkExpr jmb (Call cn e1 mn X ps) =
   mkExpr jmb e1 @ mkExprs jmb ps @ [Invoke cn mn X]
mkExprs jmb [] = []
mkExprs jmb (e#es) = mkExpr jmb e @ mkExprs jmb es
```

FIGURE 5.7: Translation from Java expressions to Java bytecode expressions

In the works pertaining to the well-known CompCert project [72, 13], the many steps that compose the compiler from C programs to assembly programs are also clearly mapping each construct of source program to target program constructs. Moreover, the pattern matching possibilities offered by languages like Coq, Isabelle, HOL and other interactive theorem provers enable a clear and concise implementation of compilers.

The cases of optimizing compilers like [72] and [105] show that, to avoid writing too complex functions when passing from a source to a target program, the compilation is decomposed into many passes. No more than 12 passes for the CakeML compiler, and up to 20 passes for CompCert. This is a way to keep the translation functions simple enough in order to ease reasoning afterwards. Indeed, the more the gap is important between the source representation and the target one, the more the translation function will be complex.

Another point that is noticeable while expressing a translation function is the necessity to keep a binding between the source and the target representations. For instance, in CompCert, when passing from transformed C programs to an RTL representation (based on registers and control flow graphs), a binding function γ links the variables of a C program to the registers generated in the RTL representation of the program. The binding is necessary for both the translation and the proof of semantic preservation. During the translation, it permits to replace the variables by their corresponding registers in the RTL code. During the proof of semantic preservation, the link that exists between a variable and a register indicates which elements must be compared to prove that the execution state of the source representation is similar to

the execution state of the target representation. The generation of this binding function must be integrated to the design of the translation function.

In [72], and [30], compilers are written within the Coq proof assistant. Compilers are expressed using the state-and-error monad, thus mimicking the traits of imperative languages into a functional programming language setting. In Section 5.3, we present the HILECOP transformation in the form of an imperative pseudo-code algorithm. The state-and-error monad is well-suited to the implementation of this kind of algorithm with a functional language like Coq; thus, we chose to apply this monad to our implementation of the transformation algorithm (see Section 5.4).

Compilers for hardware description languages

The other category of compilers that we are interested in are compilers for hardware description languages (HDL). The HILECOP methodology's goal is the design of hardware circuits. For that reason, we are interested in studying the case of compilers for HDLs. However, one can notice that compiling an HDL program into a lower level representation is one level of abstraction down compared to the transformation we propose to verify. Indeed, it corresponds to Step 3 in the HILECOP methodology (cf. Figure 1.2), i.e. the transformation of VHDL source code into an RTL representation.

In the context of formal verification applied to HDLs compilers, only a few works describe the specificities of their translation function.

In [22], the authors define the FeSi language (a refinement of the BlueSpec language, a specification language for hardware circuit behaviors), and its implementation within the Coq proof assistant. The authors present the syntax and semantics of the FeSi language and of the RTL language which is the target language of the compiler. FeSi programs are composed of simple expressions, and actions permitting to read or write from different types of memory (registers). Therefore, the abstract syntax is divided into the definition of expressions and the definition of actions, i.e: control flow instructions and operations on memory. The RTL language is composed of expressions and write operations to registers. The authors are more interested in proving that a FeSi specification is well-implemented by a given Coq program, than giving the details of the translation from FeSi to RTL. However, the translation seems straight-forward, and proceeds as usual by descending through the AST of FeSi programs.

In [18], the authors present a compiler for the language Koîka, which is also a simpler version of the BlueSpec language. A Koîka program is composed of a list of rules; each rule describes actions that must be performed atomically. Actions are read and write operations on registers. A Koîka program is accompanied by a scheduler that specifies an execution order for the rules. The described compiler transforms Koîka programs into RTL descriptions of hardware circuits. The translation function builds an RTL circuit by descending recursively down the AST of rules. Each action is translated into a specific RTL representation which are afterwards composed together to get complex circuits. The translation becomes trickier when it comes to decide the composition of RTL circuits to respect the execution order prescribed by the scheduler.

In [19], the authors present the verification of a compiler toolchain from Lustre programs to an imperative language (Obc), and from Obc to Clight. The Clight target is the one defined in

CompCert [72]. Lustre permits the definition of programs composed of nodes that are executed synchronously. Nodes treat input streams and yield output streams of values. A node body is composed of sequence of equations that determine the values of output streams based on the input. Obc programs are composed of class declarations. A class declaration has a vector of memory variables, a vector of instances of other classes, and method declarations. The translation turns each node of a Lustre program into a class of Obc accompanied by two methods: the reset method, for the initialization of the streams, and the step method, for the update of values resulting of a synchronous step.

In [76], the authors describe a compiler that transforms Verilog programs into netlists targeting given FPGA models. Verilog programs are a lot like VHDL programs; they describe a hardware circuit behavior in terms of processes. A netlist is composed of registers, variables and a list of cells corresponding to combinational components. During the translation process, the expressions of the Verilog programs are turned into netlist cells, and the composition of statements leads to the creation of complex circuits by means of cell composition.

In [60], the authors describe a High-Level Synthesis (HLS) tool that transforms C programs into Verilog descriptions. Moreover, they have formally proved the correctness of the whole process, and mechanized the proof with Coq. In HLS, the purpose of the transformation is to obtain a hardware implementation, described with an HDL, of a computer program. The verified transformation from C to Verilog described in the paper is decomposed into several transformations: from C to a CFG-representation language called 3AC, from 3AC to the intermediate language HTL, and from HTL to Verilog. The transformation from C to 3AC relies on the CompCert compiler. In the second transformation pass, i.e. from 3AC to HTL, the HTL language permits the FSM-based representation of a hardware design. Thus, going from 3AC to HTL implies transforming C statements into a complex FSM ecosystem split into a data-path part and a control logic part. The data-path part involves a RAM representation of the hardware memory. Therefore, in the generated HTL program the RAM memory is a part of the code, whereas it does not appear in the input program.

Model transformations

We will now present the works pertaining to model-to-model and model-to-text transformations in the context of formal verification. Because of the nature of the transformation we propose to verify, i.e a model-to-text transformation, the following works are of particular interest to us. We will focus here on the manner to express transformations in the case of model-to-model and model-to-text transformations. Also, we tried to find articles related to model transformations involving Petri nets.

In [7], the authors observe that Model-Driven Engineering (MDE) is all about model transformation operations. They propose to set a formal context within the Coq proof assistant to verify that model transformations preserve the structure of the source models into the target models. To illustrate their methodology, they choose to transform UML state machine diagrams into Petri net models. The translation rules from source to target models are expressed within the setting of the OMG standard QVT language (Query/View/Transform). The QVT language offers a formal way to express model transformations, partly based on the Object Constraint Language (OCL). The translation rules map the different kinds of structures that

can be found in UML state diagrams to specific structures of Petri nets. Even though the two models used as source and target of transformations are executable, the authors leverage the formal context provided by Coq to prove that the expressed transformations preserve certain structural properties.

In [25], the authors describe a process for model transformation where transformation rules are expressed with the Atlas Transformation Language (ATL). Transformation rules in ATL involve both declarative (OCL) and imperative (match rules) instructions. The authors show how the ATL rules can easily be translated into Coq relations. An example is given on the kind of model-to-model transformations that can be implemented that way. The example is a UML class diagram to relational database model transformation.

In [34], the authors explore the different ways to give a formal semantics to a Domain-Specific Language (DSL) in the context of MDE. Here, the syntax of a given DSL is expressed with a meta-model. An instantiation of this meta-model (a model) yields a DSL program. The authors specify a transformation from a DSL model to another executable model, thus providing an *translational* semantics to the DSL model. The authors illustrate their approach with a source DSL named xSPEM, which is a process description language. The target models are timed PNs. The transformation is expressed through a structural mapping; i.e, each element of an xSPEM model is mapped to a particular PN: an activity is mapped to a subnet, a resource to a single place, connection from activity to resource through parameter is mapped to a connection of transitions and places in the resulting PN...

In [46], the authors address the problem of expressing model transformations by using transformation graphs. Precisely, the kind of transformation graphs that are used are called Triple Graph Grammar (TGG). A TGG is a triplet $<s, c, t>$ where the "correspondence model c explicitly stores correspondence relationships between source model s and target model t".

The work described is [50] is really close to our own verification task. The article describes how Coloured Petri Nets (CPNs, specifically LLVM-labelled Petri nets) are transformed into LLVM (Low Level Virtual Machine) programs representing the state space (the graph of reachable markings) of these PNs. LLVM is a low-level assembly language. The aim is to enable an efficient model-checking of the CPNs. LLVM-labelled PNs are CPNs whose places, transitions and arcs have LLVM constructs for color domains. Places are labelled with data types. Transitions are labelled with boolean expressions that correspond to the guard of the transition. Arcs are labelled by multisets of expressions. A marking is a function that maps each place to a multiset of values belonging to the place's type. The authors define data structures (multisets, sets, markings...) with interfaces, i.e. sets of operations over structures, to represent the Petri nets in LLVM. They define interpretation functions that draw equivalences between Petri nets objects and LLVM data structures. The authors define two algorithms: `fire_t` and `succ_t` to compute the graph of reachable states. These are the functions that transform CPNs into concrete LLVM programs.

In [77], the authors describe a transformation from UML state machine diagrams to Coloured Petri Nets (CPNs). The aim is to leverage the means of analysis provided by Petri nets to certify certain properties over UML state machine diagrams. The authors want to verify that the transformation preserve structural properties between source and target models. The transformation function does not use a standard setting as QVT or ATL, or transformation graphs. It is expressed as a specific function written in Isabelle/HOL.

In [114], the authors present a transformation from Architecture Analysis and Design Language (AADL) models to Timed Abstract State Machines (TASMs). AADL is a language widely used in avionics to describe both hardware and software systems. AADL doesn't have a lot of tools to analyze and simulate the designed systems; therefore transforming AADL models into TASM enables the use of an important toolbox for analysis, and simulation. The transformation from AADL to TASMs are described with ATL rules.

Discussions on how to build transformation functions in the context of semantic preservation

Transformation functions are mappings from a source representation to a target representation. The more the mapping from source to target is straight-forward the easier the comparison will be when proving that the transformation is semantic preserving. Thus, in [72, 105, 30] where complex cases of optimizing compilers are presented, the compilation is split into many simple passes to ease the verification effort coming afterwards. In the case of the HILECOP transformation, we are not yet concerned with the optimization of the generated VHDL code. Thus, our transformation algorithm performs the generation of the target $\mathcal{H}$-VHDL design in a single pass. We do not need to use intermediary representations between the input SITPN model and the generated $\mathcal{H}$-VHDL design.

Also, while transforming source programs, the compiler must often generate fresh constructs belonging to the target language (for instance, generating a fresh RTL register for each variable referenced in a source C program in [72]). The compiler must keep a binding, that is, a memory of the mapping between the elements of the source program and their mirror in the target program. This consideration is of interest in our case of transformation where the elements of SITPNs are also mirrored by elements in the generated $\mathcal{H}$-VHDL design.

It remains hard to establish a standard way to express a transformation function as it really depends on the form of the input and the output representations. Compilers for programming languages tend to be a lot more compositional than model transformations. Here, the word *compositional* means that the translation rules can be split into simple and independent cases of translation, e.g. translation of expressions, then translation of statements, then translation of function bodies,...This is a huge advantage to perform the proof of semantic preservation. Indeed, this decomposition of a translation function permits to reason on simple translation cases; yet, each of these translations cases yields a piece of target code that can be executed or interpreted in an independent manner. In the case of the HILECOP, we tried as much as possible to express the transformation in a compositional way. First, we tried to devise the transformation by building up transformation functions for each element of the SITPN structure, i.e.: a transformation function for the places, another for the transitions...However, due to the interconnections that exist between the component instances of the generated $\mathcal{H}$-VHDL design, it is impossible to define transformation functions that would yield stand-alone executable code.

In the world of models, there exist some standard formalisms to express transformation rules (QVT, ATL, transformation graphs...). However, the complexity of the transformation rules depends on the richness of the elements composing the source model, and the distance to the concepts of the target model. In our case, we were not able to grab the perks of using such formalisms as QVT or ATL to devise our transformation.

5.3   The transformation algorithm

Before detailing the algorithm underlying the HILECOP model-to-text transformation, we want to point out the necessity to automate the transformation. Judging by the appearance of the $\mathcal{H}$-VHDL design generated from the input SITPN model, the reader could rightly ask why the designers of hardware circuits that are using the HILECOP methodology do not start directly by writing down the VHDL code. The reasons are many. First, handling the interconnections between PCIs and TCIs is simple enough when the number of places and transitions of the input SITPN is few, however, it becomes a lot more tedious with the increase of the size of models. To give an example, the Neurrinov company[2], which applies the HILECOP methodology to the design of critical digital circuits, has developed a digital circuit model for the control of the electro-stimulation in neuroprostheses. Once flattened down, the model is composed of 1097 places and 1666 transitions. The top-level VHDL design generated from this model represents up to 140000 lines of code. Obviously, the hand-coding of this input model into a VHDL design would be too error-prone. Moreover, the PN models offer a lot of opportunities in terms of analysis and model-checking compared to the ones that exist for VHDL code. Finally, the graphical aspect of PNs appears to be more fit for the task of digital architecture design in comparison to plain source code, as it facilitates the discussions between designers. For these reasons, we choose to preserve SITPNs as the input models of the HILECOP methodology, and to automatize the transformation into top-level $\mathcal{H}$-VHDL designs.

In this section, we give the algorithm underlying the HILECOP model-to-text transformation. This algorithm is the base of the Coq implementation of the HILECOP transformation; the implementation is presented in Section 5.4. As presented in Chapter 1, there exists a Java implementation of the HILECOP methodology. This implementation performs the generation of VHDL code from a SITPN model. However, the algorithm of the transformation has never been documented, nor a formal specification given. The following algorithm is one of the contribution of this thesis. It has been devised through the examination of the code of the existing Java implementation, and through the discussions with the designers of the HILECOP methodology.

5.3.1   The `sitpn_to_hvhdl` function

The HILECOP transformation algorithm, presented in Algorithm 3, generates a $\mathcal{H}$-VHDL design and a SITPN-to-$\mathcal{H}$-VHDL binder from an input SITPN. A SITPN-to-$\mathcal{H}$-VHDL design binder is a structure that binds the *dynamic* elements of a SITPN, namely: places, transitions, conditions, actions and functions; to the *dynamic* elements of a $\mathcal{H}$-VHDL design, namely: component instance identifiers and signal identifiers. By dynamic elements, we mean these elements that value or characteristics vary in the course of the execution/simulation of the structure. Such a binder is generated alongside the transformation and links a SITPN element to its $\mathcal{H}$-VHDL *implementation*, i.e. the $\mathcal{H}$-VHDL element that will supposedly behave similarly to the source SITPN element at runtime. Thus, the SITPN-to-$\mathcal{H}$-VHDL design binder is at the center of the state similarity relation, presented in Chapter 6, and that enables the comparison between an

---

[2]https://neurinnov.com/

SITPN state and an $\mathcal{H}$-VHDL design state. The formal definition of an SITPN-to-$\mathcal{H}$-VHDL design binder is as follows.

> **Definition 35** (SITPN-to-$\mathcal{H}$-VHDL design binder). *Given a sitpn $\in$ SITPN and a $\mathcal{H}$-VHDL design $d \in$ design, a SITPN-to-$\mathcal{H}$-VHDL design binder $\gamma \in WM(sitpn, d)$ is a tuple $<PMap, TMap, CMap, AMap, FMap>$ where:*
>
> - *$sitpn = <P, T, pre, test, inhib, post, M_0, \succ, \mathcal{A}, \mathcal{C}, \mathcal{F}, \mathbb{A}, \mathbb{C}, \mathbb{F}, I_s>$*
>
> - *$d = $ design id_e id_a gens ports sigs cs*
>
> - *$PMap \in P \rightarrow \{id \mid$ comp$(id, $place$, g, i, o) \in cs\}$*
>
> - *$TMap \in T \rightarrow \{id \mid$ comp$(id, $transition$, g, i, o) \in cs\}$*
>
> - *$CMap \in \mathcal{C} \rightarrow \{id \mid (in, id, t) \in ports\}$*
>
> - *$AMap \in \mathcal{A} \rightarrow \{id \mid (out, id, t) \in ports\}$*
>
> - *$FMap \in \mathcal{F} \rightarrow \{id \mid (out, id, t) \in ports\}$*

As presented in Definition 35, the binder is composed of five sub-environments that map the different SITPN sets to identifiers. The *PMap* and *TMap* sub-environments map the places to their corresponding PCI identifiers, and the transitions to their corresponding TCI identifiers. The *CMap* sub-environment maps the conditions to input port identifiers. The *AMap* and *FMap* sub-environments map the actions and functions to output port identifiers.

> **Notation 9.** *For a given binder γ and an element of an SITPN structure $e \in P \sqcup T \sqcup \mathcal{C} \sqcup \mathcal{A} \sqcup \mathcal{F}$, we write $\gamma(e)$ where e is looked up in the appropriate function. For instance, for a given $f \in \mathcal{F}$, $\gamma(f)$ is a shorthand notation for $FMap(f)$ where $\gamma = <\ldots, FMap>$.*

Algorithm 3 is the algorithm of the HILECOP model-to-text transformation. The algorithm as four parameters; the first one is the input SITPN model *sitpn*; id_e and id_a are the entity and the architecture identifiers for the generated $\mathcal{H}$-VHDL design; $b \in P \rightarrow \mathbb{N}$ is the function associating a maximal marking value to each place of the input SITPN. This function is the result of the analysis of the input SITPN.

Algorithm 3: sitpn_to_hvhdl($sitpn, id_e, id_a, b$)

1   $d \leftarrow$ design id_e id_a $\varnothing \varnothing \varnothing$ null

2   $\gamma \leftarrow \varnothing$

3   generate_architecture($sitpn, d, \gamma, b$)

4   generate_interconnections($sitpn, d, \gamma$)

5   generate_ports($sitpn, d, \gamma$)

6   **return** (d, γ)

In Algorithm 3, Line 1 creates the initial $\mathcal{H}$-VHDL design structure and assigns it to the variable d. Initially, the design has an empty port declaration set, an empty internal signal

declaration set, and a behavior defined by the `null` statement. The design generated by the `sitpn_to_hvhdl` function has an empty set of generic constants; this set stays empty even at the end of the transformation. Line 2 initializes the γ binder with empty sub-environments. From Lines 3 to 5, the called procedures modify the design and the binder structures. Each part of the sequence corresponds to one step of the transformation, which were outlined in Section 5.1. The content of the `generate_architecture` function is detailed in Algorithms 5, 6 and 7. The content of the `generate_interconnections` function is detailed in Algorithm 10. The content of the `generate_ports` function is detailled in Algorithms 11, 12, 13 and 14.

> **Notation 10.** *In the remainder of memoir, we write $\lfloor sitpn \rfloor_b$ to denote* `sitpn_to_hvhdl`*$(sitpn, id_e, id_a, b)$, for all $sitpn \in SITPN$, $b \in P \rightarrow \mathbb{N}$ and identifiers id_a, id_e.*

5.3.2 Primitive functions and sets

The description of further functions and algorithms appeals to some primitive functions and set definitions that we introduce here. Below are all the sets that we use in the description of the algorithms.

– `input`$(p) = \{t \mid \exists \omega \; s.t. \; post(t, p) = \omega\}$, the set of input transitions of a place p.

– `output`$(p) = \{t \mid \exists \omega, a \; s.t. \; pre(p, t) = (\omega, a)\}$, the set of output transitions of a place p.

– `acts`$(p) = \{a \mid \mathbb{A}(p, a) = \text{true}\}$, the set of actions associated with a place p.

– `input`$(t) = \{p \mid \exists \omega, a \; s.t. \; pre(p, t) = (\omega, a)\}$, the set of input places of a transition t.

– `output`$(t) = \{p \mid \exists \omega \; s.t. \; post(t, p) = \omega\}$, the set of output places of a transition t.

– `conds`$(t) = \{c \mid \mathbb{C}(t, c) = 1 \lor \mathbb{C}(t, c) = -1\}$, the set of conditions associated with a transition t.

– `trs`$(c) = \{t \mid \mathbb{C}(t, c) = 1 \lor \mathbb{C}(t, c) = -1\}$, the set of transitions to which a condition c is associated.

– `pls`$(a) = \{p \mid \mathbb{A}(p, a) = \text{true}\}$, the set of places to which an action a is associated.

– `trs`$(f) = \{t \mid \mathbb{F}(t, f) = \text{true}\}$, the set of transitions to which a function f is associated.

Every set presented above are *unordered*. However, we assume that, every time we iterate over the elements of an unordered set with a **foreach** statement, the iteration respects an arbitrary order. This order is always the same through the multiple calls to **foreach** statements. Of course, the iteration over the elements of an *ordered* set with a **foreach** statement respects the natural order of the set.

Now, let us introduce some primitive functions and procedures that we use in the description of the following algorithms.

– $\texttt{output}_c \in P \to 2^T$. The $\texttt{output}_c$ function takes a place p as input and yields an ordered set of transitions computed as follows:

1. If all conflicts between the output transitions of p are solved by mutual exclusion, or if the set of conflicting transitions of p is a singleton, then $\texttt{output}_c$ returns an empty set.

2. Otherwise, the function tries to establish a total ordering over the set of conflicting transitions of p w.r.t the firing priority relation:
   – If no such ordering can be established (in that case, the firing priority relation is ill-formed, and the input SITPN is not well-defined), $\texttt{output}_c$ raises an error.
   – Otherwise, the function returns the ordered set, with the top-level priority transition at the head.

– $\texttt{output}_{nc} \in P \to 2^T$. The $\texttt{output}_{nc}$ function takes a place p as input and yields an unordered set of transitions computed as follows:

   – If all conflicts between the output transitions of p are solved by mutual exclusion, or if the set of conflicting transitions of p is a singleton, then, the function returns the set of output transitions of p, i.e. $\texttt{output}(p)$ as defined above.

   – Otherwise, the function returns the set of output transitions of p connected through a $\texttt{test}$ or an $\texttt{inhib}$ arc, i.e. $\{t \mid \exists \omega \ s.t. \ pre(p,t) = (\omega, \texttt{test}) \lor pre(p,t) = (\omega, \texttt{inhib})\}$.

– $\texttt{cassoc}(map, id, x)$ where map is either an input port map or an output port map, id is an identifier, x is an expression or a name (i.e. a simple or indexed identifier). The $\texttt{cassoc}$ procedure adds an association of the form $(id(i), x)$ to the map structure. The index i is computed as follows based on the content of map:

1. looks up $id(j)$ with $max(j)$ in the formal parts of map

2. if no such j, adds $(id(0), x)$ in map

3. if such j, adds $(id(j+1), x)$ in map

Examples:

– $\texttt{cassoc}(\ \{(\texttt{s}(0), \texttt{true}), (\texttt{s}(1), \texttt{false})\}, \ \texttt{s}, \ \texttt{true})$ yields the resulting map $\{(\texttt{s}(0), \texttt{true}), (\texttt{s}(1), \texttt{false}), (\texttt{s}(2), \texttt{true})\}$.

– $\texttt{cassoc}(\ \{(\texttt{s}(0), \texttt{true}), (\texttt{s}(1), \texttt{false})\}, \ \texttt{a}, \ 3)$ yields the resulting map $\{(\texttt{s}(0), \texttt{true}), (\texttt{s}(1), \texttt{false}), (\texttt{a}(0), 3)\}$.

– $\texttt{get\_comp}(id_c, cstmt)$ where id_c is an identifier, and $cstmt \in \texttt{cs}$ is a $\mathcal{H}$-VHDL concurrent statement. The $\texttt{get\_comp}$ function looks up $cstmt$ for a component instantiation statement labelled with id_c as a component instance identifier, and returns the component instantiation statement when found. The $\texttt{get\_comp}$ function throws an error if no component instantiation statement with identifier id_c exists in $cstmt$, or if there exist multiple component instantiation statements with identifier id_c in $cstmt$.

– `put_comp(`id_c`,` *cistmt*`,` *cstmt*`)` where id_c is an identifier, *cistmt* is a component instantiation statement, and *cstmt* ∈ cs is a $\mathcal{H}$-VHDL concurrent statement. The `put_comp` procedure looks up in *cstmt* for a component instantiation statement with identifier id_c, and replaces the statement with *cistmt* in *cstmt*. If no CIS with identifier id_c exists in *cstmt*, then *cistmt* is directly composed with *cstmt* with the || operator. The `put_comp` procedure throws an error if multiple CIS with identifier id_c exist in *cstmt*.

– `actual(`*id*`,` *map*`)` where *id* is an identifier and *map* is a generic, an input port or an output port map. The `actual` function returns the actual part associated with the formal part *id* in *map*, i.e. returns *a* if (*id*, *a*) ∈ *map*. The function throws an error if *id* is not a formal part in *map*, or if there are multiple association with *id* as a formal part in *map*.

– `genid()`. The `genid` function returns a fresh and unique identifier. During the transformation, we appeal to it when a new internal signal, a new port or a new component instance must be declared or generated.

Algorithm 4 presents the `connect` procedure. This procedure takes an output port map *o*, an input port map *i*, a name *n* (i.e. a simple or indexed identifier), an identifier *id* and a $\mathcal{H}$-VHDL design *d* as parameters. It generates a Boolean internal signal id_s and adds it to the internal signal declaration list of design *d*. Then, the procedure adds the association between the *n* name and the internal signal id_s to the output port map *o*. Moreover, the procedure adds an association between a subelement of *id*, which index will be determined by the `cassoc` function, and the internal signal id_s to the input port map *i*. As a result, *n* is connected to a subelement of *id* through the Boolean internal signal id_s. Note that the name *n* must denote a signal of the Boolean type, and so must be the subelements of the composite signal denoted by *id*; otherwise, the output port map *o* and the input port map *i*, will not be well-typed at the end of the execution of the `connect` procedure.

Algorithm 4: `connect(`o, i, n, id, d`)`

1   $id_s \leftarrow$ `genid()`
2   $d.sigs \leftarrow d.sigs \cup \{(id_s, \texttt{boolean})\}$
3   $o \leftarrow o \cup \{(n, id_s)\}$
4   `cassoc(`i, id, id_s`)`

Notation 11. *When there is no ambiguity, id_p (resp. id_t) denotes the PCI (resp. TCI) identifier associated with a given place p (resp. transition t) through $\gamma(p) = id_p$ (resp. $\gamma(t) = id_t$), where γ is the binder returned by the HILECOP transformation function. Similarly, id_c (resp. id_a and id_f) denotes the input port (resp. output port) identifier associated with a given condition c (resp. action a and function f) through $\gamma(c) = id_c$.*

5.3.3   Generation of component instances and constant parts

The first step of the transformation generates the PCIs and TCIs, their generic map, and the constant part of their input port maps, in the behavior of the $\mathcal{H}$-VHDL design. At this moment of the transformation, places are bound to PCI identifiers, and transitions are bound to TCI identifiers in the γ binder. Also, the `marked` output port and the `fired` output port are connected to internal signals in the output port map of PCIs and TCIs. Algorithm 5 presents the content of the `generate_architecture` procedure that implements this first part of code generation. The `generate_architecture` procedure is decomposed in two procedures: the `generate_PCIs` and the `generate_TCIs` procedures.

Algorithm 5: `generate_architecture`(*sitpn*, d, γ, b)

---

1 `generate_PCIs`(*sitpn*, d, γ, b)
2 `generate_TCIs`(*sitpn*, d, γ)

The `generate_PCIs` procedure, presented in Algorithm 6, has four parameters: *sitpn* $\in$ *SITPN*, the input SITPN model; $d \in design$, the $\mathcal{H}$-VHDL design being generated; $\gamma \in WM(sitpn, d)$, the binder between *sitpn* and d; $b \in P \rightarrow \mathbb{N}$, the function assigning a maximal marking value to each place. The procedure iterates over the set of places of the *sitpn* parameter. For each place p in the set, the procedure produces a corresponding PCI id_p, and generates its generic map g_p, and its partially-built input and output port maps i_p and o_p. At the end of the procedure (Lines 24 to 26), a fresh and unique component identifier id_p is generated, and a new component instantiation statement, corresponding to the instantiation of the PCI id_p, is composed with the current behavior of design d. Finally, the γ binder receives a new couple corresponding to binding of place p to identifier id_p.

Algorithm 6: `generate_PCIs(`*sitpn, d, γ, b*`)`

---

1 foreach $p \in P$ **do**

2   **if** `input`$(p) = \emptyset$ *and* `output`$(p) = \emptyset$ **then** `err`$("p$ *is an isolated place*"$)$

3 $g_p \leftarrow \{(\texttt{mm}, b(p)), (\texttt{ian}, \begin{cases} 1 \text{ if } \texttt{input}(t) = \emptyset \\ |\texttt{input}(t)| \text{ otherwise} \end{cases}), (\texttt{oan}, \begin{cases} 1 \text{ if } \texttt{output}(t) = \emptyset \\ |\texttt{output}(t)| \text{ otherwise} \end{cases})\};$

      $i_p \leftarrow \emptyset; o_p \leftarrow \emptyset$

4   **if** `input`$(p) = \emptyset$ **then** $i_p \leftarrow i_p \cup \{(\texttt{iaw}(0), 0), (\texttt{itf}(0), \texttt{false})\}$

5   **else**

6 $i \leftarrow 0$

7   **foreach** $t \in \texttt{input}(p)$ **do**

8 $i_p \leftarrow i_p \cup \{(\texttt{iaw}(i), post(t, p))\}$

9 $i \leftarrow i + 1$

10   **if** `output`$(p) = \emptyset$ **then**

11 $i_p \leftarrow i_p \cup \{(\texttt{oaw}(0), 0), (\texttt{oat}(0), \texttt{basic}), (\texttt{otf}(0), \texttt{false})\}$

12 $o_p \leftarrow o_p \cup \{(\texttt{oav}, \texttt{open}), (\texttt{pauths}, \texttt{open}), (\texttt{rtt}, \texttt{open})\}$

13   **else**

14 $i \leftarrow 0$

15   **foreach** $t \in \texttt{output}_c(p) \cup \texttt{output}_{nc}(p)$ **do**

16 $(\omega, a) \leftarrow pre(p, t)$

17 $i_p \leftarrow i_p \cup \{(\texttt{oaw}(i), \omega), (\texttt{oat}(i), a)\}$

18 $i \leftarrow i + 1$

19   **if** `acts`$(p) = \emptyset$ **then** $o_p \leftarrow o_p \cup \{(\texttt{marked}, \texttt{open})\}$

20   **else**

21 $id_s \leftarrow \texttt{genid}()$

22 $d.sigs \leftarrow d.sigs \cup \{(id_s, \texttt{boolean})\}$

23 $o_p \leftarrow o_p \cup \{(\texttt{marked}, id_s)\}$

24 $id_p \leftarrow \texttt{genid}()$

25 $d.cs \leftarrow d.cs \parallel \texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p)$

26 $\gamma \leftarrow \gamma \cup \{(p, id_p)\}$

From Line 2 to Line 23, the procedure generates the generic map, the input port map, and the output port map of the PCI that implements place p. First, the procedure checks if the current place p is isolated, i.e. without input nor output transitions. An error, with an associated message, is raised with the `err` primitive if the test succeeds. The HILECOP transformation raises errors in the presence of an input SITPN model that does not meet the well-definition property (see Definition 28). One part of the well-definition property pertains to the absence of isolated place in the input model. Line 3 builds the variable g_p, and initializes the variables i_p and o_p, respectively holding the generic map, the input port map and the output port map of the PCI being generated. The generic map g_p holds three associations: the association between the `mm` constant to the maximal marking value returned by the b function for place p, and the association between the `ian` (resp. the `oan` constant) and a natural number that depends on the size of the set of input transitions (resp. output transitions) of place p.

Line 4 tests if the set of input transitions of p is empty. The size of the `iaw` and `itf` input ports, which are of the array type, is equal to the value of the `ian` constant. Thus, in the case where the `ian` constant is associated to 1 in the generic map g_p (i.e. the set of input transitions of p is empty), the `iaw` and `itf` input ports are composed of one subelement with index 0. At Line 4, the sole subelement of the `iaw` port is associated with 0, and the sole subelement of the `itf` port is associated with `false` in the input port map i_p. In the set of input transitions of p holds is not empty, each subelement of the `iaw` port is associated with the weight of the arc between place p and a given input transition t. Note that, in that case, the procedure does not deal with the connection of the `itf` port. As the set of input transitions of p is not empty, the connection of the `itf` port will be performed by the `generate_interconnections` described in Algorithm 10.

Line 10 tests if the set of output transitions of p is empty. The size of the `oaw`, `oat` and `otf` input ports, which are of the array type, is equal to the value of the `oan` constant. Thus, in the case where the `oan` constant is associated to 1 in the generic map g_p (i.e. the set of output transitions of p is empty), the `oaw`, `oat` and `otf` input ports are composed of one subelement with index 0. At Line 11, the sole subelement of the `oaw` port is associated with 0, the sole subelement of the `oat` port is associated with `basic`, and the sole subelement of the `otf` port is associated with `false` in the input port map i_p. Also, in the abscence of output transitions, the `oav`, `pauths` and `rtt` output ports are left unconnected, i.e. they are associated with the `open` keyword of output port map o_p.

If the set of output transitions of p is not empty, the `oan` constant is associated with the size of this set in the generic map g_p. Then, each subelement of the `oaw` (resp. the `oat`) port is associated with the weight (resp. the type) of the arc between place p and a given output transition t. Note that, in that case, the procedure does not handle the connection of the `otf` input port, nor the connection of the `oav`, `pauths` and `rtt` output ports. As the set of output transitions of p is not empty, these connections will be performed by the `generate_interconnections` described in Algorithm 10.

From Line 19 to Line 23, the `generate_PCIs` procedure connects the `marked` output port in the output port map o_p. If the place p is not associated with any action, the `marked` output port is left unconnected, i.e. connected to the `open` keyword. Otherwise, the `marked` output port is connected to a newly generated internal signal of the Boolean type. This generated signal joins the internal signal declaration list of design d. The connection between the `marked` output port and the internal signal will be used later, during the generation of the `action` process (see Section 5.3.5).

Figure 5.8 shows the generic, input port and output port map of the PCI id_{p_0} (cf. Figure 5.3) after the execution of the `generate_PCIs` procedure.

```
        ┌─────────────────────────────┐
      ▷ │ clk                         │
        │ rst              oav(0)     │
    1 ──│ im                          │
    1 ──│ iaw(0)           rtt(0)     │
basic ──│ oat(0)   id_p0              │
    1 ──│ oaw(0)          pauths(0)   │
        │ itf(0)                      │
        │ otf(0)                  m   │──▶
        └─────────────────────────────┘
```

(ian, 1), (oan, 1), (mm, $b(p_0)$)
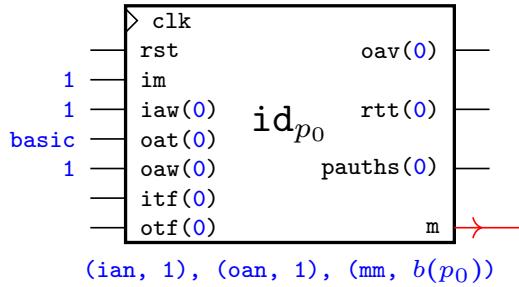
FIGURE 5.8: A graphical representation of the interface of the PCI id_{p_0} after the `generate_PCIs` procedure. The generic map associations appear in blue underneath the PCI. The indexes of composite ports appear in blue to stress the relation between the interface dimensioning and the generic constants. The m output port is connected to an internal signal represented by a red wire.

The `generate_TCIs` procedure, presented in Algorithm 7, iterates over the set of transitions T of the *sitpn* parameter. For each transition t in the set, the procedure produces a corresponding TCI id_t, and generates its generic map g_t, and its partially-built input and output port maps i_t and o_t. At the end of the procedure (Lines 11 to 13), a fresh and unique component identifier id_t is generated, and a new component instantiation statement, corresponding to the instantiation of the TCI id_t, is composed with the current behavior of design d. Finally, the γ binder receives a new couple corresponding to binding of transition t to identifier id_t.

Algorithm 7: `generate_TCIs`(*sitpn*, d, γ)

---

1 **foreach** $t \in T$ **do**

2   **if** $\text{input}(t) = \varnothing$ *and* $\text{output}(t) = \varnothing$ **then** $\text{err}("t \text{ is an isolated transition}")$

3    $g_t \leftarrow \{(\text{tt}, \text{get\_ttype}(t)), (\text{mtc}, \text{get\_mtc}(t)),$

4    $(\text{ian}, \begin{cases} 1 \text{ if } \text{input}(t) = \varnothing \\ |\text{input}(t)| \text{ otherwise} \end{cases}), (\text{cn}, \begin{cases} 1 \text{ if } \text{conds}(t) = \varnothing \\ |\text{conds}(t)| \text{ otherwise} \end{cases})\}$

5    $i_t \leftarrow \{(\text{A}, \begin{cases} 0 \text{ if } t \notin \text{dom}(I_s) \\ l(I_s(t)) \text{ otherwise} \end{cases}), (\text{B}, \begin{cases} 0 \text{ if } t \notin \text{dom}(I_s) \lor u(I_s(t)) = \infty \\ u(I_s(t)) \text{ otherwise} \end{cases})\}$

6    $id_s \leftarrow \text{genid}()$

7    $d.sigs \leftarrow d.sigs \cup \{(id_s, boolean)\}$

8    $o_t \leftarrow \{(\text{fired}, id_s)\}$

9   **if** $\text{input}(t) = \varnothing$ **then** $i_t \leftarrow i_t \cup \{(\text{iav}(0), \text{true}), (\text{pauths}(0), \text{true}), (\text{rt}(0), id_s)\}$

10   **if** $\text{conds}(t) = \varnothing$ **then** $i_t \leftarrow i_t \cup \{(\text{ic}(0), \text{true})\}$

11    $id_t \leftarrow \text{genid}()$

12    $d.cs \leftarrow d.cs \,||\, \text{comp}(id_t, \text{transition}, g_t, i_t, o_t)$

13    $\gamma \leftarrow \gamma \cup \{(t, id_t)\}$

At Line 2, the procedure checks if transition t is isolated, and raises an error accordingly. Lines 4 to 8 initialize the variables g_t, i_t and o_t, respectively holding the generic map, the input port map and the output port map of the TCI being-generated. The generic map g_t takes

four associations: the association between the `tt` constant and the result of the function call `get_ttype(t)`, the association between the `mtc` constant and the result of the function call `get_-mtc(t)`, the association between `ian` and the size of the set of input places of t, and the association between `cn` and the size of the set of conditions associated with t. The `get_ttype` function returns the type of transition t, i.e. either `NOT_TEMPORAL`, `TEMPORAL_A_A`, `TEMPORAL_A_B` or `TEMPORAL_A_INFINITE`, based on the form of the time interval associated with t. Algorithm 8 describes the `get_ttype` function.

Algorithm 8: `get_ttype(t)`

---

1 **if** $t \notin \mathrm{dom}(I_s)$ **then return** *NOT_TEMPORAL*
2 **else if** $I_s(t) = [a, a]$ **then return** *TEMPORAL_A_A*
3 **else if** $I_s(t) = [a, b]$ **then return** *TEMPORAL_A_B*
4 **else if** $I_s(t) = [a, \infty]$ **then return** *TEMPORAL_A_INFINITE*

The `get_mtc` function determines the maximal value for the time counter of t based on the form of the time interval associated with transition t. Algorithm 9 describes the `get_mtc` function.

Algorithm 9: `get_mtc(t)`

---

1 **if** $t \notin \mathrm{dom}(I_s)$ **then return** 1
2 **else if** $I_s(t) = [a, b]$ **then return** b
3 **else if** $I_s(t) = [a, \infty]$ **then return** a

In the `generate_TCIs` procedure, Line 5 sets the value of the `A` and `B` input ports while initializing the input port map i_t. The `A` port is associated with 0 if the transition t is not a time transition (i.e. t has no associated time interval, it is not in the domain of function I_s); otherwise, the `A` port is associated with the lower bound of the time interval of t. The `B` input port is associated with 0 if transition t is not a time transition or if its time interval has an infinite upper bound; otherwise, the `B` port is associated with the upper bound of the time interval of t. From Lines 6 to 8, the `generate_TCIs` procedure connects the `fired` output port to a newly generated internal signal in the output port map o_p. This internal signal will then be connected to the input port map of PCIs during the interconnection phase of the transformation (see Section 5.3.4).

Line 9 checks if the set of input places of t is empty. If the test succeeds, the `ian` constant is associated with 1 in the generic map g_t. The size of the `iav`, `pauths` and `rt` input ports, which are of the array type, is equal to the value of the `ian` constant. Thus, in the case where the set of input places of t is empty, the `iav`, `pauths` and `rt` input ports are composed of one subelement with index 0. At Line 9, the sole subelements of the `iav` and the `pauths` ports are associated with `true`, and the sole subelement of the `rt` port is associated with the signal identifier id_s. Remember that the `fired` output port has been previously connected to the internal signal id_s in the output port map o_t. Thus, the `fired` output port is connected to the subelement of the `rt` input port with index 0 through the id_s signal. This connection is mandatory to reset the value of the `s_time_counter` signal (which is an internal signal of the `transition` design) in the absence of input places.

Line 10 checks if the set of conditions attached to t is empty. The size of the ic input port, which is of the array type, is equal to the value of the cn constant. Thus, in the case where the set of conditions attached to t is empty, the ic input port is composed of one subelement with index 0. Then, the sole subelement of the ic port is associated with true in the input port map i_t. If the set of conditions attached to t is not empty, then the generate_conds procedure, presented in Algorithm 12, will handle the connection of the subelements of the ic input port.

Figure 5.9 shows the generic, input port and output port map of the TCI id_{t_0} (cf. Figure 5.3) after the execution of the generate_TCIs procedure.
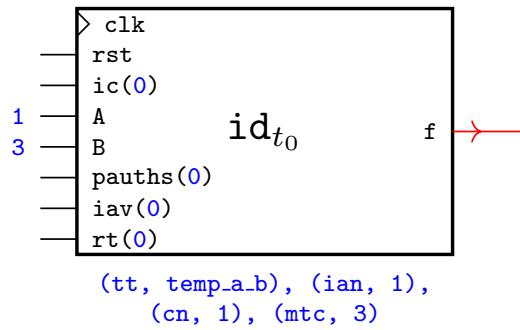
FIGURE 5.9: A graphical representation of the interface of the TCI id_{t_0} after the generate_TCIs procedure. The generic map associations appear in blue underneath the TCI. The indexes of composite ports appear in blue to stress the relation between the interface dimensioning and the generic constants. The f output port is connected to an internal signal represented by a red wire.

5.3.4 Interconnection of the place and transition component instances

After the generation of PCIs and TCIs, and of all constant associations in their generic and input port maps, the next step of the transformation performs the interconnections between the interfaces of PCIs and TCIs. The generate_interconnections procedure, presented in Algorithm 10, produces these interconnections.

Algorithm 10: `generate_interconnections(`$sitpn, d, \gamma$`)`

---

1 foreach $p \in P$ **do**

2 $\quad$ `comp(`id_p`, place,` g_p, i_p, o_p`)` $\leftarrow$ `get_comp(`$\gamma(p), d.cs$`)`

3 $\quad i \leftarrow 0$

4 $\quad$ **foreach** $t \in$ `input(`p`)` **do**

5 $\quad\quad$ `comp(`id_t`, transition,` g_t, i_t, o_t`)` $\leftarrow$ `get_comp(`$\gamma(t), d.cs$`)`

6 $\quad\quad$ $i_p \leftarrow i_p \cup \{($`itf(`$i$`)`, `actual(fired,` o_t`))\}$

7 $\quad\quad$ $i \leftarrow i + 1$

8 $\quad i \leftarrow 0$

9 $\quad$ **foreach** $t \in$ `output`$_c$`(`p`)` **do**

10 $\quad\quad$ `comp(`id_t`, transition,` g_t, i_t, o_t`)` $\leftarrow$ `get_comp(`$\gamma(t), d.cs$`)`

11 $\quad\quad$ $i_p \leftarrow i_p \cup \{($`otf(`$i$`)`, `actual(fired,` o_t`))\}$

12 $\quad\quad$ `connect(`o_p, i_t`, oav(`i`), iav,` d`)`

13 $\quad\quad$ `connect(`o_p, i_t`, rtt(`i`), rt,` d`)`

14 $\quad\quad$ `connect(`o_p, i_t`, pauths(`i`), pauths,` d`)`

15 $\quad\quad$ `put_comp(`id_t`, comp(`id_t`, transition,` g_t, i_t, o_t`),` $d.cs$`)`

16 $\quad\quad$ $i \leftarrow i + 1$

17 $\quad$ **foreach** $t \in$ `output`$_{nc}$`(`p`)` **do**

18 $\quad\quad$ `comp(`id_t`, transition,` g_t, i_t, o_t`)` $\leftarrow$ `get_comp(`$\gamma(t), d.cs$`)`

19 $\quad\quad$ $i_p \leftarrow i_p \cup \{($`otf(`$i$`)`, `actual(fired,` o_t`))\}$

20 $\quad\quad$ `connect(`o_p, i_t`, oav(`i`), iav,` d`)`

21 $\quad\quad$ `connect(`o_p, i_t`, rtt(`i`), rt,` d`)`

22 $\quad\quad$ $id_s \leftarrow$ `genid()`

23 $\quad\quad$ $d.sigs \leftarrow d.sigs \cup (id_s,$ `boolean`$)$

24 $\quad\quad$ $o_p \leftarrow o_p \cup \{($`pauths(`$i$`)`$, id_s)\}$

25 $\quad\quad$ `cassoc(`i_t`, pauths, true)`

26 $\quad\quad$ `put_comp(`id_t`, comp(`id_t`, transition,` g_t, i_t, o_t`),` $d.cs$`)`

27 $\quad\quad$ $i \leftarrow i + 1$

28 $\quad$ `put_comp(`id_p`, comp(`id_p`, place,` g_p, i_p, o_p`),` $d.cs$`)`

The `generate_interconnections` procedure iterates over the set of places of the *sitpn* parameter. For each place p, the procedure generates the interconnections between the PCI id_p and the TCIs that implement the input and output transitions of p; we will refer to them as the input and output TCIs of PCI id_p.

At Line 2, the `get_comp` function returns the PCI associated with the identifier $\gamma(p)$ (i.e. the PCI identifier associated with place p in γ) by looking up the behavior of the design d. At this step, we assume that all PCIs and TCIs, and all bindings pertaining to places and transitions in the γ binder, have been previously generated by the `generate_architecture` procedure. Otherwise, the `get_comp` function raises an error if it is not able to find the PCI id_p in the behavior of design d.

Then, from Line 3 to Line 27, the procedure modifies the input and output port map of PCI id_p and the input port map of its input and output TCIs. Finally, Line 28 replaces the old PCI id_p by the modified one in the behavior of design d.

From Line 3 to Line 7, the procedure iterates over the input transitions of place p. Note that the iteration is performed in the same order as the iteration performed by the **foreach** loop at Line 7 of the `generate_PCIs` procedure; this is mandatory to preserve a consistency between the index i and the connection to a given transition (see Remark 8). For each input transition t of p, the corresponding TCI id_t is retrieved from the behavior of design d. Then, the internal signal associated with the `fired` output port in the output port map of TCI id_t is retrieved (i.e. `actual(fired, `o_t`)`), and the signal is associated with the subelement of the `itf` input port with index i. We know that the `generate_TCIs` function has generated the association between the `fired` output port and an internal signal in the output port map of all TCIs. Thus, the `actual` function never raises an error.

Remark 8 (Connections consistency). *In the behavior of the `place` design, some processes access to the subelements of composite ports through the use of indices. For instance, the `input_tokens_sum` process (see Appendix A) increments a local variable i in range 0 to `input_arcs_number` $-$ 1 in a for loop. The process tests the value of the `itf` port's subelement with index i. If the test succeeds, the process adds the value of the `iaw` port's subelement with index i to the local variable `v_internal_input_token_sum`. Thus, the subelement with index i of the `itf` and `iaw` ports must refer to the connection to the same transition. Otherwise, the process does not compute a correct input tokens sum. Figure 5.10 illustrates the correct connection of the `itf` and `iaw` ports in the input port map of PCI id_p w.r.t. to the connection between transitions t_a, t_b, t_c and place p.*
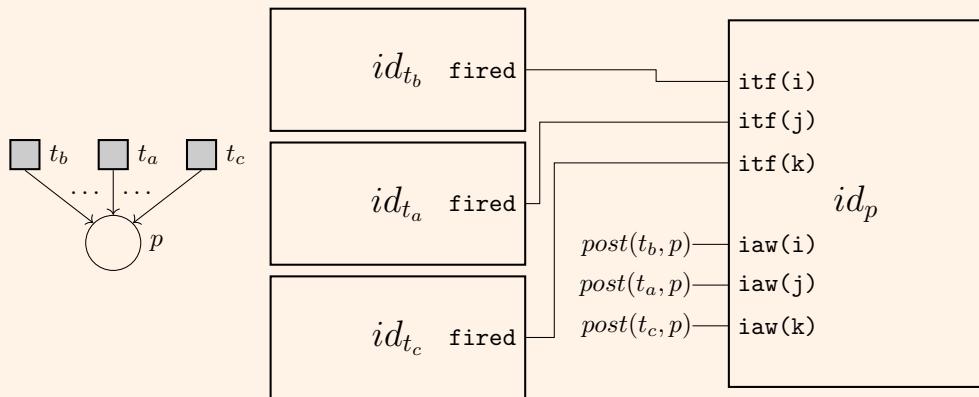
FIGURE 5.10: An example of correct connections between the PCI id_p and TCIs id_{t_a}, id_{t_b} and id_{t_c}. On the left, the input SITPN model showing the connections of the transitions t_a, t_b and t_c to the place p. The dots indicate that the place p possibly has other input transitions. On the right, the TCIs and the PCI generated by the transformation. In the input port map of PCI id_p, the subelements of the `itf` input port are connected to the `fired` port of TCIs; the subelements of the `iaw` port are connected to constant values, i.e. the weight of the arcs between place p and the input transitions of p.

It is the part of the HILECOP transformation function to ensure the consistency of the connections of the subelements in the input and output port maps of PCIs. Because the `transition` design

> *does not hold the information pertaining to the arc connections, the input and output port maps of TCIs are not subject to such a constraint. The fact that a **foreach** loop always iterates in the same order over the elements of a set ensures the consistency of the connections.*

From Line 9 to Line 16, the procedure connects the PCI id_p to the TCIs implementing the conflicting output transitions of place p. For each conflicting output transition t of p, the corresponding TCI id_t is retrieved from the behavior of design d. The function call `actual(fired, `o_t`)` returns the internal signal associated with the `fired` output port in the output port map of TCI id_t. This internal signal is then connected to the subelement of the `otf` input port with index i in the input port map of PCI id_p. At Line 12, the `connect` function generates an internal signal id and adds it to the internal signal declaration list of design d. Then, the function associates the subelement `oav(`i`)` (i.e. the subelement of the `oav` input port with index i) with the internal signal id in the output port map o_p, and it associates one subelement of the `iav` input port to the internal signal id in the input port map i_t. The `connect` function operates similarly on the `rtt` output port and the `rt` input port at Line 13, and on the `pauths` input port and the `pauths` output port at Line 14. Finally, at Line 15, the old TCI id_t is replaced by the modified one in the behavior of design d.

From Line 18 to Line 26, the procedure connects the TCIs implementing to the output transitions of p that are not in conflict. Note that the variable i is not reset between the two **foreach** loops to preserve the continuity of indices. For each non-conflicting output transition t of p, the corresponding TCI id_t is retrieved from the behavior of design d. Then, the interconnections between PCI id_p and TCI id_t are similar to the ones that have been performed for the conflicting transitions of p. The difference lies in the connection of the `pauths` ports. Between the PCI id_p and its *non-conflicting* TCIs, the `pauths` are not connected together; this to reflect the independence of non-conflicting output transitions regarding the priority authorizations. Instead, the subelement of the `pauths` output port with index i is connected to a newly generated internal signal id_s in the output port map o_p (Line 22 to Line 24); the internal signal id_s is not connected to anything, and it will be removed by the (industrial) compiler at the time of the synthesis. Also, one subelement of the `pauths` input port is associated with `true` in the input port map i_t (Line 25); this connection represents the fact that, since the transition t is not a conflicting transition of place p, then, transition t always has the authorization to be fired, given that it is firable.

Figure 5.11 shows the interconnections between the PCI id_{p_0} and the TCI id_{t_0} (cf. Figure 5.3) after the execution of the `generate_interconnections` procedure.
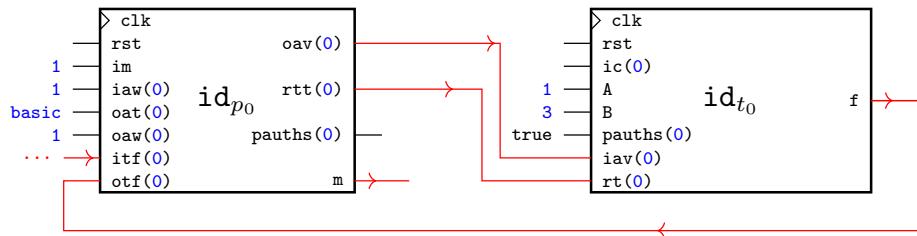
FIGURE 5.11: A graphical representation of the interconnections of the PCI id_{p_0} and the TCI id_{t_0} after the execution of `generate_interconnections` procedure.

5.3.5 Generation of ports, the `action` and the `function` process

The last part of the transformation pertains to the generation of the input and output ports of the top-level $\mathcal{H}$-VHDL design. The input ports implement the conditions declared in the input SITPN model. Each input port is associated with a condition through the γ binder. This binding is built during the transformation. The output ports of the $\mathcal{H}$-VHDL design implement the action and function of the input SITPN. Each output port is associated with an action or a function through the γ binder. During the simulation of a $\mathcal{H}$-VHDL design, the value of an output port represent the activation/execution status of the associated action/function. Algorithm 11 presents the `generate_ports` procedure. This procedure calls three procedures, namely: the `generate_condition_ports` procedure, responsible for the generation and the connection of input ports implementing conditions; the `generate_action_ports` procedure, responsible for the generation of output ports implementing actions, and for the generation of the `action` process; the `generate_function_ports` procedure, responsible for the generation of output ports implementing functions, and for the generation of the `function` process. These three procedures are detailed in Algorithms 12, 13 and 14.

Algorithm 11: `generate_ports`(*sitpn, d, γ*)

---

1   `generate_condition_ports`(*sitpn, d, γ*)
2   `generate_action_ports`(*sitpn, d, γ*)
3   `generate_function_ports`(*sitpn, d, γ*)

Algorithm 12 describes the `generate_condition_ports` procedure.

Algorithm 12: `generate_condition_ports`(*sitpn, d, γ*)

---

1   **foreach** $c \in \mathcal{C}$ **do**
2      $id_c \leftarrow \texttt{genid}()$
3      $d.ports \leftarrow d.ports \cup \{(\texttt{in}, id_c, \texttt{boolean})\}$
4      $\gamma \leftarrow \gamma \cup \{(c, id_c)\}$
5   **foreach** $t \in \texttt{trs}(c)$ **do**
6          $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \leftarrow \texttt{get\_comp}(\gamma(t), d.cs)$
7   **if** $\mathbf{C}(t, c) = 1$ **then** $\texttt{cassoc}(i_t, \texttt{ic}, id_c)$
8   **else if** $\mathbf{C}(t, c) = -1$ **then** $\texttt{cassoc}(i_t, \texttt{ic}, \texttt{not } id_c)$
9          $\texttt{put\_comp}(id_t, \texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t), d.cs)$

The `generate_condition_ports` procedure iterates over the set of conditions of the *sitpn* parameter. For each condition of the set, the `generate_condition_ports` procedure produces a corresponding input port identifier id_c, and adds an input port declaration entry in the port declaration list of design *d*. The declared input port is of the Boolean type. Also, a binding between condition *c* and identifier id_c is added to γ. Then, the procedure performs the connection between the input port id_c and the `ic` input port present in the input interface of TCIs. The `ic` input port is an array composed of Boolean subelements. Indeed, as multiple conditions can be attached to a given transition, a given TCI is possibly connected to multiple input ports

implementing conditions through its `ic` port. At Line 5, the **foreach** loop iterates over the set of transitions attached to condition c. For each such transition t, the corresponding TCI id_t is retrieved from the behavior of design d. Then, depending on the relation that exists between condition c and transition t, an association between id_c and one subelement of the `ic` input port is added to the input port map i_t. At the end of the loop, the old TCI id_t is replaced by a new TCI, with an updated input port map, in the behavior of design d.

Algorithm 13 describes the `generate_action_ports` procedure.

Algorithm 13: `generate_action_ports`($sitpn, d, \gamma$)

---

1 $rstss \leftarrow$ `null`
2 $fss \leftarrow$ `null`

3 **foreach** $a \in \mathcal{A}$ **do**
4 $\quad$ $id_a \leftarrow$ `genid()`
5 $\quad$ $d.ports \leftarrow d.ports \cup \{(\texttt{out}, id_a, \texttt{boolean})\}$
6 $\quad$ $\gamma \leftarrow \gamma \cup \{(a, id_a)\}$
7 $\quad$ $e_{id_a} \leftarrow$ `false`

8 $\quad$ **foreach** $p \in$ `pls`(a) **do**
9 $\quad\quad$ `comp`$(id_p, \texttt{place}, g_p, i_p, o_p) \leftarrow$ `get_comp`$(\gamma(p), d.cs)$
10 $\quad\quad$ $id_s \leftarrow$ `actual(marked`, $o_p)$
11 $\quad\quad$ $e_{id_a} \leftarrow id_s$ `or` e_{id_a}

12 $\quad$ $rstss \leftarrow rstss;$ $id_a \Leftarrow$ `false`
13 $\quad$ $fss \leftarrow fss;$ $id_a \Leftarrow e_{id_a}$

14 $d.cs \leftarrow d.cs \,||\,$ `process(action`, $\{\texttt{clk}\}, \varnothing, \texttt{rst}$ $(rstss)$ $(\texttt{falling}\ fss))$

The `generate_action_ports` procedure does two things. First, it generates an output port for each action of the input SITPN; second, it builds the `action` process that is responsible for the assignment of the value of *action* ports depending on the value of the `marked` output ports of PCIs. The `action` process is a synchronous process; its statement body is composed of a single `rst` block. A `rst` block is composed of two blocks of sequential statements; the first block is executed only during an initialization phase, otherwise, the second block is executed. Here, the second block corresponds to a `falling` block, i.e. a block that is only executed during a falling edge phase. Thus, the `generate_action_ports` procedure builds two blocks of sequential statements: the first one, hold in the *rstss* variable, corresponds to the first part of the `rst` block (i.e. the one executed during the initialization phase); the second one, hold in the *fss* variable, corresponds to the second part of the `rst` block, i.e. a falling edge block. The first two lines of the procedure initialize the *rstss* and *fss* with the `null` sequential statement. Then, in the absence of actions defined in the input SITPN, the statement body of the `action` process is composed of `null` statements; the execution of `null` statements has no effect on the state of design during a simulation. At Line 3, the procedure iterates over the set of actions of the *sitpn* parameter. For each action a in the set, an output port identifier id_a is generated, an output port

declaration entry is added to the port declaration list of design d, the binding between action a and identifier id_a joins the γ binder.

An action is activated at given state if one of its attached place is marked, i.e. its marking is greater than zero. An output port identifier that implements the activation status of a given action is assigned in the falling block of the `action` process. The expression assigned to the output port id_a corresponds to the `or` sum between each `marked` port of the PCIs implementing the places attached to the action a. From Line 7 to Line 13, the `generate_action_ports` procedure builds this `or` sum expression. For each place p associated with the action a, the corresponding PCI id_p is retrieved from the behavior of design d. The internal signal id_s associated with the `marked` port is looked up in the output port map of PCI id_p. Then, the signal identifier id_s is composed with the expression e_{id_a} with the `or` operator. At the end of the loop started at Line 3, the procedure adds a new signal assignment statement to the $rstss$ and to the fss variables by composition with the ; operator. In the $rstss$ variable, i.e. in the part of the `action` process executed during an initialization phase, the id_a output port is assigned to `false`. In the fss variable, i.e. the part of the `action` process executed during a falling edge phase, the id_a output port is assigned to the previously built `or` sum expression e_{id_a}. The last line of the procedure builds and adds the `action` process to the behavior of design d. The `action` process is a synchronous process, thus, it declares the `clk` signal in its sensitivity list. The `action` process has an empty set of local variables. Finally, its statement body is composed of a `rst` block with $rstss$ as a first block, and a `falling` edge block wrapping fss as a second block.

Algorithm 14 describes the `generate_function_ports` procedure.

Algorithm 14: `generate_function_ports(`*sitpn, d,* γ`)`

---

1   $rstss \leftarrow$ `null`

2   $rss \leftarrow$ `null`

3   **foreach** $f \in \mathcal{F}$ **do**

4      $id_f \leftarrow$ `genid()`

5      $d.ports \leftarrow d.ports \cup \{(\text{out}, id_f, \text{boolean})\}$

6      $\gamma \leftarrow \gamma \cup \{(f, id_f)\}$

7      $e_{id_f} \leftarrow$ `false`

8   **foreach** $t \in$ `trs(`f`)` **do**

9   `comp(`id_t`, transition`$, g_t, i_t, o_t) \leftarrow$ `get_comp(`$\gamma(t), d.cs$`)`

10         $id_s \leftarrow$ `actual(fired`$, o_t)$

11         $e_{id_f} \leftarrow id_s$ `or` e_{id_f}

12      $rstss \leftarrow rstss;\ id_f \Leftarrow$ `false`

13      $rss \leftarrow rss;\ id_f \Leftarrow e_{id_f}$

14   $d.cs \leftarrow d.cs \parallel$ `process(function,` $\{\text{clk}\}, \varnothing,$ `rst` $(rstss)$ `(rising` $rss))$

The `generate_function_ports` procedure does two things. First, it generates an output port for each function of the input SITPN; second, it builds the `function` process that is responsible for the assignment of the value of *function* ports depending on the value of the `fired`

output ports of PCIs. Similarly to the `action` process, the `function` process is a synchronous process with a statement body composed of a single `rst` block. The second part of the `rst` block is a `rising` block, i.e. a block that is only executed during a rising edge phase. Thus, the `generate_function_ports` procedure builds two blocks of sequential statements: the first one, hold in the *rstss* variable, corresponds to the first part of the `rst` block (i.e. the one executed during the initialization phase); the second one, hold in the *rss* variable, corresponds to the second part of the `rst` block, i.e. a rising edge block. The first two lines of the procedure initialize the *rstss* and *rss* with the `null` sequential statement. At Line 3, the procedure iterates over the set of functions of the *sitpn* parameter. For each function f in the set, an output port identifier id_f is generated, an output port declaration entry is added to the port declaration list of design d, the binding between function f and identifier id_f joins the γ binder.

A function is executed at given state if one of its attached transition is fired. An output port identifier that implements the execution status of a given function is assigned in the rising block of the `function` process. The expression assigned to the output port id_f corresponds to the `or` sum between each `fired` port of the TCIs implementing the transitions attached to the function f. From Line 7 to Line 13, the `generate_function_ports` procedure builds this `or` sum expression. For each transition t associated with the function f, the corresponding TCI id_t is retrieved from the behavior of design d. The internal signal id_s associated with the `fired` port is looked up in the output port map of TCI id_t. Then, the signal identifier id_s is composed with the expression e_{id_f} with the `or` operator. At the end of the loop started at Line 3, the procedure adds a new signal assignment statement to the *rstss* and to the *rss* variables by composition with the `;` operator. In the *rstss* variable, i.e. in the part of the `function` process executed during an initialization phase, the id_f output port is assigned to `false`. In the *rss* variable, i.e. the part of the `function` process executed during a rising edge phase, the id_f output port is assigned to the previously built `or` sum expression e_{id_f}. The last line of the procedure builds and adds the `function` process to the behavior of design d. The `function` process is a synchronous process, thus, it declares the `clk` signal in its sensitivity list. The `function` process has an empty set of local variables. Finally, its statement body is composed of a `rst` block with *rstss* as a first block, and a `rising` edge block wrapping *rss*.

5.4   Coq implementation of the HILECOP model-to-text transformation

This section presents the implementation of the HILECOP model-to-text transformation with the Coq proof assistant. The full implementation is available under the `sitpn2hvhdl` folder of the following Git repository: https://github.com/viampietro/ver-hilecop

Listing 5.1 gives the Coq implementation of the `sitpn_to_hvhdl` function presented in an imperative pseudo-code version in Algorithm 3.

```
1  Definition sitpn_to_hvhdl (sitpn : Sitpn)
2     (decpr : forall x y : T sitpn, {pr x y} + {~pr x y})
3     (id_e id_a : ident) (b : P sitpn → nat) :
4     (design * Sitpn2HVhdlMap sitpn) + string :=
```

```
5    RedV
6      (( do _ ← generate_sitpn_infos sitpn decpr;
7        do _ ← generate_architecture sitpn b;
8        do _ ← generate_ports sitpn;
9        do _ ← generate_comp_insts sitpn;
10       generate_design_and_binder id_e id_a)
11     ( InitS2HState sitpn Petri.ffid)).
```

LISTING 5.1: The Coq implementation of the `sitpn_to_hvhdl` function presented in Algorithm 3.

In Listing 5.1, the `sitpn_to_hvhdl` function has five parameters: `sitpn`, the input SITPN model; `decpr`, a proof that the `pr` relation (i.e. the implementation of the firing priority relation) is decidable over the set of transitions of `sitpn` (i.e. `T sitpn`); id_e and id_a, the entity and architecture identifiers for the generated $\mathcal{H}$-VHDL design; the `b` function that maps the places of the `sitpn` parameter to a maximal marking value, i.e. a natural number. The `sitpn_to_hvhdl` function returns a couple composed of the generated $\mathcal{H}$-VHDL design, of type `design`, and the generated γ binder, of type `Sitpn2HVhdlMap sitpn`; or, the `sitpn_to_hvhdl` function returns a `string` corresponding to an error message.

In the body of the `sitpn_to_hvhdl` function, the `RedV` is a notation that reduces a monadic function call to a value. Our implementation of the HILECOP transformation function relies on the state-and-error monad [110]. Each function that implements a part of the transformation function takes a *compile-time* state as a parameter, and returns either a value and a new compile-time state, or an error message. The `bind` construct of the state-and-error monad permits to pipeline multiple function calls, and, combined with the `do` notation, it permits us to write functional programs in the style of imperative languages. The sequence defined in the body of the `sitpn_to_hvhdl` function gives an example of what can be achieved with the combination of the state-and-error monad and the `do` notation. This sequence constitutes a single monadic function that takes a state of the `Sitpn2HVhdlState` type (see Listing 5.2) as input, and yields a value with a new state, or an error message. Here, the `RedV` notation retrieves only the value returned by the application of the monadic function to the parameter (`InitS2HState sitpn Petri.ffid`) (i.e. the initial compile-time state), or it retrieves the error message.

In the `do` sequence of Listing 5.1, the four first function calls do not return values that are relevant; thus, we use the underscore notation to notify that we are not interested in the returned values. Indeed, the `generate_sitpn_infos`, `generate_architecture`, `generate_ports` and `generate_comp_insts` functions directly modify the compile-time state without returning a value. They are the functional implementation of the procedures described in the previous section.

Now, let us present the content of the compile-time state. As said above, the compile-time state is carried from function to function and modified all along the transformation. Listing 5.2 gives the implementation of the compile-time state structure.

```
1  Record Sitpn2HVhdlState (sitpn : Sitpn) : Type :=
2    MkS2HState {
3      lofPs : list (P sitpn);
4      lofTs : list (T sitpn);
```

```
 5        lofCs : list (C sitpn);
 6        lofAs : list (A sitpn);
 7        lofFs : list (F sitpn);
 8        nextid : ident;
 9        sitpninfos : SitpnInfos sitpn;
10        iports : list pdecl;
11        oports : list pdecl;
12        arch : Architecture sitpn;
13        beh : cs;
14        γ : Sitpn2HVhdlMap sitpn;
15
16    }.
```

LISTING 5.2:   The compile-time state structure defined as the Coq
`Sitpn2HVhdlState` record type.

The compile-time state structure is implemented by the `Sitpn2HVhdlState` record type. This type depends on a given `sitpn` passed as a parameter. It is composed of eleven fields. The first five fields (Line 3 to 7) are the list versions of the finite sets of places, transitions, conditions, actions and functions of the `sitpn` parameter. These fields are filled at the very beginning of the transformation by the `generate_sitpn_infos` function, and are convenient to write functions in the context of dependent types. The `nextid` field (Line 8) permits us to generate fresh and unique identifiers all along the transformation. The `sitpinfos` field (Line 9) is an instance of the `SitpnInfos` type that depends on the `sitpn` parameter. The `sitpninfos` field is filled up by the `generate_sitpn_infos` function. It is a convenient way to represent the information associated with the places, transitions, conditions, actions and functions of the `sitpn` parameter. The `iports` (resp. `oports`) field, at Line 10 (resp. at Line 11), gathers the list of input (resp. output) port declarations of the generated $\mathcal{H}$-VHDL design. The `arch` field (Line 12) is an intermediary representation of the behavior of the generated $\mathcal{H}$-VHDL design. This representation is easier to modify and to handle than a $\mathcal{H}$-VHDL concurrent statement. The `beh` field (Line 13) is the behavior of the generated $\mathcal{H}$-VHDL design; it is an instance of the `cs` type, i.e. the type of concurrent statements defined in the abstract syntax of $\mathcal{H}$-VHDL. The γ field (Line 14) is the SITPN-to-$\mathcal{H}$-VHDL binder generated alongside the $\mathcal{H}$-VHDL design, and returned at the end of the transformation.

At the beginning of the transformation, an initial compile-time state is built with the `InitS2HState` function. The `InitS2HState` function gives an initial value to the fields of the state structure; mostly, the fields are initialized with empty lists, and the `beh` field is initialized with the `null` statement. The `InitS2HState` function takes an `Sitpn` instance and an identifier as inputs. The identifier parameter represents the initial value of the `nextid` field. In Listing 5.1, the second parameter of the `InitS2HState` function is `Petri.ffid`. It corresponds to the *first fresh* identifier that the transformation can use to produce a $\mathcal{H}$-VHDL design that respects the uniqueness of identifiers.

Let us now present the functions composing the `do` sequence of the `sitpn_to_hvhdl` function, and how they modify the compile-time state to produce the final $\mathcal{H}$-VHDL design and the γ binder.

5.4.1 The `generate_sitpn_infos` **function**

Listing 5.3 presents a part of the `generate_sitpn_infos`. The part that is let aside, represented by little dots, pertains to the creation of the dependently-typed lists constituting the first fields of the compile-time state structure (Line 3 to 7 in Listing 5.2).

```
1   Definition generate_sitpn_infos
2           (sitpn : Sitpn)
3           (decpr : forall x y : T sitpn, {pr x y} + {~pr x y}) :
4     Mon (Sitpn2HVhdlState sitpn) unit :=
5     ...
6     do _ ← check_wd_sitpn sitpn decpr;
7     do _ ← generate_trans_infos sitpn;
8     do _ ← generate_place_infos sitpn decpr;
9     do _ ← generate_cond_infos sitpn;
10    do _ ← generate_action_infos sitpn;
11    generate_fun_infos sitpn.
```

LISTING 5.3: A part of the `generate_sitpn_infos` function.

The `generate_sitpn_infos` function takes an `Sitpn` instance and a proof of decidability for the `pr` relation as parameters. It returns a value of type `Mon (Sitpn2HVhdlState sitpn) unit`. A value of this type can either be a couple (*state*, *value*), where *state* is of type (`Sitpn2HVhdlState sitpn`) and *value* is of type `unit`, or an error message. The `unit` type as only one possible value `tt`. The `unit` type is used here to represent a function that modifies the compile-time state without returning a value.

The aim of the `generate_sitpn_infos` function is to fill the `sitpninfos` field of the compile-time state; the `sitpninfos` field is an instance of the `SitpnInfos` record type. Listing 5.4 presents the definition of the `SitpnInfos` record type, along with the definition of the `PlaceInfo` and `TransInfo` record types.

```
1   Record PlaceInfo (sitpn : Sitpn) : Type :=
2     MkPlaceInfo { tinputs : list (T sitpn);
3                 tconflict : list (T sitpn);
4                 toutputs : list (T sitpn) }.
5
6   Record TransInfo (sitpn : Sitpn) : Type :=
7     MkTransInfo { pinputs : list (P sitpn); conds : list (C sitpn) }.
8
9   Record SitpnInfos (sitpn : Sitpn) : Type :=
10    MkSitpnInfos {
11       pinfos : list (P sitpn * PlaceInfo);
12       tinfos : list (T sitpn * TransInfo);
13       cinfos : list (C sitpn * list (T sitpn));
14       ainfos : list (A sitpn * list (P sitpn));
15       finfos : list (F sitpn * list (T sitpn));
16      }.
```

LISTING 5.4: The `PlaceInfo`, `TransInfo` and `SitpnInfos` record types.

The `PlaceInfo` record type is composed of three lists that represent the input transitions, `tinputs`, the conflicting output transitions, `tconflict`, and the non-conflicting output transitions, `toutputs`, of a place. In the `SitpnInfos` structure, the `pinfos` field maps the places of the `sitpn` parameter to their respective information, i.e. an instance of the `PlaceInfo` type. This mapping is built by the `generate_place_infos` function called in the body of `generate_-sitpn_infos` function. While building an instance of the `PlaceInfo` type for a given place p, the `generate_place_infos` function computes the list of output transitions of p that are in conflict (in the manner of the $output_c$ function described in Section 5.3.2). First, it computes the list of output transitions that are linked to the place p through a `basic` arc; then, the function checks if all conflicts between the transitions of this list are solved by means of mutual exclusion. If it is the case, the `tconflict` field is left empty, and all transitions of the list join the `toutputs` list. Otherwise, the function tries to establish a strict total order over the transitions of the list, by decreasing level of priority. If no such order can be established, the function raises an error; otherwise, the `tconflict` field is filled with the ordered list. This process never fails if the input `sitpn` parameter is indeed well-defined (cf. Definition 28).

The `TransInfo` record type is composed of two lists that represent the input places, `pinputs`, and the output places, `poutputs`, of a transition. In the `SitpnInfos` structure, the `tinfos` field maps the transitions of the `sitpn` parameter to their respective information, i.e. an instance of the `TransInfo` type. This mapping is built by the `generate_trans_infos` function called in the body of `generate_sitpn_infos` function.

In the `SitpnInfos` structure, the `cinfos` (resp. `ainfos` and `finfos`) field maps the conditions (resp. actions and functions) of the `sitpn` parameter to the list of transitions (resp. places and transitions) they are attached to. This mapping is built by the `generate_cond_infos` (resp. `generate_action_infos` and `generate_fun_infos`) function called in the body of `generate_-sitpn_infos` function.

At the beginning of the `generate_sitpn_infos` function, the `check_wd_sitpn` function partly checks the well-definition of the `sitpn` parameter. Precisely, it checks that the set of places and transitions of the `sitpn` parameter are not empty, and that the priority relation is a strict order, i.e. transitive and reflexive, over the set of transitions. The other parts of the well-definition checking are performed later during the transformation. For instance, the `generate_place_infos` function checks that, for each group of transitions in conflict, the conflicts are either solved by means of mutual exclusion or that the priority relation is a strict total order over this group. It also checks that there are no isolated places in the input `sitpn` parameter, etc.

5.4.2   The `generate_architecture` **function**

Listing 5.5 presents the `generate_architecture` function. The `generate_architecture` function implements the `generate_architecture` and the `generate_interconnections` procedures detailed in Algorithms 5 and 10. The composition of the `generate_place_map` and the `generate_trans_map` functions implements `generate_architecture` procedure of Algorithm 5. Precisely, the `generate_place_map` function implements the `generate_PCIs` procedure presented in Algorithm 6, and the `generate_trans_map` function implements the `generate_TCIs` procedure presented in Algorithm 7.

```
1   Definition generate_architecture (sitpn : Sitpn) (b : P sitpn → nat) :
```

```
2    Mon (Sitpn2HVhdlState sitpn) unit :=
3    do _ ← generate_place_map sitpn b;
4    do _ ← generate_trans_map sitpn;
5    generate_interconnections.
```

LISTING 5.5: The `generate_architecture` function that implements the `generate_architecture` procedure of Algorithm 5.

The `generate_architecture` function takes an `Sitpn` instance and the `b` function as inputs, and modifies the compile-time state. The `generate_architecture` function fills the `arch` field of the compile-time state; the `arch` field is an instance of the `Architecture` record type. Listing 5.4 presents the definition of the `Architecture` record type, along with the definition of the `InputMap`, `OutputMap` and `HComponent` type aliases.

```
1    Definition InputMap := list (ident ∗ (expr + list expr)).
2    Definition OutputMap := list (ident ∗ ((option name) + list name)).
3    Definition HComponent := (genmap ∗ InputMap ∗ OutputMap).
4
5    Record Architecture (sitpn : Sitpn) := MkArch {
6      sigs  : list sdecl;
7      plmap : list (P sitpn ∗ HComponent);
8      trmap : list (T sitpn ∗ HComponent);
9      fmap  : list (F sitpn ∗ list expr);
10     amap  : list (A sitpn ∗ list expr) }.
```

LISTING 5.6: The `Architecture` record type, and the `InputMap`, `OutputMap` and `HComponent` subsidiary types.

The `HComponent` type is an intermediate representation of an $\mathcal{H}$-VHDL component instantiation statement. This type has been devised to ease the construction of PCIs and TCIs, and of their generic, input port and output port maps all along the transformation. The `HComponent` type is a triplet composed of a generic map as defined in the $\mathcal{H}$-VHDL abstract syntax, an instance of the `InputMap` type, and an instance of the `OutputMap` type. The `InputMap` type maps an input port identifier to either a simple expression or to a list of expressions, where the `expr` type is the type of expressions defined in the $\mathcal{H}$-VHDL abstract syntax. In an `InputMap` instance, an input port identifier of a scalar type (i.e. Boolean or constrained natural) is mapped to a simple expression, whereas an input port identifier of the array type is mapped to a list of expressions. Each expression of the list represents the actual part associated with one subelement of the input port. Similarly to the `InputMap` type, the `OutputMap` type maps an output port identifier to either an option to a signal (the `None` value representing the connection to the open keyword) name, or to a list of signal names. In the definition of the `OutputMap` type, the `name` type represents the type of simple identifiers or indexed identifiers defined in the $\mathcal{H}$-VHDL abstract syntax.

The `Architecture` record type is an intermediary representation of the behavioral and declarative part of an $\mathcal{H}$-VHDL design's architecture. The `sigs` field of the `Architecture` type represents the internal signal declaration list constituting the declarative part of an $\mathcal{H}$-VHDL design's architecture. The transformation adds a new signal declaration entry to the `sigs` field

every time an internal signal must be generated, for example, during the generation of inter-connections between PCIs and TCIs. The `plmap` (resp. the `trmap`) field maps the places (resp. transitions) of the `sitpn` parameter to their corresponding PCI (resp. TCI) implemented in an intermediate format, i.e. an instance of the `HComponent` type. The `fmap` field of the `Architecture` type maps the functions of the `sitpn` parameter to a list of expressions. For a given function f, the associated list of expressions corresponds to the list of internal signals associated with the `fired` port of the TCIs implementing the transitions of the $\text{trs}(f)$ set (i.e. the set of transitions associated with function f). The `fmap` field is filled by the `generate_ports` function described in Listing 5.7. The `amap` field is the twin of the `fmap` field but on the side of the actions of the `sitpn` parameter. Thus, in the `amap` field, the list of expressions associated with an action a corresponds to the list of internal signals connected to the `marked` port of the PCIs implementing the places of a.

In the body of the `generate_architecture` function, the `generate_place_map` function implements the `generate_PCIs` procedure described in Algorithm 6. For each place of the `sitpn` parameter, the `generate_place_map` function builds an instance of the `HComponent` type, and adds an association between place and `HComponent` instance in the `plmap` field. The `generate_place_map` function fills the generic, input port and output port map of the `HComponent` instances as described in the `generate_PCIs` procedure. Following the `generate_place_map` function, the `generate_trans_map` function implements the `generate_TCIs` procedure described in Algorithm 7. For each transition of the `sitpn` parameter, the `generate_trans_map` function builds an instance of the `HComponent` type, and adds an association between transition and `HComponent` instance in the `trmap` field. The `generate_trans_map` function fills the generic, input port and output port map of the `HComponent` instances as described in the `generate_TCIs` procedure. Finally, the `generate_interconnections` function modifies the input and output port maps of the `HComponent` instances in the `plmap` and `trmap` fields, and thus, implements the interconnections described in the `generate_interconnections` procedure of Algorithm 10.

5.4.3 The `generate_ports` function

Listing 5.7 presents the `generate_ports` function called in the body of the `sitpn_to_hvhdl` function (see Listing 5.1). The `generate_ports` function implements the `generate_ports` procedure described in Algorithm 11. The `generate_ports` function calls three functions: the `generate_action_ports_and_ps` function that implements the `generate_action_ports` procedure of Algorithm 13, the `generate_fun_ports_and_ps` function that implements the `generate_function_ports` procedure of Algorithm 14, and the `generate_and_connect_cond_ports` that implements the `generate_condition_ports` procedure of Algorithm 12.

```
1  Definition generate_ports (sitpn : Sitpn) : Mon (Sitpn2HVhdlState sitpn) unit :=
2    do _ ← generate_action_ports_and_ps;
3    do _ ← generate_fun_ports_and_ps;
4    generate_and_connect_cond_ports.
```

LISTING 5.7: The `generate_ports` function implementing the `generate_ports` procedure presented in Algorithm 11.

For every action of the `sitpn` parameter, the `generate_action_ports_and_ps` function adds a port declaration entry to the `oports` field of the compile-time state, and adds a binding between action and output port identifier in the γ field. It also builds the `action` process as described in the `generate_action_ports` procedure, and adds the process to the `beh` field of the compile-time state. The `generate_fun_ports_and_ps` does the same for the functions of the `sitpn` parameter, and similarly builds the `function` process and adds it to the `beh` field. The `generate_and_connect_cond_ports` function add a port declaration entry for every condition of the `sitpn` parameter to the `iports` field of the compile-time state. Then, it modifies the input port map of `HComponent` instances in the `trmap` of the compile-time state's `arch` field. The modifications pertain to the connection of input ports to the `ic` input port of TCIs, as described in the `generate_condition_ports` procedure (see Algorithm 12).

5.4.4 The `generate_comp_insts` and `generate_design_and_binder` functions

At the end of the `sitpn_to_hvhdl` function (see Listing 5.1), the `generate_comp_insts` function transforms the `HComponent` instances, associated with places and transitions in the compile-time state's `arch` field, into real component instantiation statements as defined in the $\mathcal{H}$-VHDL abstract syntax. Then, the `generate_design_and_binder` builds up the final $\mathcal{H}$-VHDL design and the γ binder, and returns the couple. Listing 5.8 presents the `generate_comp_insts` function and the `generate_design_and_binder` function.

```
1  Definition generate_comp_insts (sitpn : Sitpn) : Mon (Sitpn2HVhdlstate sitpn) unit :=
2    do _ ← generate_place_comp_insts sitpn; generate_trans_comp_insts sitpn.
3
4  Definition generate_design_and_binder (sitpn : Sitpn) (id_e id_a : ident) :
5      Mon (Sitpn2HVhdlstate sitpn) (design * Sitpn2HVhdlMap sitpn) :=
6    do s ← Get;
7    Ret (( design_ id_e id_a [] ((iports s) ++ (oports s)) (sigs (arch s)) (beh s)), (γ s)).
```

LISTING 5.8: The `generate_comp_insts` and the `generate_design_and_binder` function.

The `generate_comp_insts` function is needed because we are using an intermediary representation for the component instantiation statements. Even though this representation is convenient to manipulate data during the different phases of the transformation, it also implies an extra generation step to complete the generation of the $\mathcal{H}$-VHDL design and the γ binder. The `generate_comp_insts` function calls the `generate_place_comp_insts` and the `generate_trans_comp_insts` functions. These two functions being similar in all points, except for the type of their inputs, we are only presenting the `generate_place_comp_insts` function here. The `generate_place_comp_insts` function calls the `generate_place_comp_inst` function for each place defined in the set of places of the `sitpn` parameter. Listing 5.9 presents the code the `generate_place_comp_inst` function.

```
1  Definition generate_place_comp_inst (sitpn : Sitpn) (p : P sitpn) :
2      Mon (Sitpn2HVhdlstate sitpn) unit :=
3
4    do id_p ← get_nextid;
```

```
5        do _      ← bind_place p id_p;
6        do pcomp  ← get_pcomp p;
7        do pci    ← HComponent_to_comp_inst id_p place_entid pcomp;
8        add_cs pci.
```

LISTING 5.9: The `generate_place_comp_inst` function.

The `generate_place_comp_inst` function generates a fresh and unique PCI identifier by appealing to the `get_nextid` function. The `get_nextid` function returns and increments the current value of the `nextid` field, defined in the compile-time state. Then, the `bind_place` function adds a binding between the place `p` and the identifier id_p in the γ field of the compile-time state. The `get_pcomp` function looks up the `plmap` field (defined under the `arch` field of the compile-time state) and returns the `HComponent` instance associated with the place `p`, i.e. `pcomp`. The `HComponent_to_comp_inst` function translates the `HComponent` instance `pcomp` into a PCI with the identifier id_p. Finally, the `add_cs` function composes the returned PCI with the current $\mathcal{H}$-VHDL design behavior, hold in the `beh` field of the compile-time state.

The transformation of a `HComponent` instance into a PCI implies the translation of the input and output port map, which are instances of the `InputMap` and `OutputMap` types, into their equivalent representation in $\mathcal{H}$-VHDL abstract syntax. The translation especially concerns the association between a port identifier of the array type and a list of expressions, or names. For instance, let us consider an instance of `InputMap` that is an intermediary representation of the input port map of a PCI id_p. In this `InputMap` instance, the `itf` port, which is a composite input port of the `place` design, is associated with the list $[id_a, id_b, id_c]$. Then, based on the previous association, the `HComponent_to_comp_inst` function generates the following associations is the concrete input port map of PCI id_p: $(\texttt{rt}(0), id_a)$, $(\texttt{rt}(1), id_b)$ and $(\texttt{rt}(2), id_c)$.

Getting back to Listing 5.8, the `generate_design_and_binder` function retrieves the current compile-time state `s` with the `Get` function. Then, based on the value of the different fields of the compile-time state, the function builds an $\mathcal{H}$-VHDL design and returns it along with the γ binder. The $\mathcal{H}$-VHDL design receives the id_e and id_a identifiers, passed as inputs, as the design's entity and architecture identifiers. The generic constant declaration list of the $\mathcal{H}$-VHDL design is empty, i.e. it receives the empty list value. The port declaration list of the $\mathcal{H}$-VHDL design is built by concatenating the content of the `iports` and `oports` fields defined in state `s`. The internal signal declaration list is filled by the `sigs` field, defined under the `arch` field of state `s`. Finally, the `beh` field receives the behavior of the $\mathcal{H}$-VHDL design.

5.5   Conclusion

The purpose of this chapter was to give to the reader a complete understanding of the HILECOP model-to-text transformation function, and of what makes it a very specific transformation case. We first gave an informal presentation of the transformation function with a high-level view of the transformation principles. Then, we presented our literature review pertaining to transformation functions in the context of formal verification, with a particular focus on the expression and the implementation of transformation functions. Two points, drawn out from the literature review, are of particular interest. First, the review showed that it is important,

during a transformation, to keep the binding between the elements of the source representation and their corresponding versions in the target representation. This binding is the base of the comparison of the run-time state of the source and target representation that permits to express the theorem of semantic preservation. Second, if the distance between the source and the target representation is too important, it is easier, while aiming at proving a semantic preservation property, to split the transformation into multiple simple transformation steps. Then, to each transformation step will correspond an intermediary representation, and a theorem of semantic preservation will be laid out and proved for each one of them. In the case of the HILECOP model-to-text transformation, even though the transformation has a lot of tricky aspects pertaining to particular cases of input models, there is no need to split the transformation into simple steps with intermediary representations. Even though the verification task is quite close, the HILECOP transformation is quite different from the certified GPL or the HDL compilers presented in the literature review. Indeed, the source representation is an input model not a programming language. Moreover, due to the interconnection of the component instances generated by the transformation function, devising a transformation algorithm that generates modular and independently executable code is impossible. As everything is connected, one has to reason over the entire transformation process to get the overall behavior of the generated $\mathcal{H}$-VHDL design. This is also one of the main difference between the HILECOP transformation and compilers for programming languages. Despite all that, the transformation algorithm, presented in this chapter, gets as close as possible to a modular expression of the HILECOP transformation.

Chapter 6

Proving semantic preservation in HILECOP

In this chapter, we present our semantic preservation theorem along with its informal "paper" proof. The written proof is about a hundred-page long. Therefore, we will only present here the "high-level" theorems and lemmas involved in the demonstration, and some points of our proof strategy. The full proof is available to the reader in Appendix D. The structure of this chapter is as follows: in Section 6.1, we present our review of the literature related to the proof of semantic preservation theorems for transformation functions; in Section 6.2, we detail our state similarity relation, i.e. the semantic relation between an SITPN and its $\mathcal{H}$-VHDL translation; in Section 6.3, we draw out our behavior preservation theorem; in Section 6.4, we detail a particular point of the proof related to the SITPN firing process, and leverage the opportunity to demonstrate some recurring points of our proof process; also, we show how this point of the proof has led to a bug detection in the code of the $\mathcal{H}$-VHDL transition design; in Section 6.5, we present some points of the mechanization of the proof with the Coq proof assistant.

6.1 Proofs of semantic preservation in the literature

In this section, we present a review of the literature about the verification of transformation functions. A transformation function is understood here as any kind of mapping from a source representation to a target representation, where the source and target representations have a behavior of their own (i.e. they are executable). Here, we will focus on verification techniques based on the proof of semantic preservation theorems, with extra interest when the proofs are mechanized within the framework of a proof assistant. We are interested in how to prove that transformation functions are semantic preserving. Especially, we are interested in the expression of semantic preservation theorems and in seeking usual proof strategies, or patterns. By proof strategy, or proof pattern, we mean the description of the way to perform a proof. For instance, if the authors use induction to prove their theorem of semantic preservation, the choice of the element on which the induction will be performed is part of the proof strategy.

The goal is to draw our inspiration from the literature and to see how far the correspondence holds between our specific case of transformation, and other cases of transformations. The

material used for the literature review is divided into three categories. Each category covers a specific case of transformation function; the three categories are:

– Compilers for generic programming languages

– Compilers for hardware description languages

– Model-to-model and model-to-text transformations

In [72], X.Leroy presents the two points of major importance to express semantic preservation theorems for GPL (Generic Programming Language) compilers, and more generally to get the meaning of semantic preservation.

The first point is to clearly state how things are compared between the source and the target programs. It is to describe the runtime state of the source and the target and draw a correspondence between the two. This is expressed through a state comparison relation.

The second point is to relate the execution of the source program to the execution of the target program through a *simulation* diagram, equivalently named *bisimulation* or *commuting* diagram. Figure 6.1, excerpt from [72], shows the different kinds of simulation diagrams possibly relating two programs together.
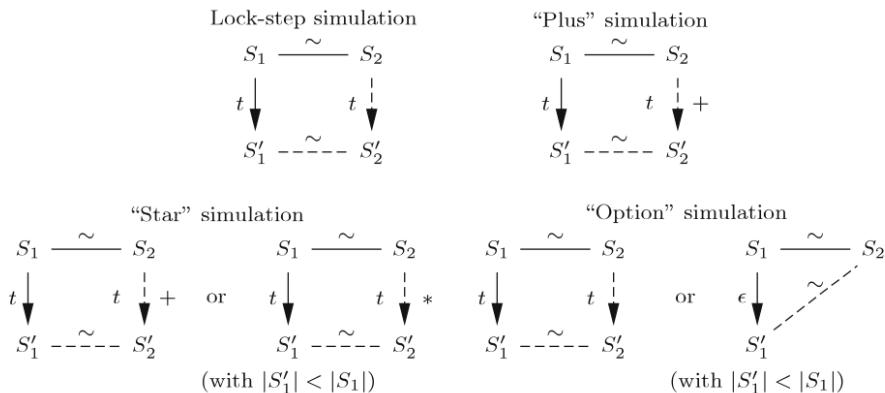
FIGURE 6.1: Simulation diagrams relating the execution of a source program to the execution of a target program; S_1 and S_2 are the initial states of the source and the target program, and S_1' and S_2' are the final states of the source and target program, i.e. the states resulting of the execution of the two programs. The $\sim$ symbol represents the state comparison relation between the source and target language states. The arrows represent the execution relation for the source and target program producing the observable execution trace t.

Choosing an adequate simulation diagram to express a semantic preservation theorem depends on the kind of possible behaviors that a given program can exhibit. In the case of GPL programs, X.Leroy lists three kinds of possible behaviors: either the program execution succeeds and returns a value, or the program execution fails and returns an error, or the program execution diverges. In the case where the source program execution succeeds, a theorem of semantic preservation takes the general form of Definition 36.

> **Definition 36** (General behavior preservation theorem). *Consider a source programming language L_1 and a target programming language L_2, and a source program $P_1 \in L_1$ compiled into a target program $P_2 \in L_2$ by compiler* comp $\in L_1 \rightarrow L_2$. *Consider an initial state S_1 for program P_1 and an initial state S_2 for program P_2 such that S_1 and S_2 are similar states w.r.t. to a given state comparison relation established between L_1 and L_2. Then, compiler* comp *is semantic preserving if it verifies the following property:*
> *If the execution of P_1 leads from state S_1 to final state S_1', then there exists a final state S_2' resulting of the execution of program P_2 from state S_2 such that S_1' and S_2' are similar w.r.t. the state comparison relation.*

Compiler verification aims at proving the kind of theorem stated above.

Now that we have clarified the meaning of semantic preservation for GPL compilers, we state that this definition of semantic preservation holds also for a more general case of transformation from a source representation to a target representation. The only condition to be able to verify that a transformation is semantic preserving is that the source and target representations must have an execution semantics (i.e, the instances of the source and target representations must be executable).

For each article used in the literature review and presenting a specific case of transformation, the following questions have been asked:

– What are the similarities/differences between source and target representations? May they be programs of GPLs, or models of a given model formalism.

– How is defined the runtime state for the source and target representations?

– How is expressed the state comparison relation?

– How is expressed the semantic preservation theorem?

– What is the employed proof strategy?

6.1.1 Compilers for generic programming languages

Taking the CompCert compiler as an example, the compilation pass from Clight programs to Cminor programs is described in [13, 72]. Clight is a subset of the C language, and Cminor is a low-level imperative language. The two languages are endowed with both a big-step and small-step operational semantics. Here, the execution state of the source and target languages are memory models for the most part, but are also completed with representations of global variable storage, registers, etc. The memory model consists of block references; each block has a lower and an upper bound. To access data, one has to specify the block reference along with the size of the accessed data (i.e, the data type) and the offset from the start of the block reference (i.e, where to begin the data reading). Regarding the proof of semantic preservation, the most difficult point is to relate the memory state of the source program to the memory state of the target program. To do so, the authors define a *memory injection* relation, which binds the

values of source and target together. They also establish a relation to compare execution environments, i.e, the environments holding the declaration of functions, global variables...The proof of semantic preservation is built incrementally. First, the authors prove a correctness lemma for the Clight expressions: if a Clight expression a evaluates to value v, then the translated Cminor expression $\lfloor a \rfloor$ evaluates to value v (the Clight and Cminor languages have the same set of values). Then, they prove a similar lemma for Clight statements, and finally for the entire Clight program. The proof strategy is to reason by induction over the evaluation relation of the Clight programs and perform case analysis on the translation function.

The pattern to compiler verification for GPLs is more or less the same as presented above. In the case of compilers for imperative languages [72, 103], or compilers for functional languages [30, 105], compiler verification proceeds as follows:

1. Establish a relationship between the memory models of the source and target languages, and between the global execution environments.

2. Prove correctness lemmas starting from simple constructs, and building up incrementally to consider entire programs.

3. Reason by induction over the evaluation relation of the source language, and the translation function.

Relating memory models is more difficult when the gap between the source and target languages is important (for instance, the translation of Cminor programs into RTL (Register Transfer Language, which is close to assembly languages) programs in [72]). As a consequence, the complexity of the memory model comparison relation increases.

6.1.2 Compilers for hardware description languages

In the case of HDL (Hardware Description Language) compilers, proving semantic preservation is very similar to the case of GPL compilers. Of course, the difference lies in the semantics of HDL languages and the description of execution states. The semantics of HDLs is intrinsically related to the notion of execution over time, or over multiple clock cycles; we are dealing with reactive systems. Therefore, the semantic preservation theorems are formulated w.r.t. the synchronous or the time-related semantics of the considered languages.

In [18, 22], the source language is a subset of the BlueSpec specification language for hardware synthesis, and the target language is an RTL representation of the circuit. The runtime state of the source and target programs are basically a mapping between registers to values. In [18], the execution state also holds a log of the read and write operations of the input program, and this log is compared to the log of the RTL representation. The semantic preservation theorem takes the general form of Definition 36, however, the final states refer to the states of source and target programs at the end of a clock cycle. Thus, the semantic preservation theorem states that starting from equal register stores after the execution of a source program and its RTL (Register Transfer Level[1]) circuit after one clock cycle leads to equal register stores.

---

[1]The acronymn RTL is used both in the world of microelectronic and computer science. In computer science, it means Register Transfer Language and refers to a language which level is close to assembly languages. In microelectronic, it means Register Transfer Level and is a method to give a high-level representation of a circuit.

In [19], the source language is a subset of Lustre and the target language is an imperative language called Obc. A Lustre program is composed of nodes; each node treats a set of input streams and publishes output streams after the computation of its statement body. In its statement body, a Lustre node possibly refers to instances of other nodes. In the compilation process, each Lustre node is translated into an Obc class. An Obc class holds a vector of variables composing its internal memory and a vector of other Obc class instances. The authors define a data flow semantics for the Lustre language; the rule instances of the semantics describe how output streams are computed based on input streams. On the side of the Obc language, the semantics define a function *step* that computes the execution of the Obc classes over one clock cycle. To prove the semantic preservation theorem, the state comparison relation binds the values of input and output streams on one side to the values of variables and Obc class instances on the other side. The semantic preservation theorem is as follows: if a Lustre node yields the output stream *o* from an input stream *i*, then the iterative execution of the *step* function for the corresponding Obc class incrementally builds the output stream *o* given the values of the input stream *i*. The proof is done by induction over the clock step count, and by induction over the evaluation relation for the Lustre statements composing the body of nodes.

In [76], the HDL compiler translates Verilog modules into netlists. The execution state of Verilog module holds the value of the variables declared in the module. The execution state of a netlist circuit holds the value of the registers declared in the circuit. Therefore, the state comparison relation, used to state the semantic preservation theorem, binds the values of variables on one side to the values of registers on the other side. The semantics of Verilog is quite similar to the one of VHDL; a set of processes composing a module are executed w.r.t. the simulation semantics of the language, i.e, composed of synchronous and combinational execution steps. The authors give a big-step operational semantics to netlists by defining an interpreter that runs a netlist over n clock cycles. The semantic preservation theorem is as follows: Assuming that a module is transformed into a circuit, and that some well-formation hypotheses hold on the module, if the module executes without error, and yields a final state *venv*, then there exists a final state *cenv* yielded by the execution of the circuit over n clock cycles s.t. *venv* and *cenv* are similar according to the relation *verilog_netlist_rel*. Here, the *verilog_netlist_rel* is the state comparison relation, which relates variables to registers.

In [115], the compiler transforms programs of the synchronous language SIGNAL into Synchronous Clock Guarded Actions programs (S-CGA programs). A SIGNAL program describes a set of processes; each process holds a set of equations describing the relation between signals. The equations can be synchronous equations (referring to a clock) or combinational ones. An S-CGA program defines a set of actions to be applied to some variables when some conditions (the guards) are met. The SIGNAL (resp. the S-CGA) language has been endowed with a trace semantics describing the computation of signal values (resp. variable values) over time. The authors describe a function to translate the traces of SIGNAL and S-CGA programs into a common trace model. Thus, the semantic preservation theorem is stated by comparing two traces of execution defined through the same model. The proof of the semantic preservation theorem is built incrementally. For each statement of a SIGNAL process, the authors exhibit a lemma proving that the trace resulting from the execution of the statement is equivalent to the trace resulting of the execution of the corresponding guarded actions (obtained through the compilation). The proof is fully mechanized within the Coq proof assistant.

In [56], the authors verify a methodology to design hardware models with SystemC models. SystemC models describe hardware systems with modules; a module is a C++ class with ports, data members and methods. The methodology describes a transformation from SystemC models to Abstract State Machine (ASM) thus enabling to model-check the hardware models. ASMs are described in the language AsmL; in AsmL, an ASM is implemented by a class with data members and methods. A denotational (fixpoint) semantics for SystemC models is defined along with a denotational semantics for AsmL. The semantics is another variant of simulation cycle, similar to all other synchronous languages. There are two phases: evaluate and update and the gap between the two is called a delta-delay. The execution state of a SystemC model is divided into a signal store, mapping signal to value, and a variable store, mapping variable to value. The execution state of an AsmL class is only composed of a variable store. The theorem of semantic preservation states that, after translation, a SystemC model has the same *observational* behavior than its corresponding AsmL class. What is compared between a SystemC model and its corresponding AsmL class through their observational behavior is the activity of the processes of the first one and the activity of the methods of the second one. Processes and methods must be active at the same delta cycles. Therefore, what is compared here are not the values that the execution states hold, but rather the activity of the source and target programs.

6.1.3   Model transformations

Regarding model transformations, a lot of articles consider semantic preservation as the preservation of structural properties in the transformed model [7, 25, 77].

Still, there are many cases where the source model and the target one have both an execution semantics. In these cases, the authors are interested in proving that the transformation is semantic preserving by showing that the computation of the source model and the target model follow a commuting diagram (see Figure 6.1).

In [34] and [114], the authors are interested in giving a translational semantics to a given model having itself a reference execution semantics. In [34], the source models are called xSpem models; they describe a set of *activities* that exchange resources and hold an internal state. The target models are PNs. Both xSpem models and PNs have a state transition semantics. The state comparison is performed by checking the correspondence between each current status of the activities describe in an xSpem model and the marking of the PN. Then, the authors prove a bisimulation theorem, illustrated in Figure 6.2.
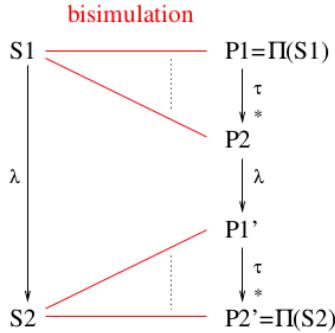
FIGURE 6.2: Bisimulation diagram relating an xSpem model execution and a Petri net execution

In Figure 6.2, on the right side of the diagram, i.e, the Petri net side, one can see that a Petri net possibly performs many internal actions (represented the arrow $\xrightarrow{\tau}{}^{*}$) before and after executing the computation step that is of interest for the proof (i.e, action λ). The proof is performed by reasoning by induction on the structure of the xSpem models, and then by reasoning of the state transition semantics of xSpem models and PNs.

In [114], the authors describe a transformation from a model of the AADL formalism (Architecture Analysis and Design Language) to a particular kind of Abstract State Machine (ASM) called Timed Abstract State Machines (TASM). To verify that the transformation is semantic preserving, the authors define the semantics of AADL models and TASMs through Timed Transition Systems (TTSs). Thus, the execution state of an AADL model is the execution state of the corresponding TTS, and the same holds for a TASM. Comparing the state of two TTSs is easier than comparing the state of two different models, thus having two different definitions. Then, the authors prove a strong bisimulation theorem to verify that the transformation is semantic preserving. The whole proof is mechanized within the Coq proof assistant.

In [50], the authors describe a transformation from LLVM-labelled Petri nets to LLVM programs, where LLVM is low-level assembly language. Precisely, the generated LLVM program implements the state space of the source Petri net (i.e, the graph of reachable markings). The authors want to verify if an LLVM program truly implements the PN state space, i.e. if each marking present in the PN state space can be reached by running a specific $fire_t$ function on the generated LLVM program. The state of an LLVM program is defined by a memory model composed of a heap and a stack. The marking of an LLVM-labelled PN is defined in such a manner that the correspondence with the LLVM program memory model is straightforward. The PN model has classical firing semantics, and LLVM programs follow a small-step operational semantics. The semantic preservation theorem states that for all transition t being fired, leading from marking M to marking M', then applying running the $fire_t$ function over the generated LLVM program at state LM (such that LM implements marking M) leads to a new state LM', such that LM' implements marking M'. To prove this theorem, the authors proceed by induction on the number of places of the input Petri net.

6.1.4   Discussions on transformations and proof strategies

In this thesis, we are interested in the verification of a semantic preservation property for a given transformation function. To achieve this kind of proof task, the way to proceed is quite similar, at least in the three cases of transformation presented above (i.e, GPL compilation, HDL compilation, and model transformations). Even though the source and target languages or models are different from one case of transformation to the other, however, semantic preservation theorems carry the same structure, i.e. the one presented in Definition 36. The state comparison relation and the choice of the commuting diagram (i.e. how much computational steps of the target representation correspond to one computational step of the source representation) are the two angular stones of the process.

One can notice that when verifying the transformation of HDL programs, the semantic preservation theorems are expressed in terms of a time-related computational step. It can either be a clock cycle or another kind of time step. The state equivalence checking is made at the end of this time-related computational step. This differs from the expression of behavior preservation theorems for GPLs, where a computational step is not related to time, but rather expresses the one-time computation of programs.

Concerning proof strategies, in the case of programming languages, proving the semantic preservation theorems is systematically done by induction over the semantics relations of the source and target languages, and by reasoning on the translation function. The semantics relations are themselves defined by following the inductive structure of the language ASTs. In the case of model transformations, when the source model makes it possible, the proofs are performed similarly by applying inductive reasoning over the structure of the input model. This enables compositional reasoning, i.e: to split the difficulty of proving the semantic preservation theorem into simpler lemmas about the execution of simpler programs or simple model structures. Based on these observations, we will now present the relation that allows us to compare the runtime state of a given SITPN model with the runtime state of an $\mathcal{H}$-VHDL design. This state similarity relation will then permit us to express our semantic preservation theorem.

6.2   The state similarity relation

Before presenting our behavior preservation theorem, we must clarify the meaning of semantic preservation between an SITPN and an $\mathcal{H}$-VHDL design. To do so, we must define:

1. What does semantic similarity mean between an SITPN state and a $\mathcal{H}$-VHDL state?

2. When, in the course of the execution of an SITPN and an $\mathcal{H}$-VHDL design, must this semantic similarity hold?

We must relate the elements that constitute the execution state of an SITPN to the elements that constitute the execution state of a $\mathcal{H}$-VHDL design. An SITPN state is an abstract structure relating the places, transitions, actions, functions and conditions of a given SITPN to the values of certain domains (see Section 3.2.2). An $\mathcal{H}$-VHDL design state is composed of a signal store mapping signals to values, and of a component store mapping component instances to their

own internal states (which are themselves design states). Thanks to the binder function γ (cf. Definition 35) generated alongside the transformation from an SITPN to an $\mathcal{H}$-VHDL design, we are able to relate the elements of the SITPN state structure to the component instance states and signal values of the $\mathcal{H}$-VHDL design state. The γ binder generated by the transformation is a bijective function. Thus, the state similarity relation, depending on a γ binder and expressing a semantic match between an SITPN state and an $\mathcal{H}$-VHDL design, is defined as follows:

> **Definition 37** (General state similarity). *For a given $sitpn \in SITPN$, an $\mathcal{H}$-VHDL design $d \in design$, an elaborated design $\Delta \in ElDesign$, and a binder $\gamma \in WM(sitpn, d)$, an SITPN state $s \in S(sitpn)$ and a design state $\sigma \in \Sigma$ are similar, written $\gamma \vdash s \sim \sigma$ if*
>
> 1. $\forall p \in P, id_p \in Comps(\Delta) \ s.t. \ \gamma(p) = id_p, \ s.M(p) = \sigma(id_p)(\texttt{s\_marking})$.
>
> 2. $\forall t \in T_i, id_t \in Comps(\Delta) \ s.t. \ \gamma(t) = id_t,$
> $\big(u(I_s(t)) = \infty \wedge s.I(t) \leq l(I_s(t)) \Rightarrow s.I(t) = \sigma(id_t)(\texttt{s\_time\_counter}) \big)$
> $\wedge \big(u(I_s(t)) = \infty \wedge s.I(t) > l(I_s(t)) \Rightarrow \sigma(id_t)(\texttt{s\_time\_counter}) = l(I_s(t)) \big)$
> $\wedge \big(u(I_s(t)) \neq \infty \wedge s.I(t) > u(I_s(t)) \Rightarrow \sigma(id_t)(\texttt{s\_time\_counter}) = u(I_s(t)) \big)$
> $\wedge \big(u(I_s(t)) \neq \infty \wedge s.I(t) \leq u(I_s(t)) \Rightarrow s.I(t) = \sigma(id_t)(\texttt{s\_time\_counter}) \big)$.
>
> 3. $\forall t \in T_i, id_t \in Comps(\Delta) \ s.t. \ \gamma(t) = id_t, s.reset_t(t) = \sigma(id_t)(\texttt{s\_reinit\_time\_counter})$.
>
> 4. $\forall c \in \mathcal{C}, id_c \in Ins(\Delta) \ s.t. \ \gamma(c) = id_c, \ s.cond(c) = \sigma(id_c)$.
>
> 5. $\forall a \in \mathcal{A}, id_a \in Outs(\Delta) \ s.t. \ \gamma(a) = id_a, \ s.ex(a) = \sigma(id_a)$.
>
> 6. $\forall f \in \mathcal{F}, id_f \in Outs(\Delta) \ s.t. \ \gamma(f) = id_f, \ s.ex(f) = \sigma(id_f)$.

In Property 1, based on the γ binder, we relate the marking value of a place p at state s to the value of the s_marking signal inside the internal state of the place component instance (PCI) id_p. The expression $\sigma(id_p)$ returns the internal state of PCI id_p by looking up the component store of state σ. Properties 2 and 3 similarly relate the value of time counters (resp. reset orders) of transitions to the value of the signals s_time_counter (resp. s_reinit_time_counter) in the internal state of the corresponding transition component instances (TCIs). In item 4 (resp. 5 and 6), the boolean value of conditions (resp. actions and functions) are compared to the value of input (resp. output) ports of the $\mathcal{H}$-VHDL design, also based on the γ binder.

As one can observe in Property 2, the relation between the value of a time counter and the value of the s_time_counter signal is particular. It is due to the definition domain of time intervals. In the definition of the SITPN structure, a time interval i is defined as follows: $i = [a, b]$ where $a \in \mathbb{N}^*$ and $b \in \mathbb{N}^* \sqcup \{\infty\}$. In the SITPN semantics, depending on certain conditions, a time counter possibly increments its value until it reaches the upper bound of the associated time interval. Therefore, a time counter associated to a time interval with an infinite upper bound will possibly increment its value indefinitely. While acceptable in the theoretical world, this is not acceptable is the world of hardware circuits where all dimensions and values are finite. On the $\mathcal{H}$-VHDL side, the signal s_time_counter, which value represents the value of a time counter, will stop its incrementation to the lower bound of the time interval in the case

where the upper bound is infinite. As long as the value of the time counter is less than or equal to the lower bound of the time interval, we look for a perfect equality between the value of the time counter and the value of the `s_time_counter` signal. When the time counter reaches the lower bound, the values possibly diverge (i.e, the time counter value continues to be incremented while the value of the `s_time_counter` signal stalls). In that case, we are only interested in knowing that the value of the `s_time_counter` signal is equal to the value of the lower bound of the time interval. The two last subformulas of Property 2 are necessary to cover the case where a time counter has overreached the upper bound of its time interval. In that case, the time counter becomes *locked*. The `s_time_counter` signal can not overreach the upper bound of the time interval without causing an overflow. Thus, the value of the `s_time_counter` signal diverges from the value of its corresponding time counter when the time counter overreaches the upper bound of its time interval. While the time counter is less than or equal to the upper bound of its time interval, we look for a perfect equality between the value of the time counter and the value of the `s_time_counter` signal. When the time counter overreaches the upper bound, the value of the time counter stalls to upper bound plus one, and the value of `s_time_counter` stalls to upper bound. In that case, we are only interested in knowing that the value of the `s_time_counter` signal is equal to the value of the upper bound of the time interval.

The second question that we asked above was: when must the state similarity relation hold in the course of the execution? The source and target representations are both synchronously executed. Thus, we find it natural to check that the state similarity relation holds at the end of a clock cycle. However, due to modifications resulting after a bug detection (see Section 6.4), the state similarity relation of Definition 6.2 does not hold at the end of a clock cycle. The equality between the value of reset orders and the value of the `s_reinit_time_counter` signals (Property 3) is not verified. However, this semantic divergence is without effect. New reset orders are computed at the beginning of a clock cycle such that the relation of Property 3 holds in the middle of the clock cycle (i.e, just before the falling edge of the clock). This is the only moment during the clock cycle where the `s_reinit_time_counter` signal is actually involved in the computation of other signals value. Thus, it is sufficient that Property 3 holds only in the middle of the clock cycle. However, we must now define two state similarity relation; one that checks the semantic similarity after the rising edge of the clock signal (i.e, in the middle of the clock cycle), and one that checks the semantic similarity after the falling edge of the clock signal (i.e, at the end of the clock cycle). The state similarity relation after a rising edge is defined as follows:

Definition 38 (Post rising edge state similarity). *For a given $sitpn \in SITPN$, an $\mathcal{H}$-VHDL design $d \in design$, an elaborated design $\Delta \in ElDesign$, and a binder $\gamma \in WM(sitpn,d)$, an SITPN state $s \in S(sitpn)$ and a design state $\sigma \in \Sigma$ are similar after a rising edge, written $\gamma \vdash s \overset{\uparrow}{\sim} \sigma$ iff*

1. *$\forall p \in P, id_p \in Comps(\Delta)$ s.t. $\gamma(p) = id_p$, $s.M(p) = \sigma(id_p)(s\_marking)$.*

2. *$\forall t \in T_i, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$,*
   $\big(u(I_s(t)) = \infty \wedge s.I(t) \leq l(I_s(t)) \Rightarrow s.I(t) = \sigma(id_t)(s\_time\_counter)\big)$

$$\wedge \big(u(I_s(t)) = \infty \wedge s.I(t) > l(I_s(t)) \Rightarrow \sigma(id_t)(\texttt{s\_time\_counter}) = l(I_s(t)) \big)$$
$$\wedge \big(u(I_s(t)) \neq \infty \wedge s.I(t) > u(I_s(t)) \Rightarrow \sigma(id_t)(\texttt{s\_time\_counter}) = u(I_s(t)) \big)$$
$$\wedge \big(u(I_s(t)) \neq \infty \wedge s.I(t) \leq u(I_s(t)) \Rightarrow s.I(t) = \sigma(id_t)(\texttt{s\_time\_counter}) \big).$$

3. $\forall t \in T_i, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t, s.reset_t(t) = \sigma(id_t)(\texttt{s\_reinit\_time\_counter})$.

4. $\forall a \in \mathcal{A}, id_a \in Outs(\Delta)$ s.t. $\gamma(a) = id_a, s.ex(a) = \sigma(id_a)$.

5. $\forall f \in \mathcal{F}, id_f \in Outs(\Delta)$ s.t. $\gamma(f) = id_f, s.ex(f) = \sigma(id_f)$.

Definition 38 is similar to Definition 37 in all points, except for the value of conditions. A condition of an SITPN is implemented by an input port in the resulting $\mathcal{H}$-VHDL top-level design. In the $\mathcal{H}$-VHDL semantics, the value of primary input ports (i.e, the input ports of the top-level design) are updated at each clock edge. In the SITPN semantics, the value of conditions are updated only at the falling edge of the clock. Consider that a given SITPN is executed at clock cycle τ; after the rising edge of the clock, the value of conditions are equal to their value at clock cycle $\tau - 1$, whereas the value primary input ports have been updated to fresh values. Thus, we will have to wait for the next falling edge to reach the equality between condition values and input port values. Therefore, there is a semantic divergence between the value of conditions and the value of input ports in the middle of the clock cycle, i.e. just before the next falling edge of the clock signal. However, similarly to the case of reset orders and `s_reinit_time_counter` signals, conditions and their corresponding input ports are only involved in computations at the falling edge of the clock cycle. Thus, it is sufficient that Property 4 holds only right after the falling of the clock signal.

The state similarity relation draws out a correspondence between the values hold by an SITPN state and the values of the signals declared in an $\mathcal{H}$-VHDL design state. However, to complete the proof of semantic preservation, we sometimes have to relate the value of signals to the value of expressions or predicates involved in the SITPN semantics. For instance, consider a given SITPN state s and a given $\mathcal{H}$-VHDL design state σ, and consider a transition t and its corresponding TCI id_t. It is useful to show that, after a rising edge, the value of signal `s_enabled` at state $\sigma(id_t)$, where $\sigma(id_t)$ denotes the internal state of component instance id_t at state σ, is equal to the predicate $t \in Sens(s.M)$ stating that the transition t is sensitized (or *enabled*) by the marking at state s (i.e, $s.M$). Thus, for the convenience of the proof, we enrich our definitions of the state similarity relations with formulas relating $\mathcal{H}$-VHDL signals to SITPN semantics predicates and expressions. Consequently, the *full* post rising edge state similarity relation is defined as follows:

Definition 39 (Full post rising edge state similarity). *For a given sitpn $\in$ SITPN, an $\mathcal{H}$-VHDL design d $\in$ design, an elaborated design $\Delta \in$ ElDesign, and a binder $\gamma \in WM(sitpn, d)$, a clock cycle count $\tau \in \mathbb{N}$, and an SITPN execution environment $E_c \in \mathbb{N} \to \mathcal{C} \to \mathbb{B}$, an SITPN state s $\in$ S(sitpn) and a design state $\sigma \in \Sigma$ are fully similar after a rising edge happening at clock cycle count τ, written $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, if $\gamma \vdash s \overset{\uparrow}{\sim} \sigma$ (Definition 38) and*

1. $\forall t \in T, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t, t \in Sens(s.M) \Leftrightarrow \sigma(id_t)(\texttt{s\_enabled}) = \texttt{true}$.

2. $\forall t \in T, id_t \in Comps(\Delta)$ *s.t.* $\gamma(t) = id_t, t \notin Sens(s.M) \Leftrightarrow \sigma(id_t)(\mathtt{s\_enabled}) = \mathtt{false}.$

3. $\forall t \in T, id_t \in Comps(\Delta)$ *s.t.* $\gamma(t) = id_t,$

$$\sigma(id_t)(\mathtt{s\_condition\_combination}) = \prod_{c \in conds(t)} \begin{cases} E_c(\tau, c) & if \; \mathbb{C}(t,c) = 1 \\ \mathtt{not}(E_c(\tau, c)) & if \; \mathbb{C}(t,c) = -1 \end{cases}$$

   where $conds(t) = \{c \in \mathcal{C} \mid \mathbb{C}(t,c) = 1 \vee \mathbb{C}(t,c) = -1\}.$

4. $\forall c \in \mathcal{C}, id_c \in Ins(\Delta)$ *s.t.* $\gamma(c) = id_c, \sigma(id_c) = E_c(\tau, c).$

Definition 39 extends Definition 38 with the correspondence of the sensitization of transitions and the value of signal $\mathtt{s\_enabled}$, and the computation of the boolean product of condition values and the value of signal $\mathtt{s\_condition\_combination}$. The last item of Definition 39 relates the value of the input port identifiers to the value of conditions yielded by the environment at the clock cycle τ. In the $\mathcal{H}$-VHDL simulation cycle, the input ports receive new values from the environment at the beginning of the clock cycle, whereas, in the SITPN semantics, the value of conditions are updated at the falling edge of the clock signal (i.e. in the middle of the clock cycle). The last item of Definition 39 is a way to register the value of input port identifiers at the end of a rising edge phase. This information will then allow us to prove the equality of value between the input port identifiers and their corresponding conditions at the occurrence of the next falling edge.

Now, let us define the state similarity relation describing how an SITPN state and an $\mathcal{H}$-VHDL design state must be compared, after the falling edge of a clock signal:

Definition 40 (Post falling edge state similarity). *For a given* $sitpn \in SITPN$, *an* $\mathcal{H}$-VHDL *design* $d \in design$, *an elaborated design* $\Delta \in ElDesign$, *and a binder* $\gamma \in WM(sitpn, d)$, *an SITPN state* $s \in S(sitpn)$ *and a design state* $\sigma \in \Sigma$ *are similar after a falling edge, written* $\gamma \vdash s \overset{\downarrow}{\sim} \sigma$, *if*

1. $\forall p \in P, id_p \in Comps(\Delta)$ *s.t.* $\gamma(p) = id_p,$ $s.M(p) = \sigma(id_p)(\mathtt{s\_marking}).$

2. $\forall t \in T_i, id_t \in Comps(\Delta)$ *s.t.* $\gamma(t) = id_t,$
   $\big(u(I_s(t)) = \infty \wedge s.I(t) \leq l(I_s(t)) \Rightarrow s.I(t) = \sigma(id_t)(\mathtt{s\_time\_counter})\big)$
   $\wedge \big(u(I_s(t)) = \infty \wedge s.I(t) > l(I_s(t)) \Rightarrow \sigma(id_t)(\mathtt{s\_time\_counter}) = l(I_s(t))\big)$
   $\wedge \big(u(I_s(t)) \neq \infty \wedge s.I(t) > u(I_s(t)) \Rightarrow \sigma(id_t)(\mathtt{s\_time\_counter}) = u(I_s(t))\big)$
   $\wedge \big(u(I_s(t)) \neq \infty \wedge s.I(t) \leq u(I_s(t)) \Rightarrow s.I(t) = \sigma(id_t)(\mathtt{s\_time\_counter})\big).$

3. $\forall c \in \mathcal{C}, id_c \in Ins(\Delta)$ *s.t.* $\gamma(c) = id_c, s.cond(c) = \sigma(id_c).$

4. $\forall a \in \mathcal{A}, id_a \in Outs(\Delta)$ *s.t.* $\gamma(a) = id_a, s.ex(a) = \sigma(id_a).$

5. $\forall f \in \mathcal{F}, id_f \in Outs(\Delta)$ *s.t.* $\gamma(f) = id_f, s.ex(f) = \sigma(id_f).$

As explained above, Definition 40 is similar to Definition 37 except for the equality between reset orders and the value of the $\mathtt{s\_reinit\_time\_counter}$ signals. The extended version of the

post falling edge state similarity relation is defined as follows:

> **Definition 41** (Full post falling edge state similarity). *For a given sitpn $\in$ SITPN, an $\mathcal{H}$-VHDL design $d \in design$, an elaborated design $\Delta \in ElDesign$, and a binder $\gamma \in WM(sitpn, d)$, an SITPN state $s \in S(sitpn)$ and a design state $\sigma \in \Sigma$ are fully similar after a falling edge, written $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$, if $\gamma \vdash s \overset{\downarrow}{\sim} \sigma$ (Definition 40) and*
>
> 1. $\forall t \in T, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t, t \in Firable(s) \Leftrightarrow \sigma(id_t)(\mathtt{s\_firable}) = \mathtt{true}$.
>
> 2. $\forall t \in T, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t, t \notin Firable(s) \Leftrightarrow \sigma(id_t)(\mathtt{s\_firable}) = \mathtt{false}$.
>
> 3. $\forall t \in T, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t, t \in Fired(s) \Leftrightarrow \sigma(id_t)(\mathtt{fired}) = \mathtt{true}$.
>
> 4. $\forall t \in T, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t, t \notin Fired(s) \Leftrightarrow \sigma(id_t)(\mathtt{fired}) = \mathtt{false}$.
>
> 5. $\forall p \in P, id_p \in Comps(\Delta)$ s.t. $\gamma(p) = id_p, \sum_{t \in Fired(s)} pre(p,t) = \sigma(id_p)(\mathtt{s\_output\_token\_sum})$.
>
> 6. $\forall p \in P, id_p \in Comps(\Delta)$ s.t. $\gamma(p) = id_p, \sum_{t \in Fired(s)} post(t,p) = \sigma(id_p)(\mathtt{s\_input\_token\_sum})$.

Definition 41 extends Definition 40 by drawing out a correspondence between:

– The firability of transitions and the value of the signal `s_firable`.

– The firing status of transitions (i.e, transitions are fired or not) and the value of the output port `fired`.

– The sum of tokens consumed by the firing process and the value of the signal `s_output_-token_sum`.

– The sum of tokens produced by the firing process and the value of the signal `s_input_-token_sum`.

6.3 Behavior preservation theorem

In this section, we describe the major theorems and lemmas stating that the HILECOP transformation function is semantic preserving. We also present the informal proofs for these theorems and lemmas. In our proofs, we often refer to theorems and lemmas that are not yet presented at the moment of the reading. Therefore, we provide, in Appendix C, a graph of the dependencies between our high-level theorems and the other theorems and lemmas to which they appeal in their proof.

6.3.1   Proof notations

To add some readability to our proofs, we use the following notations:

– The most recent framed box above the point of reading denotes the current pending goal (what we are currently trying to prove): $\boxed{\forall n \in \mathbb{N},\ n > 0 \vee n = 0}$

– A red framed box denotes a completed goal (i.e. equivalent to QED): `true = true`

– A green framed box denotes the current induction hypothesis:

> $\forall n \in \mathbb{N},\ n + 1 > 0$

– The mention **CASE** directly follows an item bullet to denote a case during a proof by case analysis.

During a proof, we constantly refer to the names of the constants and signals declared in the $\mathcal{H}$-VHDL `place` and `transition` designs. Some constants and signals have very long names, and therefore we use aliases to refer to them in the following proofs. Table D.1 gives the full correspondence between constant and signal names and their aliases. Also, during a proof and when there is no ambiguity, id_p (resp. id_t) denotes the PCI (resp. TCI) identifier associated to a given place p (resp. transition t) through $\gamma(p) = id_p$ (resp. $\gamma(t) = id_t$), where γ is the binder returned by the transformation function. Similarly, id_c (resp. id_a and id_f) denotes the input port (resp. output port and output port) identifier associated to a given condition c (resp. action a and function f) through $\gamma(c) = id_c$.

6.3.2   Preliminary definitions

We define here some relations that are necessary to formalize our theorem of behavior preservation.

In an SITPN, the conditions associated to transitions receive fresh Boolean values from an execution environment at each falling edge of the clock. During the simulation of a top-level design, the input ports of the design receive fresh values from a simulation environment at each clock event. The transformation function generates an input port in the top-level design that will reproduce the behavior of a given SITPN condition. The binder γ, generated alongside the top-level design, relates a given condition c to its corresponding input port identifier id_c. To compare the execution/simulation traces of an SITPN and a $\mathcal{H}$-VHDL design, we must assume that the execution/simulation environments assign similar values to conditions and to their corresponding input ports at a given clock cycle. Definition 42 states that the execution environment for a given SITPN and the simulation environment for a given $\mathcal{H}$-VHDL design are similar.

> **Definition 42** (Similar environments). *For a given sitpn $\in$ SITPN, a $\mathcal{H}$-VHDL design $d \in$ design, a design store $\mathcal{D} \in$ entity-id $\nrightarrow$ design, an elaborated version $\Delta \in$ ElDesign of design d, and a binder $\gamma \in WM(sitpn, d)$, the environment $E_p \in \mathbb{N} \rightarrow Ins(\Delta) \rightarrow value$, that*

> *yields the value of the primary input ports of Δ at a given simulation cycle, and the environment E_c, that yields the value of conditions of sitpn at a given execution cycle, are similar, written $\gamma \vdash E_p \stackrel{env}{=} E_c$, if for all $\tau \in \mathbb{N}$, $c \in \mathcal{C}$, $id_c \in Ins(\Delta)$ s.t. $\gamma(c) = id_c$, $E_p(\tau)(id_c) = E_c(\tau)(c)$.*

To prove that the behavior of an SITPN and a $\mathcal{H}$-VHDL design are similar, we want to compare the states composing their execution/simulation traces. As a reminder, an execution/simulation trace is a time-ordered list of states describing the evolution of a given SITPN or $\mathcal{H}$-VHDL design through a certain number of clock cycles. The relation presented in Definition 43 allows us to compare such traces.

Definition 43 (Execution trace similarity). *For a given sitpn $\in$ SITPN, a $\mathcal{H}$-VHDL design $d \in design$, an elaborated design $\Delta \in ElDesign$, and a binder $\gamma \in WM(sitpn, d)$, the execution trace $\theta_s \in \mathtt{list}(S(sitpn))$ and the simulation trace $\theta_\sigma \in \mathtt{list}(\Sigma)$ are similar if $\gamma \vdash \theta_s \stackrel{clk}{\sim} \theta_\sigma$ (where clk $\in \{\uparrow, \downarrow\}$) is derivable according to the following rules:*

$$
\frac{}{\gamma \vdash [] \stackrel{clk}{\sim} []} \; \text{SimTraceNil} \quad clk \in \{\uparrow, \downarrow\}
$$

$$
\text{SimTrace}\uparrow \quad \frac{\gamma \vdash s \stackrel{\uparrow}{\sim} \sigma \quad \gamma \vdash \theta_s \stackrel{\downarrow}{\sim} \theta_\sigma}{\gamma \vdash (s :: \theta_s) \stackrel{\uparrow}{\sim} (\sigma :: \theta_\sigma)}
$$

$$
\text{SimTrace}\downarrow \quad \frac{\gamma \vdash s \stackrel{\downarrow}{\sim} \sigma \quad \gamma \vdash \theta_s \stackrel{\uparrow}{\sim} \theta_\sigma}{\gamma \vdash (s :: \theta_s) \stackrel{\downarrow}{\sim} (\sigma :: \theta_\sigma)}
$$

In Definition 43, the clock event symbol on top of the $\sim$ sign indicates the kind of clock event that led to the production of the states at the head of the traces. The execution trace similarity relation expects that the states composing the traces have been alternatively produced by a rising edge step and then by a falling edge step. By construction, the traces must have the same length to respect the execution trace similarity relation.

To handle the case of an execution/simulation trace beginning by an initial state, that is, a state neither reached after a rising nor after a falling edge, we give a slightly different definition of the execution trace similarity relation in Definition 44.

Definition 44 (Full execution trace similarity). *For a given sitpn $\in$ SITPN, a $\mathcal{H}$-VHDL design $d \in design$, an elaborated design $\Delta \in ElDesign(d, \mathcal{D}_\mathcal{H})$, and a binder $\gamma \in WM(sitpn, d)$, the execution trace $\theta_s \in \mathtt{list}(S(sitpn))$ and the simulation trace $\theta_\sigma \in \mathtt{list}(\Sigma)$ are fully similar, written $\gamma \vdash \theta_s \sim \theta_\sigma$, according to the following rules:*

$$
\frac{}{\gamma \vdash [] \sim []} \; \text{FullSimTraceNil}
$$

$$
\text{FullSimTraceCons} \quad \frac{\gamma \vdash s \sim \sigma \quad \gamma \vdash \theta_s \stackrel{\uparrow}{\sim} \theta_\sigma}{\gamma \vdash (s :: \theta_s) \sim (\sigma :: \theta_\sigma)}
$$

The full execution trace similarity relation indicates that the head states of traces must verify the general state similarity relation, and that the tail of the traces must respect the execution state similarity relation starting with a rising edge step.

6.3.3   The behavior preservation theorem

Theorem 1 expresses our behavior preservation theorem. Theorem 1 states that the HILECOP transformation function is semantic preserving when the input model is a well-defined (see Definition 28) and bounded (see Definition 29) SITPN. As a complementary task, we could show that if the transformation function returns a couple $\mathcal{H}$-VHDL design and binder, and not an error, then the input SITPN is well-defined. To prove Theorem 1, we must first exhibit an elaborated version of the returned $\mathcal{H}$-VHDL design (Theorem 2), an initial state (Theorem 3), and a simulation trace over τ simulation cycles (Theorem 4). Finally, we can establish that the behaviors are similar by comparing the respective SITPN execution and $\mathcal{H}$-VHDL design simulation traces (Theorem 5). In this thesis, we are focusing on the proof that the execution/simulation traces are similar when they are produced by the SITPN execution relation and the $\mathcal{H}$-VHDL simulation relation over τ clock cycles. This corresponds to the proof of Theorem 5. For the moment, we choose to consider Theorems 2, 3 and 4 as axioms.

Theorem 1 (Behavior preservation). *For all well-defined $sitpn \in SITPN$, $\mathcal{H}$-VHDL design $d \in design$, binder $\gamma \in WM(sitpn,d)$, clock cycle count $\tau \in \mathbb{N}$, execution environment $E_c \in \mathbb{N} \to \mathcal{C} \to \mathbb{B}$, execution trace $\theta_s \in \mathtt{list}(S(sitpn))$ and maximal marking function $b \in P \to \mathbb{N}$ such that*

– *SITPN sitpn is transformed into the $\mathcal{H}$-VHDL design d and yields the binder γ: $\lfloor sitpn \rfloor_b = (d, \gamma)$*

– *SITPN sitpn is bounded through b: $\lceil sitpn \rceil^b$*

– *SITPN sitpn yields the execution trace θ_s after τ execution cycles in environment E_c:*

  $E_c, \tau \vdash sitpn \xrightarrow{full} \theta_s$

then there exist an elaborated design $\Delta \in ElDesign$ and a simulation trace $\theta_\sigma \in \mathtt{list}(\Sigma)$ s.t. for all simulation environment $E_p \in \mathbb{N} \to Ins(\Delta) \to value$ verifying $\gamma \vdash E_p \overset{env}{=} E_c$ (simulation and execution environments are similar), we have:

– *In the context of the HILECOP design store $\mathcal{D}_\mathcal{H}$ and with an empty generic constant dimensioning function ($\varnothing$), design d elaborates into Δ and yields the simulation trace θ_σ after τ simulation cycles:*

  $\mathcal{D}_\mathcal{H}, \Delta, \varnothing, E_p, \tau \vdash d \xrightarrow{full} \theta_\sigma$

– *Traces θ_s and θ_σ are fully similar: $\theta_s \sim \theta_\sigma$*

Proof.

Given a $sitpn \in SITPN$, a $d \in design$, a $\gamma \in WM(sitpn,d)$, a $\tau \in \mathbb{N}$, an $E_c \in \mathbb{N} \to \mathcal{C} \to \mathbb{B}$, a $\theta_s \in \mathtt{list}(S(sitpn))$, and a $b \in P \to \mathbb{N}$, let us show that

$$\exists \Delta, \theta_\sigma, \forall E_p, \ \gamma \vdash E_p \stackrel{env}{=} E_c \Rightarrow (\mathcal{D_H}, \Delta, \varnothing, E_p, \tau \vdash \mathrm{d} \xrightarrow{full} \theta_\sigma) \wedge \theta_s \sim \theta_\sigma$$

Appealing to Theorems 2 (p. 208), 3 (p. 208) and 4 (p. 208), let us take an elaborated design $\Delta \in \textit{ElDesign}$, two design states $\sigma_e, \sigma_0 \in \Sigma$, and a simulation trace $\theta_\sigma \in \texttt{list}(\Sigma)$ such that:

- Δ is the elaborated version of design d, and σ_e is the default design state of Δ:

  $$\mathcal{D_H}, \varnothing \vdash d \xrightarrow{elab} (\Delta, \sigma_e)$$

- σ_0 is the initial simulation state: $\mathcal{D_H}, \Delta, \sigma_e \vdash d.cs \xrightarrow{init} \sigma_0$

- Design d yields the simulation trace θ_σ after τ simulation cycles, starting from initial state σ_0:
  $$\mathcal{D_H}, E_p, \Delta, \tau, \sigma_0 \vdash \mathrm{d}.cs \to \theta_\sigma$$

Let us use this Δ and this θ_σ to prove the current goal. Given an E_p such that $\gamma \vdash E_p \stackrel{env}{=} E_c$, it remains to be proved that:

$$(\mathcal{D_H}, \Delta, \varnothing, E_p, \tau \vdash \mathrm{d} \xrightarrow{full} \theta_\sigma) \wedge \theta_s \sim \theta_\sigma$$

First, we must prove that $\boxed{(\mathcal{D_H}, \Delta, \varnothing, E_p, \tau \vdash \mathrm{d} \xrightarrow{full} \theta_\sigma)}$ holds. By definition of the $\mathcal{H}$-VHDL full simulation relation, we have:

$$\mathcal{D_H}, \Delta, \varnothing, E_p, \tau \vdash \mathrm{d} \xrightarrow{full} \theta_\sigma \equiv \exists \sigma_e, \sigma_0 \in \Sigma(\Delta), \ \mathcal{D_H}, \varnothing \vdash d \xrightarrow{elab} (\Delta, \sigma_e)$$
$$\wedge \mathcal{D_H}, \Delta, \sigma_e \vdash d.cs \xrightarrow{init} \sigma_0 \qquad (6.1)$$
$$\wedge \mathcal{D_H}, E_p, \Delta, \tau, \sigma_0 \vdash \mathrm{d}.cs \to \theta_\sigma$$

Thus, it is equivalent to prove:

$$\exists \sigma_e, \sigma_0 \ \text{s.t.} \ \mathcal{D_H}, \varnothing \vdash d \xrightarrow{elab} (\Delta, \sigma_e) \wedge \mathcal{D_H}, \Delta, \sigma_e \vdash d.cs \xrightarrow{init} \sigma_0 \wedge \mathcal{D_H}, E_p, \Delta, \tau, \sigma_0 \vdash \mathrm{d}.cs \to \theta_\sigma$$

To prove the goal, let us use $\sigma_e, \sigma_0 \in \Sigma$ previously introduced by the invocation of Theorems 2, 3 and 4. Then, the three first points of the goal are previously assumed hypotheses.

Finally, appealing to Theorem 5, we can prove final point of the theorem, i.e. $\boxed{\theta_s \sim \theta_\sigma}$.

$\square$

The behavior preservation theorem is a case of *simulation* theorem. It states that if an execution trace is computed on the SITPN side, then there exists an simulation trace on the $\mathcal{H}$-VHDL side and that both traces are similar. To express a *bisimulation* theorem, the other way around

must also be true, i.e. if a simulation trace is computed on the $\mathcal{H}$-VHDL side, then an execution trace exists on the SITPN side, and both traces are similar.

Theorem 2 states that every $\mathcal{H}$-VHDL design returned by the HILECOP transformation function can be elaborated. The elaboration relation verifies that a given $\mathcal{H}$-VHDL design is well-typed and well-formed w.r.t. to the VHDL language standards, and builds an elaborated version of the $\mathcal{H}$-VHDL design that will act as a simulation environment. Thus, Theorem 2 states that the HILECOP transformation function produces *acceptable* code, for instance, code that could be the input to a simulator program.

> **Theorem 2** (Elaboration). *For all well-defined sitpn $\in$ SITPN, $d \in$ design, $\gamma \in WM(sitpn, d)$ and $b \in P \rightarrow \mathbb{N}$ such that*
>
> – $\lfloor sitpn \rfloor_b = (d, \gamma)$
>
> *then there exists an elaborated design $\Delta \in$ ElDesign and a design state $\sigma_e \in \Sigma$ s.t. Δ is the elaborated version of design d, and σ_e is the default design state of Δ: $\mathcal{D}_{\mathcal{H}}, \varnothing \vdash d \xrightarrow{elab} (\Delta, \sigma_e)$.*

Theorem 3 states that one can always build an initial state for every $\mathcal{H}$-VHDL design returned by the HILECOP transformation function, if the input SITPN model is well-defined and bounded.

> **Theorem 3** (Initialization). *For all well-defined sitpn $\in$ SITPN, $d \in$ design, $b \in P \rightarrow \mathbb{N}$, $\gamma \in WM(sitpn, d)$, $\Delta \in$ ElDesign, $\sigma_e \in \Sigma(\Delta)$ s.t.*
>
> – $\lfloor sitpn \rfloor_b = (d, \gamma)$ *and* $\lceil sitpn \rceil^b$ *and* $\mathcal{D}_{\mathcal{H}}, \varnothing \vdash d \xrightarrow{elab} (\Delta, \sigma_e)$
>
> *then there exists a design state $\sigma_0 \in \Sigma(\Delta)$ s.t. σ_0 is the initial simulation state: $\mathcal{D}_{\mathcal{H}}, \Delta, \sigma_e \vdash d.cs \xrightarrow{init} \sigma_0$.*

Theorem 4 states that one can always build a simulation trace over τ clock cycles for every $\mathcal{H}$-VHDL design returned by the HILECOP transformation function. This means that the simulation of an $\mathcal{H}$-VHDL design never fails when it is the result of the transformation of a well-defined SITPN.

> **Theorem 4** (Trace existence). *For all well-defined sitpn $\in$ SITPN, $d \in$ design, $b \in P \rightarrow \mathbb{N}$, $\gamma \in WM(sitpn, d)$, $\Delta \in$ ElDesign, $\sigma_e, \sigma_0 \in \Sigma$ s.t.*
>
> – $\lfloor sitpn \rfloor_b = (d, \gamma)$ *and* $\lceil sitpn \rceil^b$ *and* $\mathcal{D}_{\mathcal{H}}, \varnothing \vdash d \xrightarrow{elab} (\Delta, \sigma_e)$ *and* $\mathcal{D}_{\mathcal{H}}, \Delta, \sigma_e \vdash d.cs \xrightarrow{init} \sigma_0$
>
> *then there exists a simulation trace $\theta_\sigma \in$ list(Σ) such that for all simulation environment $E_p \in \mathbb{N} \rightarrow Ins(\Delta) \rightarrow value$ and simulation cycle count $\tau \in \mathbb{N}$, design d yields the simulation trace θ_σ after τ simulation cycles, starting from initial state σ_0:*
> $\mathcal{D}_{\mathcal{H}}, E_p, \Delta, \tau, \sigma_0 \vdash d.cs \rightarrow \theta_\sigma$

Remark 9 (Bounded SITPN and behavior preservation). *A part of the analysis is interested in determining the maximal number of tokens that a place can hold during the execution of a SITPN. If each place of the SITPN can only hold a limited number of tokens during the execution of the model, then the model is said to be* bounded. *In that case, it is possible to compute a function that associates the places of the SITPN with a maximal marking value. In the case of an unbounded input model, there exists a place that can accumulate an infinite number of tokens during the model execution. In the world of hardware description, and especially when aiming at hardware synthesis, every element must have a finite dimension. In the definition of the* `place` *design, the internal signal* `s_marking` *represents the marking value of a place. The maximal value of the* `s_marking` *signal is bounded by the generic constant* `maximal_marking`. *Thus, when generating a PCI from a place in the course of the transformation, we must give a value to the* `maximal_marking` *generic constant. However, even with a settled* `maximal_marking` *value, the execution of a* $\mathcal{H}$*-VHDL design, resulting from the transformation of an unbounded SITPN model, infallibly leads to the overflow of the value of the* `s_marking` *signals in the internal states of PCIs. Consider an unbounded place p and its corresponding PCI* id_p. *There exist a clock cycle count* τ *for which the value of the* `s_marking` *internal signal (which reflects the marking of place p) of PCI* id_p *overreaches the* `maximal_marking` *value, thus causing an overflow. In that case, because of the overflow, the next design state (i.e. at clock count* $\tau + 1$*) can never be derived. Thus, passed the clock cycle count* τ*, the simulation of the* $\mathcal{H}$*-VHDL design ends, the execution of the corresponding unbounded SITPN model continues, and we are no more able to prove the equivalence between the two behaviors.*

6.3.4 The trace similarity theorem

Here, we present the trace similarity theorem. The trace similarity theorem states that if an SITPN and its corresponding $\mathcal{H}$-VHDL design are executed/simulated over τ execution/simulation cycles, then the produced traces are semantically similar, i.e. they verify the full execution trace similarity relation of Definition 44. In this thesis, we have proved this particular theorem. The proofs of Theorems 2, 3 and 4 have been left for future work. We chose to focus our work on the trace similarity theorem, because it directly addresses the semantic preservation property of HILECOP's transformation function.

In the proof of Theorem 5, in the case where $\tau > 0$, we must show that the state similarity relation holds between the states produced by the first execution cycle, and then use Lemma 1 (p. 213) to complete the proof of similarity between the tail traces. First, we must show that the initial states of both SITPN and $\mathcal{H}$-VHDL design verify the general state similarity relation (Definition 37); this is done by appealing to Lemma 5 (p. 266). The first execution cycle is particular because, by definition of the SITPN full execution relation, no transitions are fired during the first rising edge. Therefore, after the first rising edge, the SITPN state is still equal to its initial state s_0. We prove that the post rising edge similarity relation is verified after the first rising edge by appealing to Lemma 15 (p. 276). The detailed proofs for Lemmas 5 and 15 are given in Sections D.1 and D.2.

Theorem 5 (Full trace similarity). *For all $sitpn \in SITPN$, $b \in P \to \mathbb{N}$, $d \in design$, $\gamma \in WM(sitpn, d)$, $\tau \in \mathbb{N}$, $E_c \in \mathbb{N} \to \mathcal{C} \to \mathbb{B}$, $\theta_s \in \text{list}(S(sitpn))$, $\Delta \in ElDesign$, $E_p \in \mathbb{N} \to Ins(\Delta) \to value$, $\theta_\sigma \in \text{list}(\Sigma)$ such that*

– $\lfloor sitpn \rfloor_b = (d, \gamma)$

– $\gamma \vdash E_p \stackrel{env}{=} E_c$

– $E_c, \tau \vdash sitpn \xrightarrow{full} \theta_s$

– $\mathcal{D}_\mathcal{H}, \Delta, \varnothing, E_p, \tau \vdash d \xrightarrow{full} \theta_\sigma$

then $\gamma \vdash \theta_s \sim \theta_\sigma$

Proof.

Assuming the above hypotheses, let us show $\boxed{\gamma \vdash \theta_s \sim \theta_\sigma}$ (1).
Let us perform case analysis on the given clock count τ; there are two cases:

– **CASE $\tau = 0$.** By definition of the SITPN full execution and the $\mathcal{H}$-VHDL full simulation relations, we have:

  – $E_c, 0 \vdash sitpn \xrightarrow{full} [s_0]$ and $\theta_s = [s_0]$

  – $\mathcal{D}_\mathcal{H}, \varnothing \vdash d \xrightarrow{elab} (\Delta, \sigma_e)$ and $\Delta, \sigma_e \vdash d.cs \xrightarrow{init} \sigma_0$ and $\mathcal{D}_\mathcal{H}, E_p, \Delta, 0, \sigma_0 \vdash d.cs \to [\,]$ and $\theta_\sigma = [\sigma_0]$

  Rewriting θ_s as $[s_0]$, and θ_σ as $[\sigma_0]$ in goal (1), and by definition of the full execution trace similarity relation, what is left to prove is: $\boxed{\gamma \vdash s_0 \sim \sigma_0}$

  Appealing to Lemma 5 (p. 266), we can show $\gamma \vdash s_0 \sim \sigma_0.$

– **CASE $\tau > 0$.** By definition of the SITPN full execution relation (i.e. $E_c, \tau \vdash sitpn \xrightarrow{full} \theta_s$) and the $\mathcal{H}$-VHDL full simulation relation (i.e, $\mathcal{D}_\mathcal{H}, \Delta, \varnothing, E_p, \tau \vdash d \xrightarrow{full} \theta_\sigma$), we have:

  – $E_c, \tau \vdash s_0 \xrightarrow{\uparrow_0} s_0$ and $E_c, \tau \vdash s_0 \xrightarrow{\downarrow} s$ and $E_c, \tau - 1 \vdash sitpn, s \to \theta$ and $\theta_s = s_0 :: s_0 :: s :: \theta$

  – $\mathcal{D}_\mathcal{H}, \varnothing \vdash d \xrightarrow{elab} (\Delta, \sigma_e)$ and $\Delta, \sigma_e \vdash d.cs \xrightarrow{init} \sigma_0$ and $E_p, \Delta, \tau, \sigma_0 \vdash d.cs \to \theta'$ and $\theta_\sigma = \sigma_0 :: \theta'$

  Rewriting θ_s as $s_0 :: s_0 :: s :: \theta$ and θ_σ as $\sigma_0 :: \theta'$ in goal (1), the new goal is: $\boxed{\gamma \vdash (s_0 :: s_0 :: s :: \theta) \sim (\sigma_0 :: \theta')}$ (2)

By definition of the $\mathcal{H}$-VHDL simulation relation (i.e. $E_p, \Delta, \tau, \sigma_0 \vdash \text{d.cs} \to \theta'$), we have:

$E_p, \Delta, \tau, \sigma_0 \vdash \text{d.cs} \xrightarrow{\uparrow,\downarrow} \sigma, \sigma'$ and $E_p, \Delta, \tau - 1, \sigma' \vdash \text{d.cs} \to \theta''$ and $\theta' = \sigma :: \sigma' :: \theta''$.

Rewriting θ' as $\sigma :: \sigma' :: \theta''$ in goal (2), the new goal is:

$\boxed{\gamma \vdash (s_0 :: s_0 :: s :: \theta) \sim (\sigma_0 :: \sigma :: \sigma' :: \theta'')}$ (3)

By definition of the full execution trace similarity relation, there are four points to prove:

1. $\boxed{\gamma \vdash s_0 \sim \sigma_0.}$ Appealing to Lemma 5, we can show $\boxed{\gamma \vdash s_0 \sim \sigma_0}$.

2. $\boxed{\gamma, E_c, \tau \vdash s_0 \overset{\uparrow}{\sim} \sigma.}$ Appealing to Lemma 15 (p. 276), we have $\gamma, E_c, \tau \vdash s_0 \overset{\uparrow}{\approx} \sigma$.

   By definition of $\gamma, E_c, \tau \vdash s_0 \overset{\uparrow}{\approx} \sigma$, we can show $\boxed{\gamma, E_c, \tau \vdash s_0 \overset{\uparrow}{\sim} \sigma}$.

3. $\boxed{\gamma \vdash s \overset{\downarrow}{\sim} \sigma'.}$ Appealing to Lemma 15 and 3 (p. 217), we have $\gamma \vdash s \overset{\downarrow}{\approx} \sigma'$.

   By definition of $\gamma \vdash s \overset{\downarrow}{\approx} \sigma'$, we can show $\boxed{\gamma \vdash s \overset{\downarrow}{\sim} \sigma'}$.

4. $\boxed{\gamma \vdash \theta \overset{\updownarrow}{\sim} \theta''.}$

   Appealing to Lemma 15 and 3, we have $\gamma \vdash s \overset{\downarrow}{\approx} \sigma'$.

   Then, we can appeal to Lemma 1 to show $\boxed{\gamma \vdash \theta \overset{\updownarrow}{\sim} \theta''}$.

   $\square$

To prove the semantic preservation property, we want to prove that a given SITPN and its translated $\mathcal{H}$-VHDL version follow the simulation diagram of Figure 6.3.
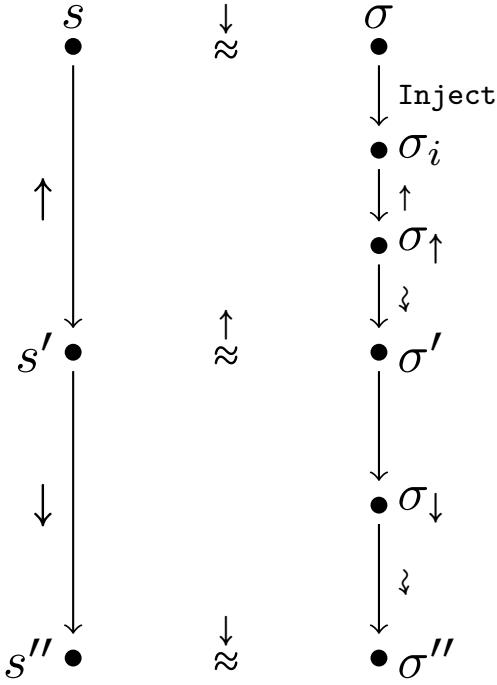
FIGURE 6.3: Simulation diagram over one clock cycle for a source SITPN and a target $\mathcal{H}$-VHDL design; the left part of the diagram presents the execution of an SITPN over one clock cycle, and the right part of the diagram presents the simulation of an $\mathcal{H}$-VHDL design over one clock cycle; the upper part of the diagram corresponds to the rising edge phase of the clock cycle, and the lower part illustrates the falling edge phase of the clock cycle.

The upper part of the diagram, corresponding to the rising edge phase of the clock cycle, is proved by Lemma 2 (p. 215). First, we assume that the starting SITPN state and the starting $\mathcal{H}$-VHDL design state verify the full post falling edge state similarity relation at the beginning of the clock cycle (i.e, $s \overset{\downarrow}{\approx} \sigma$ in Figure 6.3). Then, Lemma 2 states that after the computation of a rising edge step on the SITPN part and on the $\mathcal{H}$-VHDL part the resulting states verify the full post rising edge state similarity relation. The lower part of the diagram, corresponding to the falling edge phase of the clock cycle, is proved by Lemma 3 (p. 217). First, we assume that the starting SITPN state and the starting $\mathcal{H}$-VHDL state verify the full post rising edge state similarity relation (i.e. $s' \overset{\uparrow}{\approx} \sigma'$ in Figure 6.3). Then, Lemma 3 states that after the computation of a falling edge step on the SITPN part and on the $\mathcal{H}$-VHDL part the resulting states verify the full post falling edge state similarity relation.

Lemma 1 is a generalization of Theorem 5, and it is used in the proof of Theorem 5 to solve the induction case. Lemma 1 is similar to Theorem 5 excepts that the execution/simulation traces are not produced starting from the initial states, but starting from two states verifying the full post falling edge state similarity relation (i.e, $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$, corresponding to the kind of similarity relation that must hold when considering two states at the beginning of a random

clock cycle). The proof of Lemma 1 is based on the simulation diagram of Figure 6.3. The SITPN execution relation and the $\mathcal{H}$-VHDL simulation relation execute one computational step at clock count τ and then decrement the clock count and call themselves recursively to produce the rest of the execution/simulation traces. Therefore, the proof of Lemma 1 is naturally done by induction over the clock count τ.

Lemma 1 (Trace Similarity). *For all sitpn* $\in$ *SITPN, $b \in P \to \mathbb{N}$, $d \in$ design, $\gamma \in WM(sitpn, d)$, $\Delta \in ElDesign$, $E_p \in \mathbb{N} \to Ins(\Delta) \to value$, $E_c \in \mathbb{N} \to \mathcal{C} \to \mathbb{B}$, $\tau \in \mathbb{N}$, $s \in S(sitpn)$, $\sigma_e, \sigma \in \Sigma$, $\theta_s, \theta_\sigma \in$ list(Σ), such that*

- $\lfloor sitpn \rfloor_b = (d, \gamma)$ *and* $\gamma \vdash E_p \overset{env}{=\!=} E_c$ *and* $\mathcal{D}_\mathcal{H}, \varnothing \vdash d \xrightarrow{elab} \Delta, \sigma_e$

- *Starting states are fully similar as intended after a falling edge:* $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$

- $E_c, \tau \vdash sitpn, s \to \theta_s$

- $E_p, \Delta, \tau, \sigma \vdash d.cs \to \theta_\sigma$

then $\gamma \vdash \theta_s \overset{\uparrow}{\sim} \theta_\sigma$.

Proof.

Given a *sitpn*, b, d, γ, E_p, E_c, τ, such that $\lfloor sitpn \rfloor_b = (d, \gamma)$ and $\gamma \vdash E_p \overset{env}{=\!=} E_c$ and $\mathcal{D}_\mathcal{H}, \varnothing \vdash d \xrightarrow{elab} \Delta, \sigma_e$, let us show

$$\forall s, \sigma, \theta_s, \theta_\sigma \text{ s.t. } \gamma \vdash s \overset{\downarrow}{\approx} \sigma \text{ and } E_c, \tau \vdash sitpn, s \to \theta_s \text{ and } E_p, \Delta, \tau, \sigma \vdash d.cs \to \theta_\sigma \text{ then } \gamma \vdash \theta_s \overset{\uparrow}{\approx} \theta_\sigma.$$

Let us reason by induction on the given clock cycle τ.

- **BASE CASE**: $\tau = 0$. Then, $\sigma_s = \sigma_\sigma = [\,]$ and by definition of the execution trace similarity relation, we can show $\gamma \vdash [\,] \overset{\uparrow}{\approx} [\,]$.

- **INDUCTION CASE**: Assuming the following induction hypothesis

$$\forall s, \sigma, \theta_s, \theta_\sigma \text{ s.t. } \gamma \vdash s \overset{\downarrow}{\approx} \sigma \text{ and } E_c, \tau \vdash sitpn, s \to \theta_s \text{ and } E_p, \Delta, \tau\sigma \vdash d.cs \to \theta_\sigma \text{ then } \gamma \vdash \theta_s \overset{\uparrow}{\approx} \theta_\sigma.$$

we must prove the goal at $\tau + 1$, i.e.:

$\forall s, \sigma, \theta_s, \theta_\sigma$ s.t. $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$ and $E_c, \tau + 1 \vdash sitpn, s \to \theta_s$ and $E_p, \Delta, \tau + 1, \sigma \vdash d.cs \to \theta_\sigma$ then $\gamma \vdash \theta_s \overset{\uparrow}{\approx} \theta_\sigma$.

Given $s, \sigma, \theta_s, \theta_\sigma$ such that $(\gamma \vdash s \overset{\downarrow}{\approx} \sigma)$ and $(E_c, \tau + 1 \vdash sitpn, s \to \theta_s)$ and $(E_p, \Delta, \tau + 1, \sigma \vdash d.cs \to \theta_\sigma)$, let us show $\boxed{\gamma \vdash \theta_s \overset{\uparrow}{\approx} \theta_\sigma}$.

By definition of $(E_c, \tau + 1 \vdash sitpn, s \to \theta_s)$ and $(E_p, \Delta, \tau + 1, \sigma \vdash d.cs \to \theta_\sigma)$, we have:

- $E_c, \tau + 1 \vdash s \overset{\uparrow}{\to} s'$ and $E_c, \tau + 1 \vdash s' \overset{\downarrow}{\to} s''$ and $E_c, \tau \vdash sitpn, s'' \to \theta$.

- $\texttt{Inject}(\sigma, E_p, \tau + 1, \sigma_i)$

- $\Delta, \sigma_i \vdash d.cs \overset{\uparrow}{\to} \sigma_\uparrow$ and $\Delta, \sigma_\uparrow \vdash d.cs \overset{\rightsquigarrow}{\to} \sigma'$

- $\Delta, \sigma' \vdash d.cs \overset{\downarrow}{\to} \sigma_\downarrow$ and $\Delta, \sigma_\downarrow \vdash d.cs \overset{\rightsquigarrow}{\to} \sigma''$

- $E_p, \Delta, \tau, \sigma'' \vdash d.cs \to \theta'$

- $\theta_s = s' :: s'' :: \theta$ and $\theta_\sigma = \sigma' :: \sigma'' :: \theta'$

Then, the new goal is: $\boxed{\gamma \vdash (s' :: s'' :: \theta) \overset{\uparrow}{\sim} (\sigma' :: \sigma'' :: \theta').}$

By definition of the execution trace similarity relation, there are three points to prove:

1. $\boxed{\gamma \vdash s' \overset{\uparrow}{\sim} \sigma'.}$ Appealing to Lemma 3 (p. 217), we have $\gamma \vdash s' \overset{\uparrow}{\approx} \sigma'$.

   By definition of $\gamma \vdash s' \overset{\uparrow}{\approx} \sigma'$, we can show $\boxed{\gamma \vdash s' \overset{\uparrow}{\sim} \sigma'.}$

2. $\boxed{\gamma \vdash s'' \overset{\downarrow}{\sim} \sigma''.}$ Appealing to Lemmas 3 and 2, we have $\gamma, E_c, \tau \vdash s' \overset{\downarrow}{\approx} \sigma'$.

   By definition of $\gamma, E_c, \tau \vdash s' \overset{\downarrow}{\approx} \sigma'$, we can show $\boxed{\gamma \vdash s' \overset{\downarrow}{\sim} \sigma'.}$

3. $\boxed{\gamma \vdash \theta \overset{\uparrow}{\sim} \theta'.}$

   We can apply the induction hypothesis with $s = s''$, $\sigma = \sigma''$, $\theta_s = \theta$ and $\theta_\sigma = \theta'$.

   Then, what is left to prove is: $\boxed{\gamma \vdash s'' \overset{\downarrow}{\approx} \sigma''.}$

   Using Lemmas 3 and 2, we can show $\boxed{\gamma \vdash s'' \overset{\downarrow}{\approx} \sigma''.}$

   $\square$

Now, let us present Lemma 2 and Lemma 3, along with their proofs. In the two lemmas, we added an hypothesis, which can always be verified at the beginning of a clock cycle, about

the starting state of the $\mathcal{H}$-VHDL design: $\mathcal{D}_{\mathcal{H}}, \Delta, \sigma \vdash \text{d.cs} \xrightarrow{comb} \sigma$. This hypothesis states that all signal values are stable at the beginning of the considered clock phase. This means that the execution of the combinational part of the $\mathcal{H}$-VHDL design does not change the value of signals anymore. This hypothesis is required to determine the expression associated to combinational signals, i.e. the *combinational equations*, at the beginning of the clock phase (see Section 6.4 for more details about combinational equations).

To prove Lemmas 2 and 3, one must prove that every point of the state similarity relation in the conclusion holds. For each point, the proof is given as a separate lemma that the reader will find in Appendix D. The proof strategy to show the equalities or equivalences involved in the state similarity relation follows the same two-fold pattern:

– First, reason on the SITPN structure and on the transformation function to determine the content of the target $\mathcal{H}$-VHDL design.

– Then, reason on the SITPN state transition relation and the $\mathcal{H}$-VHDL "simulation" relations (i.e, the Inject, $\uparrow$, $\downarrow$ and $\rightsquigarrow$ relations) to establish the equality between the values coming from the SITPN world (i.e, marking, time counters, reset orders, etc. and also predicates) and the values of the signals declared in the $\mathcal{H}$-VHDL design and in its internal component instances.

The application of this proof strategy will be detailed in Section 6.4.

Lemma 2 (Rising edge). *For all* $sitpn \in SITPN$, $b \in P \rightarrow \mathbb{N}$, $d \in design$, $\gamma \in WM(sitpn, d)$, $E_c \in \mathbb{N} \rightarrow \mathcal{C} \rightarrow \mathbb{B}$, $\Delta \in ElDesign$, $E_p \in \mathbb{N} \rightarrow Ins(\Delta) \rightarrow value$, $\tau \in \mathbb{N}$, $s, s' \in S(sitpn)$, $\sigma_e, \sigma, \sigma_i, \sigma_\uparrow, \sigma' \in \Sigma$, *such that*

– $\lfloor sitpn \rfloor_b = (d, \gamma)$ *and* $\gamma \vdash E_p \overset{env}{=} E_c$ *and* $\mathcal{D}_{\mathcal{H}}, \emptyset \vdash \text{d} \xrightarrow{elab} \Delta, \sigma_e$

– $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$

– $E_c, \tau \vdash s \xrightarrow{\uparrow} s'$

– $\text{Inject}(\sigma, E_p, \tau, \sigma_i)$ *and* $\mathcal{D}_{\mathcal{H}}, \Delta, \sigma_i \vdash \text{d.cs} \xrightarrow{\uparrow} \sigma_\uparrow$ *and* $\mathcal{D}_{\mathcal{H}}, \Delta, \sigma_\uparrow \vdash \text{d.cs} \xrightarrow{\rightsquigarrow} \sigma'$

– *State* σ *is a stable design state:* $\mathcal{D}_{\mathcal{H}}, \Delta, \sigma \vdash \text{d.cs} \xrightarrow{comb} \sigma$

then $\gamma, E_c, \tau \vdash s' \overset{\uparrow}{\approx} \sigma'$.

Proof.

By definition of the Full post rising edge state similarity relation, there are 9 points to prove:

1. $\forall p \in P, id_p \in Comps(\Delta)$ $s'.t.$ $\gamma(p) = id_p$, $s'.M(p) = \sigma'(id_p)(\texttt{s\_marking})$.

2. $\forall t \in T_i, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$,
   $\big(u(I_s(t)) = \infty \wedge s'.I(t) \leq l(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(\texttt{s\_time\_counter})\big)$
   $\wedge \big(u(I_s(t)) = \infty \wedge s'.I(t) > l(I_s(t)) \Rightarrow \sigma'(id_t)(\texttt{s\_time\_counter}) = l(I_s(t))\big)$
   $\wedge \big(u(I_s(t)) \neq \infty \wedge s'.I(t) > u(I_s(t)) \Rightarrow \sigma'(id_t)(\texttt{s\_time\_counter}) = u(I_s(t))\big)$
   $\wedge \big(u(I_s(t)) \neq \infty \wedge s'.I(t) \leq u(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(\texttt{s\_time\_counter})\big)$.

3. $\forall t \in T_i, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$,
   $s'.reset_t(t) = \sigma'(id_t)(\texttt{s\_reinit\_time\_counter})$.

4. $\forall a \in \mathcal{A}, id_a \in Outs(\Delta)$ s.t. $\gamma(a) = id_a$, $s'.ex(a) = \sigma'(id_a)$.

5. $\forall f \in \mathcal{F}, id_f \in Outs(\Delta)$ s.t. $\gamma(f) = id_f$, $s'.ex(f) = \sigma'(id_f)$.

6. $\forall t \in T, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$,
   $t \in Sens(s'.M) \Leftrightarrow \sigma'(id_t)(\texttt{s\_enabled}) = \texttt{true}$.

7. $\forall t \in T, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$,
   $t \notin Sens(s'.M) \Leftrightarrow \sigma'(id_t)(\texttt{s\_enabled}) = \texttt{false}$.

8. $\forall t \in T, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$,
   $$\sigma'(id_t)(\texttt{s\_condition\_combination}) = \prod_{c \in conds(t)} \begin{cases} E_c(\tau, c) & if\ \mathbb{C}(t,c) = 1 \\ \texttt{not}(E_c(\tau, c)) & if\ \mathbb{C}(t,c) = -1 \end{cases}$$
   where $conds(t) = \{c \in \mathcal{C} \mid \mathbb{C}(t,c) = 1 \vee \mathbb{C}(t,c) = -1\}$.

9. $\forall c \in \mathcal{C}, id_c \in Ins(\Delta)$ s.t. $\gamma(c) = id_c$, $\sigma'(id_c) = E_c(\tau, c)$.

Each point is proved by a separate lemma:

– Apply the Rising edge equal marking lemma (p. 285) to solve Point 1.

– Apply the Rising edge equal time counters lemma (p. 289) to solve Point 2.

– Apply the Rising edge equal reset orders lemma (p. 290) to solve Point 3.

– Apply the Rising edge equal action executions lemma (p. 298) to solve Point 4.

– Apply the Rising edge equal function executions lemma (p. 299) to solve Point 5.

– Apply the Rising edge equal sensitized lemma (p. 301) to solve Point 6.

– Apply the Rising edge equal not sensitized lemma (p. 305) to solve Point 7.

– Apply the Rising edge equal condition combination lemma (p. 286) to solve Point 8.

– Apply the Rising edge equal conditions lemma (p. 288) to solve Point 9.

All the lemmas used above, and their corresponding proofs, are to be found in Appendix D, Section D.3. $\square$

Lemma 3 (Falling edge). *For all $sitpn \in SITPN$, $b \in P \to \mathbb{N}$, $d \in design$, $\gamma \in WM(sitpn, d)$, $E_c \in \mathbb{N} \to \mathcal{C} \to \mathbb{B}$, $\Delta \in ElDesign$, $E_p \in \mathbb{N} \to Ins(\Delta) \to value$, $\tau \in \mathbb{N}$, $s, s' \in S(sitpn)$, $\sigma_e, \sigma, \sigma_\downarrow, \sigma' \in \Sigma$, such that*

– $\lfloor sitpn \rfloor_b = (d, \gamma)$ *and* $\gamma \vdash E_p \overset{env}{=} E_c$ *and* $\mathcal{D}_\mathcal{H}, \emptyset \vdash d \xrightarrow{elab} \Delta, \sigma_e$

– $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$

– $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$

– $\Delta, \sigma \vdash d.cs \xrightarrow{\downarrow} \sigma_\downarrow$ *and* $\Delta, \sigma_\downarrow \vdash d.cs \xrightarrow{\leadsto} \sigma'$

– *State σ is a stable design state:* $\mathcal{D}_\mathcal{H}, \Delta, \sigma \vdash d.cs \xrightarrow{comb} \sigma$

then $\gamma \vdash s' \overset{\downarrow}{\approx} \sigma'$.

Proof.

By definition of the Post falling edge state similarity relation, there are 11 points to prove:

1. $\forall p \in P, id_p \in Comps(\Delta)$ s.t. $\gamma(p) = id_p$, $s'.M(p) = \sigma'(id_p)(\text{s\_marking})$.

2. $\forall t \in T_i, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$,
   $\big(u(I_s(t)) = \infty \wedge s'.I(t) \leq l(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(\text{s\_time\_counter})\big)$
   $\wedge \big(u(I_s(t)) = \infty \wedge s'.I(t) > l(I_s(t)) \Rightarrow \sigma'(id_t)(\text{s\_time\_counter}) = l(I_s(t))\big)$
   $\wedge \big(u(I_s(t)) \neq \infty \wedge s'.I(t) > u(I_s(t)) \Rightarrow \sigma'(id_t)(\text{s\_time\_counter}) = u(I_s(t))\big)$
   $\wedge \big(u(I_s(t)) \neq \infty \wedge s'.I(t) \leq u(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(\text{s\_time\_counter})\big)$.

3. $\forall c \in \mathcal{C}, id_c \in Ins(\Delta)$ s.t. $\gamma(c) = id_c$, $s'.cond(c) = \sigma'(id_c)$.

4. $\forall a \in \mathcal{A}, id_a \in Outs(\Delta)$ s.t. $\gamma(a) = id_a$, $s'.ex(a) = \sigma'(id_a)$.

5. $\forall f \in \mathcal{F}, id_f \in Outs(\Delta)$ s.t. $\gamma(f) = id_f$, $s'.ex(f) = \sigma'(id_f)$.

6. $\forall t \in T, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$,
   $t \in Firable(s') \Leftrightarrow \sigma'(id_t)(\text{s\_firable}) = \text{true}$.

7. $\forall t \in T, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$,
   $t \notin Firable(s') \Leftrightarrow \sigma'(id_t)(\text{s\_firable}) = \text{false}$.

8. $\forall t \in T, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, $t \in Fired(s') \Leftrightarrow \sigma'(id_t)(\text{fired}) = \text{true}$.

9. $\forall t \in T, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, $t \notin Fired(s') \Leftrightarrow \sigma'(id_t)(\text{fired}) = \text{false}$.

10. $\forall p \in P, id_p \in Comps(\Delta)$ s.t. $\gamma(p) = id_p$,
    $\sum\limits_{t \in Fired(s')} pre(p,t) = \sigma'(id_p)(\text{s\_output\_token\_sum})$.

11. $\forall p \in P, id_p \in Comps(\Delta)$ s.t. $\gamma(p) = id_p$,
    $\sum\limits_{t \in Fired(s')} post(t,p) = \sigma'(id_p)(\text{s\_input\_token\_sum})$.

Each point is proved by a separate lemma:

– Apply the Falling edge equal marking lemma (p. 305) to solve Point 1.

– Apply the Falling edge equal time counters lemma (p. 313) to solve Point 2.

– Apply the Falling edge equal condition values lemma (p. 319) to solve Point 3.

– Apply the Falling edge equal action executions lemma (p. 319) to solve Point 4.

– Apply the Falling edge equal function executions lemma (p. 322) to solve Point 5.

– Apply the Falling edge equal firable lemma (p. 322) to solve Point 6.

– Apply the Falling edge equal not firable lemma (p. 335) to solve Point 7.

> – Apply the Falling edge equal fired lemma (p. 220) to solve Point 8. The proof of the Falling edge equal fired lemma is detailled in Section 6.4.
>
> – Apply the Falling edge equal not fired lemma (p. 351) to solve Point 9.
>
> – Apply the Falling edge equal output token sum lemma (p. 306) to solve Point 10.
>
> – Apply the Falling edge equal input token sum lemma (p. 309) to solve Point 11.
>
> All the lemmas used above, and their corresponding proofs, are to be found in Appendix D, Section D.4. □

6.4 A detailled proof: equivalence of fired transitions

The goal of this section is to present the overall proof strategy that we adopted to establish the semantic preservation property for the HILECOP model-to-text transformation. We take advantage of the proof of Lemma 4, involved in the proof of Lemma 3, to illustrate our demonstration techniques. The proof of Lemma 4 has been one complex part of the overall demonstration; we believe it is worth to be mentioned. Also, it has led to a bug detection. We give a full account on this bug detection, and on how we manage to correct it, at the end of the section.

6.4.1 An accompanied journey along the proof

The proof of Lemma 4 is related to the set of fired transitions. In an SITPN, the firing process, based on the set of fired transitions, is responsible for the computation of the new marking, the reset orders, and the execution of functions during the rising edge phase. To prove the semantic preservation property, we must have the equivalence between the set of fired transitions as defined on the SITPN side and the set of fired transitions as defined on the $\mathcal{H}$-VHDL side. The equivalence must hold at the beginning of the rising edge phase, i.e. when the set of fired transitions will be used to compute a new SITPN state. Thus, the falling edge phase prepares the ground so that the equivalence between the set of fired transitions holds at the beginning of the next rising edge phase. To express Lemma 4, we must first define the hypotheses stating that a falling edge phase happened in the course of the execution of an SITPN and its corresponding $\mathcal{H}$-VHDL design, plus some hypotheses about the similarity of the states at the beginning of the falling edge phase:

> **Definition 45** (Falling edge hypotheses). *Given a sitpn $\in$ SITPN, $b \in P \to \mathbb{N}$, $d \in design$, $\gamma \in WM(sitpn, d)$, $E_c \in \mathbb{N} \to \mathcal{C} \to \mathbb{B}$, $\Delta \in ElDesign$, $E_p \in \mathbb{N} \to Ins(\Delta) \to value$, $\tau \in \mathbb{N}$, $s, s' \in S(sitpn)$, $\sigma_e, \sigma, \sigma_\downarrow, \sigma' \in \Sigma$, assume that:*
>
> – *SITPN sitpn is transformed into the $\mathcal{H}$-VHDL design d and yields the binder γ: $\lfloor sitpn \rfloor_b = (d, \gamma)$*

- *Simulation/Execution environments are similar:* $\gamma \vdash E_p \stackrel{env}{=} E_c$

- Δ *is the elaborated version of design d, and* σ_e *is the default design state of* Δ: $\mathcal{D}_{\mathcal{H}}, \varnothing \vdash \mathrm{d} \xrightarrow{elab} \Delta, \sigma_e$

- *Starting states are similar according to the full post rising edge similarity relation:* $\gamma, E_c, \tau \vdash s \stackrel{\uparrow}{\approx} \sigma$

- *On the SITPN side, the execution of a falling edge phase starting from state s leads to state s':* $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$

- *On the* $\mathcal{H}$*-VHDL side, the simulation of a falling edge phase starting from state* σ *leads to state* σ': $\Delta, \sigma \vdash \mathrm{d.cs} \xrightarrow{\downarrow} \sigma_{\downarrow}$ *and* $\Delta, \sigma_{\downarrow} \vdash \mathrm{d.cs} \xrightarrow{\leadsto} \sigma'$

- *State* σ *is a stable design state:* $\mathcal{D}_{\mathcal{H}}, \Delta, \sigma \vdash \mathrm{d.cs} \xrightarrow{comb} \sigma$

The hypotheses of Definition 45 are used in all the lemmas expressing properties of the falling edge phase. Therefore, Definition 45 enables the conciser expression of these lemmas. Then, we can express Lemma 4 as follows:

Lemma 4 (Falling edge equal fired). *For all sitpn, b, d,* γ, Δ, σ_e, E_c, E_p, τ, s, s', σ, $\sigma_{\downarrow}$, σ' *that verify the hypotheses of Definition 45, then for all* $t \in T, id_t \in Comps(\Delta)$ *s.t.* $\gamma(t) = id_t$, $t \in Fired(s') \Leftrightarrow \sigma'(id_t)(fired) = \mathtt{true}$.

Now, let us detail the proof of Lemma 4. To prove Lemma 4, we must reason on a given transition t of the input SITPN *sitpn* and a TCI id_t in the output $\mathcal{H}$-VHDL design d. Transition t and TCI id_t are bound together through the γ binder returned by the transformation function. This means that the TCI id_t structurally represents the transition t in the output $\mathcal{H}$-VHDL design d. In this setting, we want to prove that t is in the set of fired transitions at the end of the falling edge phase if and only if the $\mathtt{fired}$ port of id_t equals $\mathtt{true}$ at the end of the falling edge phase. Formally, we want to prove: $\boxed{t \in Fired(s') \Leftrightarrow \sigma'(id_t)(\mathtt{fired}) = \mathtt{true}.}$

As a reminder, the expression $Fired(s')$ qualifies the set of fired transitions at the SITPN state s', and $\sigma'(id_t)(\mathtt{fired})$ denotes the value of the $\mathtt{fired}$ port of TCI id_t at design state σ'. The expression $\sigma'(id_t)$ denotes the internal state, i.e. a design state, of TCI id_t at state σ'.

To prove the equivalence, we must first look at the definition of the set of fired transitions on the SITPN side and on the $\mathcal{H}$-VHDL side, and then think of a way to relate the two definitions.

On the SITPN side, the set of fired transitions is an intentional and recursive definition (see Definition 20) depending on a given SITPN state. In Lemma 4, we are interested in the definition of the set of fired transitions at state s', i.e. the state at the end of the falling edge phase. A transition belongs to the set of fired transitions if it is *firable* (see Definition 19) and sensitized by the *residual* marking (see Sections 3.1.2 and 3.2.3) at the considered SITPN state. Figure 6.4 gives the set of fired transitions, i.e. $Fired(s)$, on an example of SITPN at a given state

s. Here, transitions t_a, t_b and t_c are all firable at state *s*; however, only transition t_c is sensitized by the residual marking.
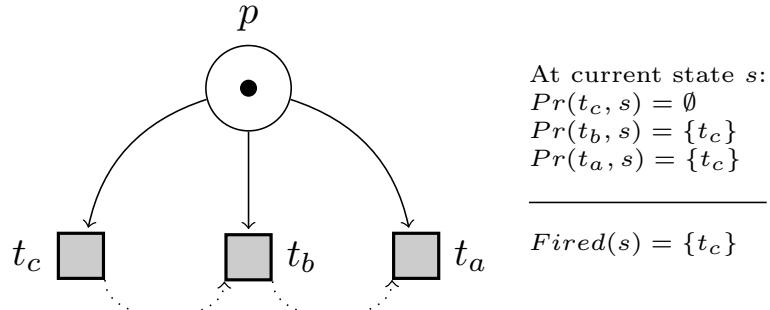
FIGURE 6.4: The set of fired transitions on an example SITPN at a given SITPN state *s*; on the left side, the dotted arrows indicates the priority relation between the three transitions (t_c is the top-priority transition); on the right side, each transition is associated to its *Pr* set, which is necessary to compute the residual marking.

The computation of the residual marking involves the *Pr* sets, which are, for a given transition *t* and a state *s*, the set of transitions with a higher firing priority than *t* which are actually fired at *s*. This is where the recursive definition of the set of fired transitions begins. The definition is correct, i.e. the recursion ends, if the priority relation is a strict order over the set of transitions, and therefore, there are always transitions of top-priority (e.g, t_c in Figure 6.4). The condition of the priority relation being a strict order over the set of transitions is part of the definition of a well-defined SITPN (see Definition 28). By definition, top-priority transitions have an empty *Pr* set. There exists no transition with a higher firing priority than a top-priority transition. Thus, a top-priority transition that is firable is also fired. Note that one cannot determine the *Pr* set of a transition before having determined the firing status of all the transitions with a higher firing priority. For instance, in Figure 6.4, it is impossible to know the content of $Pr(t_a, s)$ before having determined if transition t_b is fired or not. To know if t_b is fired or not, we must determine the content of $Pr(t_b, s)$. To do so, we must first determine the firing status of t_c. Even though the definition of the set of fired transitions is very declarative, this provides a natural way to establish an algorithm to build the set of fired transitions at a given SITPN state.

On the $\mathcal{H}$-VHDL side, the set of fired transitions is defined through the value of the `fired` port of TCIs. The transition design declares an output port of the Boolean type with the identifier `fired`. What we want to prove in Lemma 4 is that, at the end of the falling edge phase (i.e. at state σ'), the value of the `fired` port of a TCI reflects the firing status of the corresponding transition. The `fired` port is a combinational signal. This means that its value depends on an equation that is verified when all signals are stable, i.e. at the end of stabilization phases happening during the simulation. In the point of view of the circuit synthesis, this equation reflects the wiring of the port in the described hardware circuit. Figure 6.5 shows a part of the transition design architecture describing how the `fired` port is connected to the other internal signals.
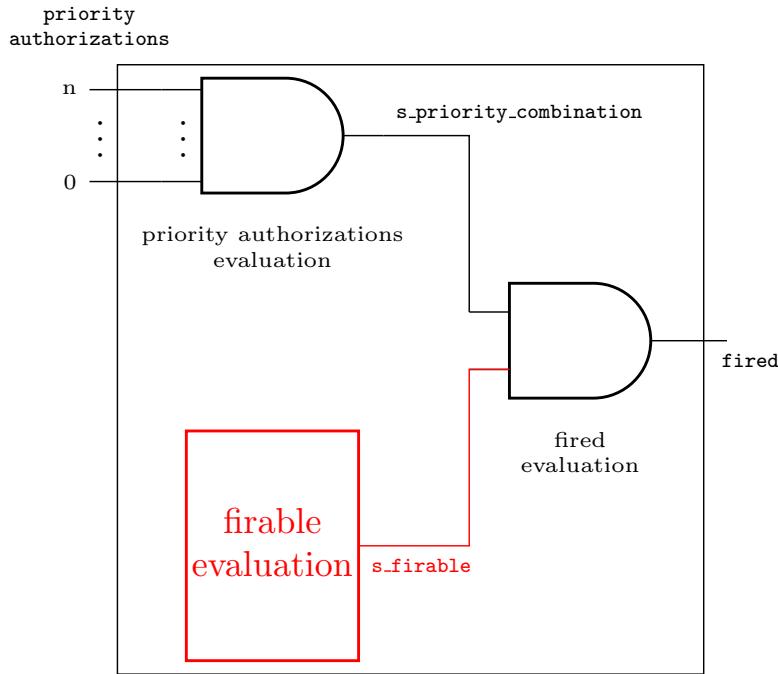
FIGURE 6.5: Wiring of the `fired` output port in the transition design architecture; on the left side is the input interface of the transition design; on the right side is the output interface of the transition design, with the `fired` port; in red are the parts of the architecture that depend on synchronous logic and in black are the parts that are purely combinational.

In Figure 6.5, the labels underneath the *and* logic ports and inside the block denote the names of the processes defined in the transition design architecture as VHDL code. As a matter of fact, Figure 6.5 is a graphical transcription of the code defining the transition design architecture. Therefore, by looking at the VHDL code, we are able to determine the combinational equation associated to the `fired` port. Given a TCI id_t in a top-level design and a state σ denoting a current stable state of the design (remember that a combinational equation holds when all signal values are stable), the `fired` port equation at σ is:

$$\sigma(id_t)(\texttt{fired}) = \sigma(id_t)(\texttt{s\_firable}) \,.\, \sigma(id_t)(\texttt{s\_priority\_combination}) \tag{6.2}$$

Equation (6.2) states that the value of the `fired` port is a simple "and" expression[2] between the value of the internal signal `s_firable` and `s_priority_combination`.

> **Remark 10** (Signals and combinational equations). *In the proceeding of the proof, a lot of combinational equations are established (e.g, Equation (6.2)). These equations relate the value of a given signal to the value of other signals or expressions. All these equations are deduced by applying the H-VHDL semantics rules on the internal behavior (i.e., the processes) of the*

---

[2]To differentiate the formulas of the first-order logic from the expressions of the Boolean logic, we use (".", "+") to denote the *and* and *or* operators in Boolean expressions, and ($\wedge,\vee$) to denote the conjunction and the disjunction in the first-order logic formulas.

transition and the place designs. A combinational equation is always the result of a signal assignment statement happening inside the statement body of a process. For instance, in the transition design, the `fired_evaluation` *process, presented in Listing 6.1, assigns the* `fired` *output port. Reasoning on the* `fired_evaluation` *process statement body and on the H-VHDL semantics rules permits us to deduce Equation (6.2).*

```
fired_evaluation : process (s_firable, s_priority_combination)
begin
    fired ⇐ s_firable and s_priority_combination;
end process fired_evaluation;
```

> LISTING 6.1: The `fired_evaluation` process in the transition design architecture; its statement body assigns the `fired` output port; symbol ⇐ is the signal assignment operator.

Listing 6.2 presents the `priority_authorizations_evaluation` *process, responsible for the assignment of the* `s_priority_combination` *in the transition design.*

```
priority_authorization_evaluation : process(priority_authorizations)
    variable v_priority_combination : std_logic;
begin
    v_priority_combination := '1';

    for i in 0 to input_arcs_number − 1 loop
      v_priority_combination := v_priority_combination and priority_authorizations(i);
    end loop;

    s_priority_combination ⇐ v_priority_combination; -- Assignment of the result
end process priority_authorization_evaluation;
```

> LISTING 6.2: The `priority_authorizations_evaluation` process in the transition design's architecture. The local variable `v_priority_-combination` accumulates the product of the subelements of the `priority_-authorizations` input port in the `for` loop; then the last statement assigns the value of `v_priority_combination` to the `s_priority_combination` internal signal.

Equation (6.3) gives the combinational equation deduced from the execution of the `priority_-authorizations_evaluation` *process for a given TCI id_t in a top-level design d. State σ denotes the current state of d, and $\sigma(id_t)$ denotes the internal state of id_t at state σ. The elaborated design Δ is the elaborated version of design d, and $\Delta(id_t)$ is the elaborated version of id_t.*

$$\sigma(id_t)(spc) = \prod_{i=0}^{\Delta(id_t)(input\_arcs\_number)-1} \sigma(id_t)(priority\_authorizations)[i] \qquad (6.3)$$

In Equation (6.3), `spc` is an alias for the `s_priority_combination` signal. The `for` loop of the `priority_authorization_evaluation` process has been converted into a product expression

where the index i follows the bounds of the loop. The `priority_authorizations` signal is an input port of type `array`, thus we use the bracketed notation a[i] to access the element of index i in array a. Also, we know that `input_arcs_number` identifies a generic constant of the transition design, thus, we can retrieve its value in the elaborated design $\Delta(id_t)$.

In the proofs laid out in Appendix D and in this chapter, we do not detail how the execution of processes' statement body permit to deduce combinational equations. We find that the proofs are easier to follow without entering in so much details. We let aside the task of proving that these equations hold until the time of the mechanization with the Coq proof assistant. For now, the reader can convince himself/herself that an equation holds by looking at the code of the `place` and the `transition` designs (see Appendices A and B).

Now that we know which combinational equation is attached to the value of the output port `fired` for a given TCI, we must relate this equation to the definition of the set of fired transitions on the SITPN side. By definition of the set of fired transitions, we know that $t \in Fired(s')$ is equivalent to $t \in Firable(s') \wedge t \in Sens(s'.M - \sum_{t_i \in Pr(t,s')} pre(t_i))$ where $Pr(t,s') = \{t' \mid t' \succ t \wedge t' \in Fired(s')\}$. By definition of the `fired` port equation, we know that $\sigma'(id_t)(\texttt{fired}) = \sigma'(id_t)(\texttt{s\_firable}) . \sigma'(id_t)(\texttt{s\_priority\_combination})$. Using these definitions to rewrite the terms of the current goal, the new goal to prove is:

$$t \in Firable(s') \wedge t \in Sens(s'.M - \sum_{t_i \in Pr(t,s')} pre(t_i)) \Leftrightarrow$$
$$\sigma'(id_t)(\texttt{s\_firable}) . \sigma'(id_t)(\texttt{s\_priority\_combination}) = \texttt{true}$$

Thanks to Lemma 41, we know that $t \in Firable(s')$ iff $\sigma'(id_t)(\texttt{s\_firable}) = \texttt{true}$. Then, we can get rid of these two terms in the current goal; what is left to prove is:

$$t \in Sens(s'.M - \sum_{t_i \in Pr(t,s')} pre(t_i)) \Leftrightarrow \sigma'(id_t)(\texttt{s\_priority\_combination}) = \texttt{true}$$

Based on Equation (6.3), we can replace the value of the `s_priority_combination` signal by its equivalent product expression:

$$t \in Sens(s'.M - \sum_{t_i \in Pr(t,s')} pre(t_i)) \Leftrightarrow$$
$$\left(\prod_{i=0}^{\Delta(id_t)(\texttt{input\_arcs\_number})-1} \sigma'(id_t)(\texttt{priority\_authorizations})[i] \right) = \texttt{true}$$

Then, the proof is in two parts:

1. Assuming $t \in Sens(s'.M - \sum_{t_i \in Pr(t,s')} pre(t_i))$, let us show that

$$\left(\prod_{i=0}^{\Delta(id_t)(\texttt{ian})-1} \sigma'(id_t)(\texttt{pauths})[i] \right) = \texttt{true}.$$

2. Assuming $\big(\prod\limits_{i=0}^{\Delta(id_t)(\mathtt{ian})-1} \sigma'(id_t)(\mathtt{pauths})[i]\big) = \mathtt{true}$, let us show that

$$\boxed{t \in Sens(s'.M - \sum_{t_i \in Pr(t,s')} pre(t_i)).}$$

Let us prove the first point. Assuming that $t \in Sens(s'.M - \sum\limits_{t_i \in Pr(t,s')} pre(t_i))$, let us show

$$\boxed{\big(\prod\limits_{i=0}^{\Delta(id_t)(\mathtt{ian})-1} \sigma'(id_t)(\mathtt{pauths})[i]\big) = \mathtt{true}.}$$

To prove the current goal, we can equivalently show that:

$$\boxed{\forall i \in [0, \Delta(id_t)(\mathtt{ian}) - 1],\ \sigma'(id_t)(\mathtt{pauths})[i] = \mathtt{true}.}$$

For a given $i \in [0, \Delta(id_t)(\mathtt{ian}) - 1]$, let us show that $\boxed{\sigma'(id_t)(\mathtt{pauths})[i] = \mathtt{true}.}$ As shown in Figure 6.5, the `priority_authorizations` signal is an input port of the `transition` design. Therefore, to know what is the value of the i-th element of the `priority_authorizations` port at state $\sigma'(id_t)$, we must know how the `priority_authorizations` port is connected in the top-level design. Basing ourselves on the transformation function (cf. Algorithm 10, p. 174), the connection of the `priority_authorizations` port for the TCI id_t depends on the set of input places of the transition t. If the set of input places of t is empty, then, all elements of the `priority_authorizations` port are connected to the constant `true`, and proving the goal is trivial. If the set of input places of t is not empty, then, the connection of the i-th element of the `priority_authorizations` port reflects the connection of some place p to the transition t by an input arc. Then, we must reason on the nature of the input arc connecting p to t. The interested case happens when p and t are connected by a `basic` arc, and when the conflicts in the output transitions of p are handled by the priority relation. In that case, the i-th element of the `priority_authorizations` input port of the TCI id_t is connected to the j-th element of the `priority_authorizations` output port of the PCI id_p. Figure 6.6 shows the connection of the `priority_authorizations` port between the component instances id_p and id_t.
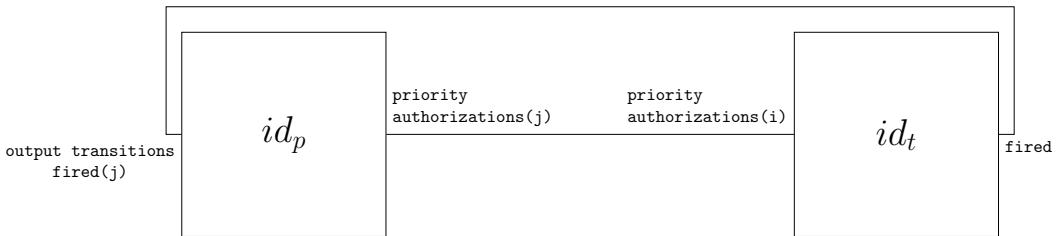
FIGURE 6.6: Connection of the j-th element of the `priority_authorizations` output port of the PCI id_p to the i-th element of the `priority_authorizations` input port of the TCI id_t; also the `fired` output port of id_t is connected to the j-th element of the `output_transitions_fired` input port of id_p.

Thus, we know that the value of the i-th element of the `priority_authorizations` input port of id_t is bound to the value of the j-th element of the `priority_authorizations` output port of id_p. Therefore, to show that $\boxed{\sigma'(id_t)(\mathtt{pauths})[i] = \mathtt{true}}$, we must now show that

$\boxed{\sigma'(id_p)(\texttt{pauths})[j] = \texttt{true.}}$ We must now look at the architecture of the `place` design to de-
termine the combinational equation associated to the *j*-th element of the `priority_authoriza-`
`tions` output port. Figure 6.7 illustrates the wiring of the `priority_authorizations` output
port in a `place` design.
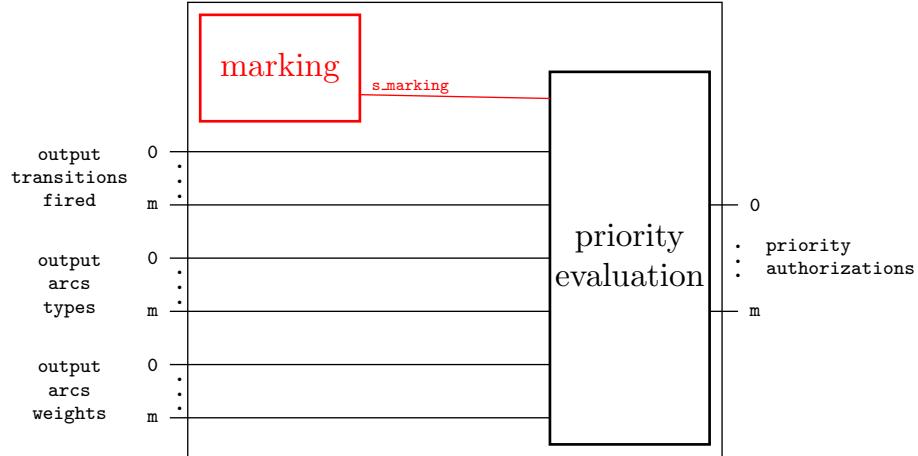
FIGURE 6.7: Wiring of the `priority_authorizations` output port in the architec-
ture of the place design; the input port interface is on the left side and the output
port interface is on the right side; the synchronous parts are in red and the combi-
national ones are in black.

Figure 6.7 shows that the value of the elements of the `priority_authorizations` output
port is computed by the `priority_evaluation` process. This process reads the value of the
`s_marking` signal, assigned by the synchronous process `marking`. It also reads the value of
the `output_transitions_fired`, `output_arcs_types` and `output_arcs_weights` input ports.
In Figure 6.7, the ports of the input and output interface are composite ports (i.e., of the array
type) with an upper bound index equal to `m`. The number `m` is equal to the expression `output_-`
`arcs_number`−1, where `output_arcs_number` is a generic constant of the `place` design. The
value of the `output_arcs_number` constant is set at the generation of the generic map of a PCI
id_p, and is equal to the number of output transitions of place *p*. Listing 6.3 presents the code of
the `priority_evaluation` process defined in the architecture of the `place` design.

```
1  priority_evaluation : process (output_transitions_fired, s_marking, output_arcs_types,
       output_arcs_weights)
2    variable v_saved_output_token_sum : local_weight_t;
3  begin
4    v_saved_output_token_sum := 0;
5
6    for k in 0 to output_arcs_number − 1 loop
7
8      priority_authorizations(k) ⇐ (s_marking − v_saved_output_token_sum >=
         output_arcs_weights(k));
9
10     if (output_transitions_fired(k) = '1') and (output_arcs_types(k) = arc_t(BASIC)) then
```

```
11        v_saved_output_token_sum := v_saved_output_token_sum + output_arcs_weights(k);
12      end if;
13
14    end loop;
15  end process priority_evaluation;
```

LISTING 6.3: The `priority_evaluation` process in the place design's architecture.

In the statement body of the `priority_evaluation` process, each subelement of the `priority_authorizations` output port is assigned at Line 8 inside the `for` loop. The statement of Line 8 assigns the result of the test `s_marking − v_saved_output_token_sum >= output_arcs_weights(k)` to the k-th element of `priority_authorizations`. The test checks that the value of the `s_marking` signal, representing the current marking of the PCI, minus the value of the local variable `v_saved_output_token` is greater than or equal to the value of the k-th element of the `output_arcs_weights` signal. The test corresponds to the test of sensitization by the residual marking for the TCI connected through index k.

Getting back to our proof, the following combinational equation holds for the j-th element of the `priority_authorizations` port at state σ':

$$\sigma'(id_p)(\texttt{pauths})[j] = (\sigma'(id_p)(\texttt{s\_marking}) - \texttt{vsots} \geq \sigma'(id)(\texttt{output\_arcs\_weights})[j]) \quad (6.4)$$

Then, rewriting the goal with Equation (6.4), the new goal is:

$$\boxed{(\sigma'(id_p)(\texttt{s\_marking}) - \texttt{vsots} \geq \sigma'(id)(\texttt{output\_arcs\_weights})[j]) = \texttt{true}.}$$

Here $\geq$ denotes a Boolean operator, i.e. $\geq \in \mathbb{N} \to \mathbb{N} \to \mathbb{B}$. As the $\geq \subseteq (\mathbb{N} \times \mathbb{N})$ relation is decidable for all pairs of natural numbers, we can interchange an expression $a \geq b = \texttt{true}$ with $a \geq b$ where $a, b \in \mathbb{N}$. We will generalize this practice to every Boolean operator having a corresponding decidable relation. Thus, the new goal is:

$$\boxed{\sigma'(id_p)(\texttt{s\_marking}) - \texttt{vsots} \geq \sigma'(id)(\texttt{output\_arcs\_weights})[j].}$$

Here, the term `vsots` corresponds to the value of the local variable `v_saved_output_token_sum` at the moment of the assignment in the `for` loop. By looking at the code of Listing 6.3 (Lines 10 to 12), we can deduce the value of the `vsots` variable:

$$\texttt{vsots} = \sum_{l=0}^{j-1} \begin{cases} \sigma'(id_p)(\texttt{oaw})[l] \text{ if } \sigma'(id_p)(\texttt{otf})[l] \,.\, (\sigma'(id_p)(\texttt{oat})[l] = \texttt{basic}) \\ 0 \text{ otherwise} \end{cases} \quad (6.5)$$

The `vsots` term is equal to the sum of the output arc weights for all TCIs, representing output transitions of p, connected through an index l comprised between 0 and $j-1$. An output arc weight is taken into account in the sum only if the TCI connected through index l has a `fired` port equals to `true` (i.e. the `output_transitions_fired` input port of id_p equals `true` at index l) and is linked to the place p through a `basic` input arc (i.e. the `output_arcs_types` input port of id_p equals `basic` at index l).

Based on the fact that all conflicts in the output transitions of place p are handled with the priority relation, then, as a property deduced from the HILECOP transformation function,

we know that the order of the indexes from 0 to `output_arcs_number` − 1 reflects the priority order of the output transitions of place p. Therefore, the indexes from 0 to $j − 1$ are linked to transitions with a higher firing priority than the transition connected to the index j. Figure 6.8 reuses the SITPN of Figure 6.4 to illustrate how the indexes are ordered when the connection between the PCI id_p and its output TCIs id_{t_a}, id_{t_b} and id_{t_c} is set (i.e., in the course of the model-to-text transformation).
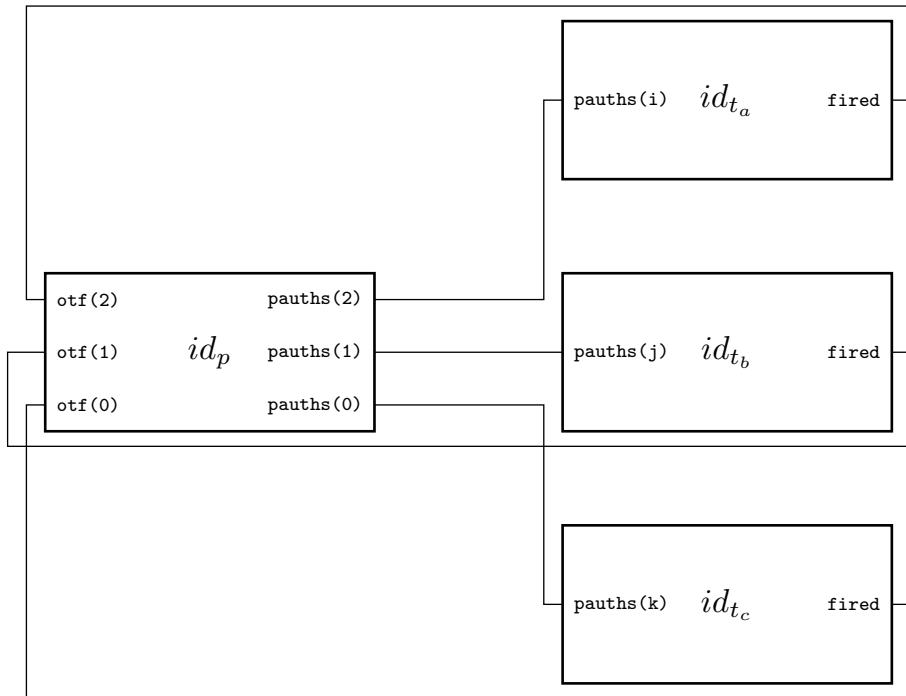
FIGURE 6.8: Connection between the `priority_authorizations` output port of PCI id_p and the `priority_authorizations` input port of TCIs id_{t_a}, id_{t_b} and id_{t_c}, and between the `output_transitions_fired` input port of id_p and the `fired` ports of id_{t_a}, id_{t_b} and id_{t_c}. `pauths` stands for `priority_authorizations` and `otf` stands for `output_transitions_fired`.

In Figure 6.8, the indexes in the interface of id_p respect the priority order of the output transitions. The index increases as the priority level of the connected TCI decreases. Thus, id_{t_c} is connected to index 0 as transition t_c is the top-priority transition in the output transitions of p, id_{t_b} is connected to index 1 as $t_c \succ t_b$, etc.

As a reminder, the current goal to prove is:

$$\boxed{\sigma'(id_p)(\texttt{s\_marking}) - \texttt{vsots} \geq \sigma'(id)(\texttt{output\_arcs\_weights})[j].}$$

The current goal is the $\mathcal{H}$-VHDL implementation of the test that the residual marking in place p enables transition t. At the beginning of the proof, we assumed that transition t is sensitized by the residual marking for all its input places, i.e. $t \in Sens(s'.M - \sum_{t_i \in Pr(t,s')} pre(t_i))$.

By looking at the definition of the *Sens* set (see Definition 18), and knowing that a basic arc of weight ω connects place p to transition t, we can deduce that $s'.M(p) - \sum_{t_i \in Pr(t,s')} pre(p, t_i) \geq \omega$. Now, we must relate the terms of the latter formula to the terms of the goal. We can easily show, appealing to Lemma 34, that $s'.M(p)$ equals $\sigma'(id_p)(\texttt{s\_marking})$. Then, by construction, and knowing that TCI id_t is connected to PCI id_p through the index j, we can deduce that the j-th element of the $\texttt{output\_arcs\_weights}$ input port denotes the weight of the arc between place p and transition t, i.e. the natural number ω. The last thing to show is the equality between the two sum terms:

$$\sum_{t_i \in Pr(t,s')} \begin{cases} \omega \text{ if } pre(p, t_i) = (\omega, \texttt{basic}) \\ 0 \text{ otherwise} \end{cases}$$
$$=$$
$$\sum_{l=0}^{j-1} \begin{cases} \sigma'(id_p)(\texttt{oaw})[l] \text{ if } \sigma'(id_p)(\texttt{otf})[l] \ . \ (\sigma'(id_p)(\texttt{oat})[l] = \texttt{basic}) \\ 0 \text{ otherwise} \end{cases}$$

On the upper part of the equality, we have unfolded term $\sum_{t_i \in Pr(t,s')} pre(t_i)$ to its full definition (see Notation 5 in Section 3.2.3). On the lower part is the full definition of the $\texttt{vsots}$ term. Let us rewrite the two sum terms in a manner that will come handy to prove the equality. Let us define the Pr_p set, which is the set of fired transitions with a higher priority than t that are conflicting with t through the place p:

$$t_i \in Pr_p \equiv t_i \succ t \wedge \exists \omega \ s.t. \ pre(p, t_i) = (\omega, \texttt{basic}) \wedge t_i \in Fired(s')$$

Let us also define the IPr_p set, which the set of indexes from 0 to $j-1$ for which the $\texttt{otf}$ port of id_p equals $\texttt{true}$ at state σ' and the $\texttt{oat}$ port of id_p equals $\texttt{true}$ at state σ':

$$l \in IPr_p \equiv l \in [0, j-1] \wedge (\sigma'(id_p)(\texttt{oat})[l] = \texttt{basic}) \wedge (\sigma'(id_p)(\texttt{otf})[l] = \texttt{true})$$

We can equivalently rewrite the goal as follows: $\boxed{\sum_{t_i \in Pr_p} \texttt{fst}(pre(p, t_i)) = \sum_{l \in IPr_p} \sigma'(id_p)(\texttt{oaw})[l]}$

In the left sum term, the $pre(p, t_i)$ always returns a couple $(\omega, \texttt{basic})$ as t_i verifies that there exists an ω such that $pre(p, t_i) = (\omega, \texttt{basic})$. Thus, the expression $\texttt{fst}(pre(p, t_i))$ denotes the first element of the couple returned by $pre(p, t_i)$, i.e. the weight ω.

Then, to prove the above equality, we must show that there exists a bijection β between Pr_p and IPr_p such that for all $t_i \in Pr_p$, we have $\texttt{fst}(pre(p, t_i)) = \sigma'(id_p)(\texttt{oaw})[\beta(t_i)]$. By property of the transformation function, we know that the order of the indexes in the $\texttt{priority\_authorizations}$ output port of id_p reflects the priority order of the conflicting output transitions of place p (see Figure 6.8). Then, we can exhibit a bijection β_0 between the output transitions of p with a higher priority than t and the indexes l of interval $[0, j-1]$ verifying $\sigma'(id_p)(\texttt{oat})[l] = basic$. However, to build a complete bijection β from Pr_p to IPr_p, we need to know that for a transition t_i that is a conflicting transition of place p with a higher priority than t, we have $t_i \in Fired(s') \Leftrightarrow \sigma'(id_p)(\texttt{otf})[\beta_0(t_i)] = \texttt{true}$. By property of the transformation function, we know that the element of index $\beta_0(t_i)$ of the $\texttt{otf}$ input port of PCI id_p is in fact

connected to the `fired` output port of TCI id_{t_i}. Thus, what we must assume to build a bijection from Pr_p to IPr_p is $t_i \in Fired(s') \Leftrightarrow \sigma'(id_{t_i})(\texttt{fired}) = \texttt{true}$. This is exactly the property of equivalence between the set of fired transitions and the value of the `fired` output ports that we are currently trying to prove.

Thus, to carry out the proof, we need a strong hypothesis stating that the equivalence between the set of fired transitions and the `fired` ports holds for all transitions with a higher firing priority than t, thus including the ones that are in conflict with t through place p. Therefore, we must think of a way to build the set of fired transitions iteratively such that the previous hypothesis becomes an invariant over the many iterations. The recursive definition of the set of fired transitions naturally calls for a proof by structural induction over the set of fired transitions. As stated before, the actual definition of the set of fired transitions is very declarative. However, we can easily convert it into an algorithm that will build the set iteratively. The result is Algorithm 15.

Algorithm 15: fired(s)

Data: An SITPN state s
Result: Returns the set of fired transitions at state s

1   $F \leftarrow \varnothing$
2   $T_s \leftarrow T$

3   **while** $T_s \neq \varnothing$ **do**
4      $t \leftarrow \texttt{GetTopPriorityTransition}(T_s, \succ)$
5   **if** $t \in Firable(s)$ **and** $t \in Sens(s.M - \sum\limits_{t_i \in Pr(t,F)} pre(t_i))$ **then** $F \leftarrow F \cup \{t\}$

6
7      $T_s \leftarrow T_s \setminus \{t\}$

8   **return** F

Algorithm 15 builds the set of fired transitions at state s by iterating over the set of transitions T. Local variables are initialized in the two first lines. Variable F carries the set of fired transitions, which is initially empty. Variable T_s represents the set of transitions still to be processed; T_s is equal to T at the beginning of the algorithm. At Line 3, the `while` loop iterates until all transitions of the T_s set have been elected to be fired or have been discarded. At Line 4, function `GetTopPriorityTransition` returns a top-priority transition of T_s, i.e. a transition t for which there exists no transition t' in T_s such that $t' \succ t$. The statement of Line 5 tests if the top-priority transition t is firable at state s and is sensitized by the residual marking computed by the expression $s.M - \sum\limits_{t_i \in Pr(t,F)} pre(t_i)$. Here, $Pr(t, F)$ is the set of transitions with the higher priority than t that are in the set F, i.e.: $Pr(t, F) = \{t_i \mid t_i \succ t \wedge t_i \in F\}$. We know that the following property holds: all fired transitions with a higher firing priority than t and that have been elected to be fired are inside the set F. Therefore, F contains all the transitions necessary to compute the residual marking that is necessary to elect the transition t as a fired transition; if t passes the test of Line 5 then it joins the set F. The statement of Line 7 withdraws the transition t from the set T_s before beginning another iteration. Because the priority relation $\succ$ is a strict order over the set of transitions T, we can always find a top-priority transition in T_s. Thus, there can be no iteration where T_s is not decreasing. Thus, the algorithm always terminates

and returns the set of fired transitions at state s.

Being more accustomed to handle relations while performing a proof, we make a relational definition of Algorithm 15 through the definition of the *IsFiredSet* and the *IsFiredSetAux* relations given in Definition 47 and 48. Definition 46 states that a given transition is fired in relation to the *IsFiredSet* relation.

> **Definition 46** (Fired). *A transition $t \in T$ is said to be fired at the SITPN state $s = <M, I, reset_t, ex, cond>$, iff there exists a subset $Fset \subseteq T$ such that $IsFiredSet(s, Fset)$ and $t \in Fset$.*

> **Definition 47** (IsFiredSet). *Given an $sitpn \in SITPN$, a SITPN state $s \in S(sitpn)$, and a subset $Fset \subseteq T$, the IsFiredSet relation is defined as follows:*
> $IsFiredSet(s, Fset) \equiv IsFiredSetAux(s, T, \emptyset, Fset)$

> **Definition 48** (IsFiredSetAux). *The IsFiredSetAux relation is defined by the following rules:*
>
> $$\text{FSetFired} \quad \frac{\begin{array}{c} t \in Firable(s) \\ t \in Sens(s.M - \sum_{t_i \in Pr(t,F)} pre(t_i)) \\ IsFiredSetAux(s, T_s, F \cup \{t\}, Fset) \end{array}}{IsFiredSetAux(s, T_s \cup \{t\}, F, Fset)} \quad \begin{array}{l} \nexists t' \in T_s \text{ s.t. } t' \succ t \\ Pr(t,F) = \{t' \mid t' \succ t \wedge t' \in F\} \end{array}$$
>
> $$\text{FSetEmp} \quad \frac{}{IsFiredSetAux(s, \emptyset, F, F)}$$
>
> $$\text{FSetNotFirable} \quad \frac{\begin{array}{c} t \notin Firable(s) \\ IsFiredSetAux(s, T_s, F, Fset) \end{array}}{IsFiredSetAux(s, T_s \cup \{t\}, F, Fset)} \quad \nexists t' \in T_s \text{ s.t. } t' \succ t$$
>
> $$\text{FSetNotSens} \quad \frac{\begin{array}{c} t \notin Sens(s.M - \sum_{t_i \in Pr(t,F)} pre(t_i)) \\ IsFiredSetAux(s, T_s, F, Fset) \end{array}}{IsFiredSetAux(s, T_s \cup \{t\}, F, Fset)} \quad \begin{array}{l} \nexists t' \in T_s \text{ s.t. } t' \succ t \\ Pr(t,F) = \{t' \mid t' \succ t \wedge t' \in F\} \end{array}$$

We are now satisfied with the definition of the set of fired transitions provided by the *IsFiredSet* relation and the *IsFiredSetAux* relation. Therefore, we give a new expression to Lemma 4 by using the *IsFiredSet* relation to qualify the set of fired transitions instead of using the first declarative definition. The result is Lemma 45.

The full formal proof of Lemma 45 is given in Section D.4 of Appendix D. The inductive definition of the *IsFiredSetAux* relation permits us to express the hypothesis that we lacked to perform the proof of Lemma 4. The hypothesis saying that for a given transition t, the "fired" equivalence holds for all transitions with a higher firing priority. This is stated in the "extra" hypothesis used in Lemma 46.

6.4.2 A report on a bug detection

In the previous section, we showed the equivalence between fired transitions and `fired` port values at the end of the falling edge phase. In a previous definition of the SITPN state, preceding the bug detection, the set of fired transitions was a member of the SITPN state record. For a given $sitpn \in SITPN$, we defined an SITPN state s by the record $s = <Fired, M, I, cond, ex>$ where *Fired* was the set of fired transitions. The *Fired* set was involved in the computation of time counter values during the falling edge phase. Thus, we needed the proof that the equivalence between the set of fired transitions and the value of the `fired` ports was effective at the beginning of the falling edge phase. In the previous SITPN semantics, the set of fired transitions stayed the same during the rising edge phase. Therefore, between two SITPN states s, s' verifying the rising edge state transition relation, i.e. $s \xrightarrow{\uparrow} s'$, we had $s.Fired = s'.Fired$. However, we showed that it wasn't the case on the $\mathcal{H}$-VHDL side, i.e. the values of the `fired` ports in TCIs would not stay the same during the rising edge phase. Thus, the equivalence fired transitions and `fired` port values at the end of the falling edge phase. The consequence was a divergence between the value of time counters and the value of the `s_time_counter` signals, both computed during the falling edge phase. Figure 6.9 shows a case of divergence between time counters and `s_time_counter` signals values in the course of an execution.
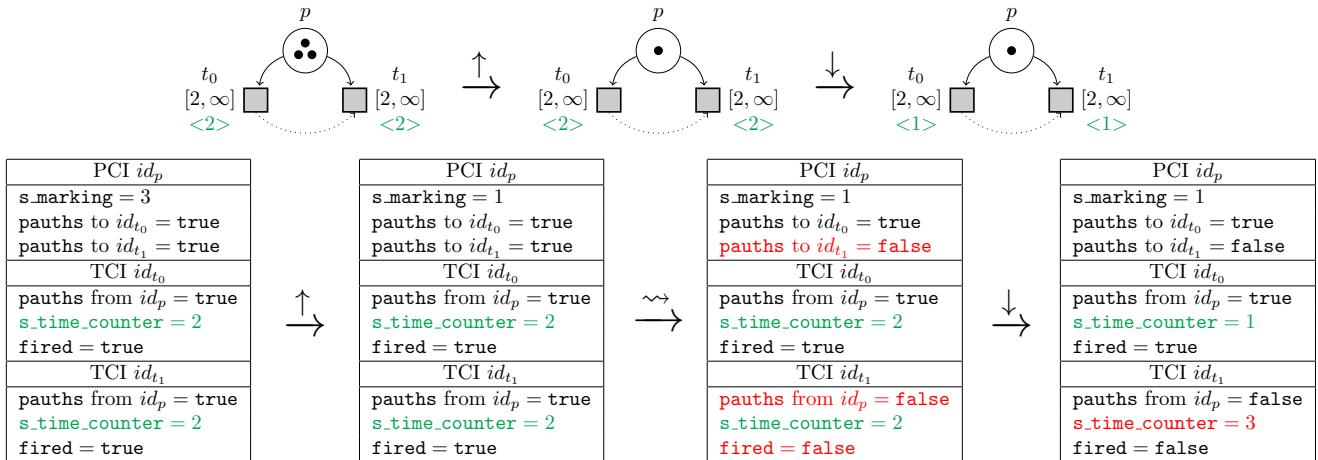
FIGURE 6.9: Bug detection: divergence between the value of time counters and the value of the `s_time_counter` signals due to the loss of the firing status information during the stabilization phase; the value of time counters and of the `s_time_counter` signals are in green; the value of diverging signals are in red.

In Figure 6.9, during the stabilization phase coming right after the rising edge of the clock, the value of the `fired` port of TCI id_{t_1} passes to `false`. After the update of the `s_marking` signal value during the rising edge phase, PCI id_p computes new priority authorizations for its output TCIs. As the marking is only sufficient to fire transition t_0 but not transition t_1, PCI id_p indicates to TCI id_{t_1} that it no longer has the authorization to fire. Consequently, through the connection of `priority_authorizations` ports, the value of the `fired` port of id_{t_1} is set to `false`. Following the rules of the SITPN semantics, on the next falling edge, the value of time

counters must be reset for transition t_0 and t_1, because both were fired at the previous rising edge. As a part of the behavior of a TCI, the `time_counter` process, executed at the falling edge of the clock, resets the value of the `s_time_counter` signal given that the value of the `fired` port is true. Thus, as the value of the `fired` port of TCI id_{t_1} is false at the falling edge, the `time_counter` process increments the value of the `s_time_counter` signal instead of resetting its value. The consequence is a divergence between the value of the time counter of transition t_1 and the value of the `s_time_counter` signal in TCI id_{t_1}.

As demonstrated above, the `time_counter` process can not rely on the value of the `fired` ports to determine if the value of the `s_time_counter` signal must be reset or not. We proved that there is an equivalence between the fired transitions and the value of the `fired` ports at the end of a falling edge phase. We need a way to memorize the value of `fired` ports at the moment where the equivalence holds (i.e. at the end of the falling edge phase) so that the `time_counter` process can use this information to reset the `s_time_counter` signal. To do so, we have modified the SITPN semantics and the behavior of the transition design. In the actual version of the SITPN semantics, if a transition is fired at the beginning of the rising edge phase then a reset order is sent to the transition. As a consequence, the time counter associated to this transition will be reset at the next falling edge. In the actual version of the transition design behavior, the value of the `fired` port is involved in the computation of the `s_reinit_time_counter` signal; the `s_reinit_time_counter` signal value follows the value of the reset order assigned to a given transition. Thus, as the equivalence between reset orders and the value of the `s_reinit_time_counter` signal holds at the beginning of the falling edge phase, the `time_counter` process can rely on the value of the `s_reinit_time_counter` signal to reset the value of the `s_time_counter` signal. As a consequence, the set of fired transitions is no longer involved in the SITPN semantics premises of the falling edge phase. Therefore, we chose to withdraw the *Fired* set from the definition of the SITPN state record. We opted for an intentional definition of the set of fired transitions at given SITPN state (i.e., Definition 20). After these changes, we were able to prove that there were no more divergence between the time counters and the value of the `s_time_counter` signals in the course of the execution (see Lemmas 28, p. 289, and 37, p. 313, about the equivalence of time counters).

6.5   Mechanized verification of the proof

The work of mechanizing the proof of Theorem 5 is an ongoing task. At the time of the writing, we have only verified the two first points (the Initial states equal marking lemma and the Initial states equal time counters lemma) of the proof concerning the Similar initial states lemma. However, the effort to achieve this little part of the overall verification amounts to three months of work. In this section, we give metrics to measure the gap between the size of the "paper" proof (see Appendix D) and the size of the computer-checked proof written in Coq. We point out some of the reasons that may explain the gap, and comment some employed techniques to reduce the size of proof scripts. As a remainder, the full code including specifications and proof scripts is available at `https://github.com/viampietro/ver-hilecop`.

Listing 6.4 presents the Coq implementation of Theorem 5 along with the sequence of tactics constituting its proof. We also declared the Behavior preservation theorem, and the Elaboration, Initialization, Trace existence theorems as axioms in the Soundness.v file under the soundness folder of the Git repository.

```
1   Theorem sitpn2hvhdl_full_trace_sim :
2     forall τ sitpn decpr id_ent id_arch E_c θ_s d E_p b θ_σ γ Δ,
3
4       (* sitpn is well-defined. *)
5       IsWellDefined sitpn →
6
7       (* sitpn translates into (d, γ). *)
8       sitpn_to_hvhdl sitpn decpr id_ent id_arch b = (inl (d, γ)) →
9
10      (* Environments are similar. *)
11      SimEnv sitpn γ E_c E_p →
12
13      (* SITPN sitpn yields execution trace θ_s after τ execution cycles. *)
14      SitpnFullExec sitpn E_c γ τ θ_s →
15
16      (* Design d yields simulation trace θ_σ after τ simulation cycles. *)
17      hfullsim E_p τ Δ d θ_σ →
18
19      (* ** Conclusion: traces are similar. ** *)
20      SimTrace γ θ_s θ_σ.
21  Proof.
22    (* Case analysis on τ *)
23    destruct τ;
24      intros *;
25      inversion_clear 4;
26      inversion_clear 1;
27
28    (* - CASE τ = 0, GOAL γ ⊢ s_0 ∼ σ_0. Solved with sim_init_states lemma.
29       - CASE τ > 0, GOAL γ ⊢ (s_0 :: s_0 :: s :: θ) ∼ (σ_0 :: σ :: σ' :: θ'').
30     Solved with [first_cycle] and [simulation] lemmas. *)
31      lazymatch goal with
32      | [ Hsimloop: simloop _ _ _ _ _ _ _ |- _ ] ⇒
33        inversion_clear Hsimloop; constructor; eauto with hilecop
34      end.
35  Qed.
```

LISTING 6.4: Coq implementation of the Full trace similarity theorem and the mechanized version of its proof.

The proof laid out in Listing 6.4 follows the structure of the informal proof of Theorem 5. First, we perform case analysis on the structure of the τ variable through the destruct tactic. Then, the intros * introduces all universally-bound variables in the proof context. Then, at Lines 25 and 26, we use a variant of the inversion tactic (i.e. inversion_clear) to unfold the

definition of the SITPN full execution relation and the $\mathcal{H}$-VHDL full simulation relations. The number passed as an argument to the `inversion_clear` tactic refers to the index of the premise in the arrow-separated list of premises constituting the declaration of the theorem. At Line 31, we perform pattern matching on the proof context and on the conclusion to be proved. This permits to identify the hypothesis associated to the $\mathcal{H}$-VHDL simulation relation, implemented in Coq by the `simloop` relation; the hypothesis of `simloop` is named `Hsimloop`. To give an example, the `simloop` relation can take the following form (cf. Section 4.6.2 for more details on the $\mathcal{H}$-VHDL simulation relation):

$$\texttt{simloop D}_s \texttt{ E}_p \; \Delta \; \sigma \; \texttt{behavior} \; \tau \; \theta$$

This hypothesis has been introduced in the context of the proof as a side effect of the inversion `tactic` used at Line 26. Then, we introduce in the proof context new hypotheses based on the definition of the `Hsimloop` hypothesis (i.e. the definition of the $\mathcal{H}$-VHDL simulation relation) by invoking `inversion_clear` tactic on `Hsimloop`. Then, the `constructor` tactic builds sub-goals to be proved based on the definition of the full trace similarity relation. We let the `eauto` tactic decide which lemma apply to solve the sub-goals generated by the `constructor` tactics. We give a hint to the `eauto` tactic so that it looks in the user-defined `hilecop` database of theorems and lemmas to solve the sub-goals. The `hilecop` database contains the Coq implementation of all the theorems and lemmas used to prove the Full trace similarity theorem.

Robustness to change

The proof laid out in Listing 6.4 is representative of our strategy to keep our mechanized proofs robust to change. The robustness criterion is important for multiple reasons. First, in the proceeding of the proof, we can always realize that some case is missing in the expression of the transformation function or discover that the semantics of the SITPNs or the $\mathcal{H}$-VHDL language is incomplete or incorrect. Therefore, we want to structure our proofs in a way that will lower the impact of correcting the transformation function or completing the semantics. Second, we know that the SITPN structure and the $\mathcal{H}$-VHDL code of the place and transition designs will be evolving in the future. Therefore, we want to be able to adapt our proofs with a minimum effort. To reach robustness to change, we follow the indications laid out in [30]. Mainly, we make an important use of the pattern matching constructs, such as `lazymatch` or `match`, to seek hypotheses in the current proof context. Also, we build hint databases and rely as much as possible on the use of the `auto` and `eauto` to solve the conclusions.

Automation

To shorten the size of proofs, we develop user-defined tactics using the Coq `Ltac` language. The tactic that most contributed to the reduction of the size of the proof scripts is the `minv` tactic (see `StateAndErrorMonadTactics.v` under the `common` folder). The `minv` tactic automates the proof of certain lemmas regarding the properties of the HILECOP transformation function in the context of the state-and-error monad. Our Coq implementation of the HILECOP transformation function implements the state-and-error monad. This monad simulates imperative language

traits into functional languages. All functions involved in the HILECOP transformation func-
tion carry a compile-time state, defined as the Coq type `CompileTimeState`. Each function either
returns a value, modifies the compile-time state or does both. To give an example of the use of
the `minv` tactic, Listing 6.5 shows the implementation of the `generate_place_comp_inst` func-
tion involved in HILECOP transformation function. The `generate_place_comp_inst` function
generates a $\mathcal{H}$-VHDL PCI statement from a place p passed as a parameter. As a side effect, the
`generate_place_comp_inst` function adds the PCI statement to the behavior of the top-level
design currently built in the compile-time state.

```
1  Definition generate_place_comp_inst (p : P sitpn) : CompileTimeState unit :=
2
3      do id        ← get_nextid;
4      do _         ← bind_place p id;
5      do pcomp     ← get_pcomp p;
6      do pcomp_inst ← HComponent_to_comp_inst id place_entid pcomp;
7      add_cs pcomp_inst.
```

LISTING 6.5: Coq implementation of the `generate_place_comp_inst` function; the
function takes an SITPN place p as a parameter, and modifies the compile-time
state without returning a value (i.e. the function return type is `unit`)

In its definition body, function `generate_place_comp_inst` sequentially calls to functions
that sometimes modify the compile-time state (e.g. the `bind_place` function adds a binding
between p and id in the generated γ binder, i.e. $\gamma(p) = id$ after the call to `bind_place`), or
sometimes simply return a value without modifying the state (e.g. `get_pcomp` returns an in-
termediate structure representing the place component instance associated to place p in the
compile-time state). During the mechanization of the proof, we often need to prove that some
properties hold between the input compile-time state and the output compile-time after the
call to a certain function. For example, after calling the `generate_place_comp_inst` function
on a given place p and for a given input state s, let us say that a new compile-time state s' is
returned. We want to show that the part of the γ binder pertaining to the binding of transitions
to TCI identifiers has not changed between state s and state s'[3]. To perform the proof, we need
to show that each function call composing the sequence of the `generate_place_comp_inst`
function returns a compile-time state verifying the wanted property. Proving simple property
like verifying that part of the compile-time states are equal through the multiple invocation of
functions is highly automatable. We adapt the tactic `monadInv` defined in the CompCert project
[72] to automate proof for such properties. The result is the tactic `minv` massively used in the
proofs pertaining to state invariants[4].

Gap between informal and formal proof

There is a huge gap of size between the informal proof of the Full trace similarity theorem
given in this Chapter and in Appendix D and the machine-checked formal proof. Right now,

---

[3]Remember that the γ binder is part of the compile-time state record type.

[4]State invariance lemmas are to be found in the `GenerateInfosInvs.v`, `GenerateArchitectureIns.v`,
`GeneratePortsInvs.v` and `GenerateHVhdlInvs.v` under the `sitpn2hvhdl` folder of the Git repository.

the Coq proof wins the size competition. The most significant distance between the size of the informal and the formal proof comes from the two following points: the statement of the combinational equations defining the value of $\mathcal{H}$-VHDL signals and the statement of properties about the HILECOP transformation function. Stating that a combinational equation holds for a given signal in the context of an informal proof is a one-line sentence. The same goes when invoking the properties of the PCIs and TCIs populating the top-level design behavior based on the definition of the transformation function. However, proving these statements represents a tremendous mechanization effort within the Coq proof assistant. To give an example, we begin the proof of Lemma 6 by taking a place p and a PCI identifier id_p linked through the γ binder returned by the transformation function. Then, we state the existence of a PCI statement, identified by id_p and with an associated generic map, input port map and output port map, in the behavior of the top-level design returned by the transformation function. To do so, we use the following the sentence:

"Let us take a $p \in P$ and an $id_p \in Comps(\Delta)$ such that $\gamma(p) = id_p$. By construction, there exist g_p, i_p, o_p s.t. $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$."

The expression "by construction" is a shorthand expression for "knowing how the target $\mathcal{H}$-VHDL design is constructed by the transformation function", "based on the definition of the transformation function", or again "by property of the transformation function". In Coq, proving the lemma that states the existence of a PCI for a given place p amounts to 1500 lines of proof script. The lemmas regarding properties of PCI and TCI statements deduced from the transformation function tend to have complicated proofs. We believe that the implementation of the HILECOP transformation function could be more straightforward in order to simplify this kind of proof. By straightforward, we mean that the number of steps separating a given place or a given transition from the generation of their corresponding PCI or TCI could be diminished, maybe at the cost of time performance. Right now, ease of proof is more important than time performance, considering that our goal is to prove the semantic preservation theorem in a reasonable amount of time. Still, the major complexity of the transformation function, i.e. what makes the proofs so hard, lies in the generation of the interconnections between PCIs and TCIs. Some engineering effort could be spent to simplify this particular of the transformation.

Also, we spent a lot of time proving some uninteresting, however necessary, properties about the $\mathcal{H}$-VHDL design states and the $\mathcal{H}$-VHDL simulation relations. For instance, we proved a lot of lemmas pertaining the preservation of identifiers through the simulation phases (e.g. if a signal identifier is present in a design state at the beginning of a stabilization phase, then it is still present at the end of the phase). We also proved a lot of uninteresting properties about the $\mathcal{H}$-VHDL elaborated designs and the $\mathcal{H}$-VHDL elaboration relation. For instance, properties on the uniqueness of identifiers in design states, in elaborated designs… We believe that a more systematic use of dependent types, especially to implement the $\mathcal{H}$-VHDL design state and the elaborated design structure, could prevent us from proving this kind of lemmas.

6.6 Conclusion

In this chapter, the aim was to present the behavior preservation theorem stating that the HILE-COP transformation is semantic preserving along with its informal proof. By presenting the

work of the literature pertaining to the verification of *transformation functions* through the proof of behavior preservation theorems, we wanted to convince the reader that the expression of our semantic preservation theorem is "correct", i.e. it follows a common expression pattern. We saw that the expression of semantic preservation theorems is quite similar in its form even when considered transformations are not of the same nature (i.e. GPL compilers, HDL compilers and model transformations). Our semantic preservation theorem takes the form of a state similarity checking between the states composing the execution traces of our source model and our target program. At each point of the execution (i.e. at each clock signal event), the state of the input model and the state of the output representation must be similar to ensure the behavior preservation property. This definition of the behavior preservation property is particular to reactive systems, i.e. we are dealing with systems that are executing over time, and that are synchronized with a clock signal. Naturally, the behavior preservation theorem must ensure that the behaviors are similar, independently of the number of execution cycles performed. Hopefully, leveraging the inductive reasoning, proving such a thing comes down to proving that behaviors are preserved through a clock cycle.

The study of the literature showed that the state comparison relation, i.e. the relation that describes how things are compared between the source and the target representation, is a significant element in the expression of the behavior preservation theorem. Especially, in our case, the state structure of the source and target representations are quite different. Indeed, we are dealing with an abstract set of data in the SITPN world, while in the $\mathcal{H}$-VHDL representation all is converted into signal values and internal states of component instances. Thus, relating these two kinds of states is not straightforward, and we thoroughly presented our state similarity relation in Section 6.2.

In this chapter, we wanted to stress another point pertaining no more to the "how" but to the "when" the states of the input and output representations must be compared in the course of the execution. Here, we are dealing with two kinds of models that are synchronously executed. However, the synchronous execution of an SITPN stays at a level that is quite abstract compared to the concrete execution of a synchronous hardware system. Indeed, the execution of a synchronous hardware system is related to the rules of the combinational and the synchronous logic, while it is not the case at the SITPN level. Thus, a $\mathcal{H}$-VHDL design goes through a lot more different states in the proceeding of a clock cycle compared to its corresponding SITPN. Figure 6.3 illustrates when the state comparison must be performed in the course of a clock cycle.

While presenting the proof of Theorem 1, we used certain theorems declared as axioms (Theorems 2, 3 and 4). These theorems express the fact that we can always derive a simulation trace from the execution of a $\mathcal{H}$-VHDL design resulting of a successful HILECOP transformation. It means that the execution of a generated $\mathcal{H}$-VHDL design never results into an error at some point of the simulation. We chose not to represent errors in the $\mathcal{H}$-VHDL semantics due to the fact that the concept of error is nonexistent in the SITPN semantics. However, we argue that proving a theorem stating the existence of a simulation trace, independently of the number of simulation cycles considered, is a way to rectify the lack of error representation in our semantics. By presenting Theorems 2, 3 and 4 as axioms, we chose to prove the theorem of semantic preservation in the case where a simulation trace has been produced for the generated $\mathcal{H}$-VHDL design. This is the setting of Theorem 5 for which the full proof is detailed in

this chapter and in Appendix D. However, we are not giving up on the proof of Theorems 2, 3 and 4. Indeed, proving a theorem stating the similarity of execution traces is useless if the execution a generated $\mathcal{H}$-VHDL design always fails at some point while the execution of the corresponding SITPN goes on. However, we are confident in the fact that if the execution of a generated $\mathcal{H}$-VHDL design fails, then it can only reflect a divergence in relation to the behavior of the input SITPN. Thus, proving that the execution traces are similar contributes to the proof that we can always derive an execution trace for a generated $\mathcal{H}$-VHDL design.

The informal "paper" proof of Theorem 5 given in this chapter and Appendix D is long; about a hundred pages. However, as we explained in Section 6.4, the strategy used in the overall proof is pretty much the same. To prove that the behavior of an SITPN and its corresponding $\mathcal{H}$-VHDL design is preserved through an execution cycle, we must reason on the execution relations ruling both worlds. But first, to relate the execution of our input and output representations, we must structurally relate the SITPN to the translated $\mathcal{H}$-VHDL design. In the proceeding of the proof, we will first reason on the structure of the input SITPN; based on the structure of the SITPN and by property of the HILECOP transformation, we can determine the structure of the top-level $\mathcal{H}$-VHDL design. Once we know the structure of the SITPN and the $\mathcal{H}$-VHDL design, we can unfold their execution rules to prove that their behaviors are the same; i.e. at the end of a computational step, states are similar.

The mechanization of the proof of Theorem 5 is at its very beginning in terms of completion. However, we have already spent three months on it. Thus, the mechanization is a very slow process. We explain the hardness of the mechanization task by pointing out the two points where the distance between informal and formal proof is most important. The first point corresponds to the statement of the construction of the $\mathcal{H}$-VHDL design based on the structure of the SITPN and the HILECOP transformation function. Reasoning on the transformation function is not an easy task as the transformation itself is not as straightforward as the transformation from a source program of a GPL to a target program of another GPL. In Section 6.5, we pointed out the distance between a property of the transformation function expressed in one sentence in the informal proof and the thousands of lines that it represents in the formal proof. The second point digging the distance between the informal and the formal proof comes from the establishment of the synchronous and combinational equations that are verified by the internal signals of the PCIs and TCIS. This also results in one sentence statement in the informal proof while representing thousands of lines of code in the formal proof. The De Bruijn factor [111], that permits to measure the distance in terms of number of characters between an informal proof and its machine-checked version (i.e. the formal program), is tremendously high in our case when considering these intermediary results.

Chapter 7

Conclusion

In this thesis, we were interested in the formal verification of the HILECOP methodology. The HILECOP methodology has been devised to assist the design and the production of safety-critical digital systems. To summarize, the HILECOP methodology proposes a high-level graphical modelling formalism; the aim of the formalism is to provide the engineers with a framework to model safety-critical digital systems in a way that foster the communication around the design models (hence the graphical aspect of the formalism). The formalism is based on component diagrams and the internal behavior of components are described with a particular kind of Petri nets. The mathematical foundations of the Petri net formalism provides the possibility to formally analyze the HILECOP high-level models, and thereby to bring the proof that the models verify certain *soundness* properties. The high-level formalism of HILECOP has been designed to be handy for the humans; however, the ultimate goal of the methodology is to obtain a physical version of the safety-critical digital system on an FPGA or ASIC. Thus, from the state of high-level model to its concrete implementation as a hardware circuit, the designed digital system goes through multiple transformations. In this thesis, we considered the formal verification of one of these transformations: the *model-to-text* transformation from a flattened Petri net version of the high-level model of the safety-critical digital system (i.e. an SITPN) to its implementation into a VHDL design. This transformation is performed by a computer program. It was our purpose to formally verify that the transformation is *semantic preserving*; that is, for any input model of the transformation the corresponding output model behaves in the same way as the input model. Pragmatically, the research question that we formalized in the introduction of this thesis was:

CAN WE PROVE THAT THE MODEL-TO-TEXT TRANSFORMATION DESCRIBED IN THE HILECOP METHODOLOGY IS SEMANTIC PRESERVING?

As pointed out in the literature reviews at the beginning of Chapters 5 and 6, the task of formally verifying that a transformation from a source representation to a target one is semantic preserving has been thoroughly studied, and in different application contexts (generic compilers, domain-specific compilers, model transformations, etc.). From this fact arises the will to compare the HILECOP model-to-text transformation to the other kinds of transformations found in the literature. In a research point of view, the complementary questions that gravitated around our main research question were:

– What are the similarities and the differences between the HILECOP model-to-text transformation and the other kind of transformations that have been formally verified?

– Are there standard techniques to prove that a transformation is semantic preserving? Do these techniques apply in the case of the HILECOP model-to-text transformation?

In other words, what elects the formal verification of the HILECOP model-to-text transformation as a concrete research task, and not as another yet interesting engineering challenge? In this thesis, the verification of the HILECOP transformation has been conducted with the help of the Coq proof assistant; thus, the relation between our formalization choices and the engineering difficulties that they brought was one topic of interest. Especially in the world of formal verification, and more truly in the domain of deductive verification and interactive proving, the mechanization of proofs with the help of a proof assistant may be very time consuming. We believe that it is a part of the research task to evaluate the feasibility of the mechanization of proofs within a reasonable time-span, and also, to try to bring an understanding regarding the formalization choices and their impacts on the mechanization.

To formally verify the semantic preservation property of the HILECOP transformation, we followed the usual proceeding applied in the domain of deductive verification, which is:

1. Formalize the execution semantics of the source representation (Chapter 3) and the target representation (Chapter 4), and implement them using the proof assistant.

2. Describe and implement the transformation within the proof assistant (Chapter 5).

3. Express the theorem of semantic preservation, prove the theorem, and mechanize the proof using the chosen proof assistant (Chapter 6).

Following these steps, we brought the proof that the HILECOP transformation is semantic preserving. Even though the mechanization of the proof is not completed, each step of the approach brought its own part of contributions.

During the formalization of the SITPN semantics, i.e. the source representation of the transformation, we helped clarify the semantics of these models, even though it was established in two previous theses [71, 78]. Especially, we formalized the concept of a well-defined SITPN, that is, an *acceptable* model for the transformation. As a matter of fact, the HILECOP transformation raises errors if the input SITPN model is not well-defined. Also, all the theorems and lemmas that we proved rely on the well-definition condition of the input SITPN model. Through the formalization of a well-defined SITPN, we precisely characterized the way to handle a conflict between the transitions of an SITPN. This aspect of the SITPN semantics is complex and has been one particularly subtle point of the proof of semantic preservation.

The reflection around the formal semantics of the VHDL language and how it could ease the reasoning around the HILECOP's VHDL programs also brought new contributions. From the rather complex semantics of the VHDL, and all is protean expressions found in the literature, we devised a simple simulation algorithm, and formalized it into a simulation relation for the execution of synchronous VHDL designs. We defined the abstract syntax of a subset of VHDL, called $\mathcal{H}$-VHDL, that suited our needs regarding the VHDL programs generated by the transformation and the ones that were previously defined by the methodology (i.e. the `place` and `transition` designs). However, the $\mathcal{H}$-VHDL syntax and semantics is well-suited to express any kind of synchronous and synthesizable digital systems. Moreover, the implementation

of the $\mathcal{H}$-VHDL syntax and semantics in Coq provides a framework to reason about $\mathcal{H}$-VHDL designs. To the best of our knowledge, it is the only example of implementation of the VHDL language using the Coq proof assistant.

About the expression of the HILECOP model-to-text transformation itself, the contribution of this thesis was the design of the algorithm of the transformation, which had never been expressed before, and its implementation within the Coq proof assistant. For the implementation of the transformation, we tried as much as possible to draw our inspirations from the techniques found in the literature. Especially, we tried to produce clear, maintainable code, through the use of functional design patterns, while anticipating the additions of new elements in the input models. We also tried as much as possible to implement the transformation in order to ease the mechanization of the proof of semantic preservation.

The last part of the work was related to the expression and the proof of the semantic preservation theorem. While expressing the theorem of semantic preservation, we formalized the way to compare an input SITPN model with the corresponding VHDL design, result of the transformation. This point is the angular stone of the theorem of semantic preservation, moreover considering that the gap between the source and the target representations is substantial. The major contribution of the thesis to this part of the work is, of course, to have brought the proof of the semantic preservation theorem, more especially the proof of Theorem 5. Although the mechanization of the proof is far from being completed, establishing the informal proof that the HILECOP model-to-text transformation is semantic preserving has revealed a bug in the VHDL implementation of the `place` and `transition` design (cf. Section 6.4.2).

We stated above that from our very pragmatic research question arose a lot of additional questions. These questions pertain to the position of the HILECOP model-to-text transformation with respect to the other examples of transformation for which a work of formal verification has been realized. In other words, what makes this transformation specific? What aspects of this transformation and of its formal verification motivate a research interest? The very context of the design and production of critical digital systems, in which the HILECOP transformation is involved, brings out interesting research challenges. In terms of semantics, it means that we are dealing with *reactive* systems, i.e. systems which are characterized by a time-related execution and their interactions with an outside environment. Considering the work done on the formal verification of compilers for GPLs, where programs are *transformational* systems (i.e. there is a one-time end-to-end execution of the program), this already constitutes an originality. The reactivity of systems must be taken into account in the expression of the theorem of semantic preservation. However, some works [19, 56, 22, 114, 76, 58, 18, 115] have already been conducted on the formal verification of hardware description language (HDL) compilers. In that case, the source language and its semantics permit us to describe reactive systems. As there exist a lot of works pertaining to the formal verification of transformations relating a source programming *language* to a target one, the first originality of the HILECOP transformation is that the source representation is a *graphical formalism*. This graphical formalism is a particular brand of Petri nets with a synchronous semantics, which is also an original point as most of the Petri net classes have an asynchronous semantics. These SITPNs have been designed to give as much power of expression as possible to the engineers that are designing safety-critical digital systems. Blending these considerations with the context of formal methods, and the necessity to produce *safe* models of critical digital systems, the result is that the semantics of SITPNs is

rather complex; especially the handling of conflicts between transitions.

Another point of originality of the HILECOP transformation comes from the distance between the SITPN models, which are yet abstract mathematical objects, and their target representation as VHDL designs, which already deeply tied to the functioning of hardware systems. Moreover, two designs are at the base of the representation of SITPN models into VHDL programs: the `place` and `transition` designs. These two designs represent more or less of a hundred lines of VHDL code each. The VHDL code describing the behavior of the `place` and `transition` designs comes with a lot of implementation-related particularities that are sometimes difficult to relate to the semantics of SITPNs (and sometimes impossible to relate at all, hence the bug detection, cf. Section 6.4.2).

Finally, the HILECOP transformation function is itself rather complex. It cannot be expressed by rules following the inductive structure of the abstract syntax of a source programming language, as it is usually done in compilers. Specifically because of the nature of the SITPN structure, the HILECOP transformation has to cover a lot of particular cases related to the form of the input models. The specificities of the HILECOP transformation function relate to the difficulties that we encountered to mechanize the proof of semantic preservation.

Although the proof of semantic preservation has been established as a *semi-formal* paper proof, we were not able to fully mechanize it within the Coq proof assistant; at least not in the time span of the thesis. This has brought a lot of thinking about the reasons surrounding the difficulties of the mechanization, and also about the solutions that would allow us to go over these difficulties. Specifically, we were wondering if the mechanization could not be brought out entirely because of a lack of engineering skills or because of other considerations. These considerations included:

– The complexity of the $\mathcal{H}$-VHDL semantics: during the mechanization of the proof, we realized that the $\mathcal{H}$-VHDL semantics, and especially the rules related to the simulation loop, although much more simplified than the complete simulation loop of the VHDL LRM, was not convenient to reason about the VHDL designs generated by the transformation, nor to reason about the `place` and `transition` design behaviors. Therefore, some changes have been made in the $\mathcal{H}$-VHDL semantics and the final result has been presented in Chapter 4. However, at the moment of the writing, we have not yet measured the impact of these changes on the mechanization of the proof.

– The complexity of the source models: one solution to be able to complete the mechanization could have been to consider an even smaller subset for the source PNs models. For instance, we could have let aside the time and interpretation aspects in SITPNs.

– The implementation of the transformation function: the current implementation of the transformation function corresponds to the implementation of a former version of the transformation algorithm. A new and simpler version of the transformation algorithm has been presented in Chapter 5. The current implementation of the transformation includes some intermediate steps, between the input model and the final $\mathcal{H}$-VHDL design, that might not be necessary and add further complexities at the time of proofs.

– The bootstrap cost of the mechanization task: at the beginning of the mechanization, a lot of intermediary lemmas must be proved that will later be extensively used in other proofs. The

consequence is that the overall completion of the mechanization advances very slowly at the beginning of the project because a lot of little bricks must be set. Eventually, the verification goes much faster when all the necessary tools are in place (notably thanks to the `auto` tactic of the Coq proof assistant, and the hint databases system).

Pondering all these considerations, it remains clear that the HILECOP methodology is an industrial product with all its subtleties. Our guess is that, to complete the mechanization of the proof, it will require one person (already acquainted with the overall system) doing the job at full time during one year. However, we are confident in the fact that we cleared the way enough for the proof of semantic preservation to be fully mechanized; now, it is only a question of human and time resources to complete it.

Once the mechanization of the proof will be completed, we will have the formal proof that the HILECOP model-to-text transformation is semantic preserving. Then, this formal proof can help to certify the HILECOP methodology as an eligible methodology for the design and production of safety-critical digital systems. The Neurinnov company exploits the HILECOP methodology for the production of neuroprostheses, which are considered highly critical medical devices. To certify the neuroprostheses as eligible for market, the UE regulation asks for the thorough testing of all the programs involved in the production chain, thus including the HILECOP methodology. For the moment, the UE regulation standards for medical devices do not consider a formal proof as sufficient to certify a given program. All the standards in the domain of avionics, railways, power plants and many others consider a formal proof as sufficient. Therefore, we are confident in the fact that the UE regulation standards for medical devices will soon evolve in this way.

7.1 Future work and perspectives

In the immediate future, the first work to complete is of course the mechanization of the proof of Theorem 5. Then, the proofs of Theorems 2, 3 and 4, which are actually considered as axioms, must be completed as well. Finally, we must take into account all the aspects of the HILECOP high-level models, which actually have a richer structure than the one presented in Chapter 3.

The first aspect to reconsider in the definition of the SITPN structure is *interpretation*. The interpretation aspect has been simplified in the actual version of SITPNs. We could at least consider the set of VHDL signals, i.e. the variables of the interpretation, as being a part of the SITPN structure. Depending on the precision level we want to achieve, we can relate the conditions to concrete Boolean expressions, and the actions/functions to concrete operations. Moreover, we can equip the SITPN semantics with an operational semantics to evaluate the expressions and operations performed over VHDL signals. In that case, the SITPN state would include a mapping between the set of VHDL signals and their current value.

The two other main aspects still to be integrated are *macroplaces* and *multi-clock domains*. The macroplace mechanism, illustrated in Figure 7.1, enables the encapsulation of an SITPN subnet, called a refinement, into an environment that handles exceptions.
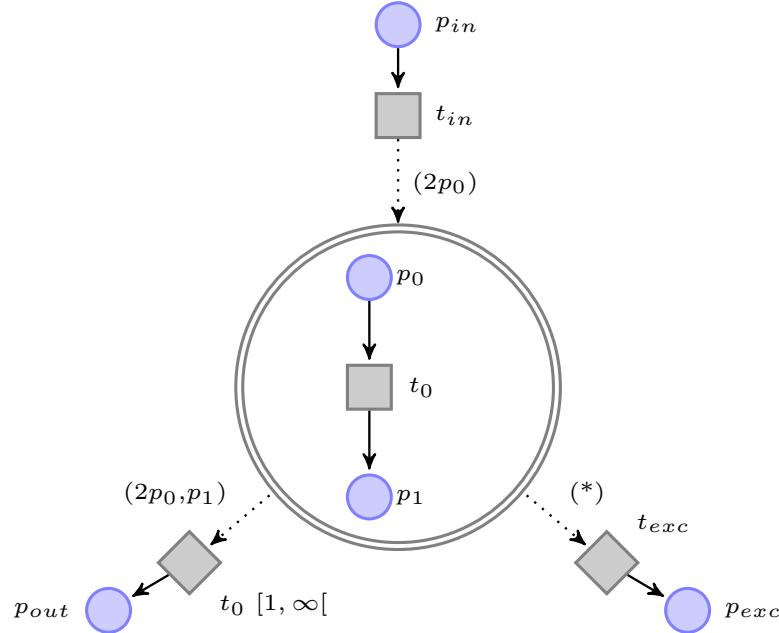
FIGURE 7.1: The macroplace is the double-lined circle that encapsulates an SITPN subnet; the subnet is called a *refinement*. The arcs that enter and go out of the macroplace are particular arcs, thus with a particular semantics, represented by dotted arrows.

The formal definition of the SITPN structure with macroplaces and its formal semantics have been described in [71]. Adding macroplaces to the actual SITPN structure will impact the transformation function, and all the surrounding proofs. It will also bring a new $\mathcal{H}$-VHDL design (i.e. the one defining macroplace components) in the loop, and will modify the code of the `place` and `transition` designs.

In the actual semantics of SITPNs, we considered that only one clock signal regulated the evolution of the system. However, the formalism of the HILECOP high-level models includes the possibility to assign different clock *domains* to different parts of the same input model. Thus, the modeled system is qualified as a multi-clock domain system. It means that the different parts of the system are not evolving at the same pace. Therefore, a mechanism of *asynchronous* message sending relates two parts with different clock domains, and allows these parts to communicate together. The system is said to have a Globally Asynchronous Locally Synchronous (GALS) architecture. The semantics of SITPNs that integrate multi-clock domains has not been formalized yet. The multi-clock domain aspect also implies modifying the $\mathcal{H}$-VHDL semantics to integrate multiple clock signals in the simulation loop.

The Coq proof assistant provides a way to extract OCaml or Haskell code from a Coq function. Thus, proving that our Coq implementation of the HILECOP model-to-text transformation is semantic preserving would allow us to extract a sound OCaml or Haskell program out of it. This program could then replace the existing Java implementation of the HILECOP methodology. At least, the engineers in charge of maintaining the current HILECOP implementation are open to the idea. However, in the certification standards of safety-critical software programs, the

executed code may not be the one over which the formal verification has been conducted (e.g. EAL 7 in the Common Criterion standard).

All the programming tasks of this thesis have been performed within the framework of the Coq proof assistant. The produced code is fully accessible under the following Git repository:

https://github.com/viampietro/ver-hilecop

Appendix A

The place design in concrete and abstract VHDL syntax

```vhdl
1   entity place is
2     generic(
3       input_arcs_number : natural := 1;
4       output_arcs_number : natural := 1;
5       maximal_marking : natural := 1
6     );
7     port(
8       clock : in std_logic;
9       reset_n : in std_logic;
10      initial_marking : in natural range 0 to maximal_marking;
11      input_arcs_weights : in weight_vector_t(input_arcs_number1 downto 0);
12      output_arcs_types : in arc_vector_t(output_arcs_number1 downto 0);
13      output_arcs_weights : in weight_vector_t(output_arcs_number1 downto 0);
14      input_transitions_fired : in std_logic_vector(input_arcs_number1 downto 0);
15      output_transitions_fired : in std_logic_vector(output_arcs_number1 downto 0);
16      output_arcs_valid : out std_logic_vector(output_arcs_number1 downto 0);
17      priority_authorizations : out std_logic_vector(output_arcs_number1 downto 0);
18      reinit_transitions_time : out std_logic_vector(output_arcs_number1 downto 0);
19      marked : out std_logic
20    );
21  end place;
22
23  architecture place_architecture of place is
24
25    subtype local_weight_t is natural range 0 to maximal_marking;
26
27    signal s_input_token_sum : local_weight_t;
28    signal s_marking : local_weight_t;
29    signal s_output_token_sum : local_weight_t;
30
31  begin
32
```

```
33    input_tokens_sum : process(input_arcs_weights, input_transitions_fired)
34       variable v_internal_input_token_sum : local_weight_t;
35    begin
36      v_internal_input_token_sum := 0;
37
38      for i in 0 to input_arcs_number − 1 loop
39        if (input_transitions_fired(i) = '1') then
40          v_internal_input_token_sum := v_internal_input_token_sum + input_arcs_weights(i
                );
41        end if;
42      end loop;
43
44      s_input_token_sum ⇐ v_internal_input_token_sum;
45    end process input_tokens_sum;
46
47    output_tokens_sum : process(output_arcs_types, output_arcs_weights,
          output_transitions_fired)
48       variable v_internal_output_token_sum : local_weight_t;
49    begin
50      v_internal_output_token_sum := 0;
51
52      for i in 0 to output_arcs_number − 1 loop
53        if (output_transitions_fired(i) = '1' and output_arcs_types(i) = arc_t(BASIC)) then
54          v_internal_output_token_sum := v_internal_output_token_sum +
                output_arcs_weights(i);
55        end if;
56      end loop;
57
58      s_output_token_sum ⇐ v_internal_output_token_sum;
59    end process output_tokens_sum;
60
61    marking : process(clock, reset_n, initial_marking)
62    begin
63      if (reset_n = '0') then
64        s_marking ⇐ initial_marking;
65      elsif rising_edge(clock) then
66        s_marking ⇐ s_marking + (s_input_token_sum − s_output_token_sum);
67      end if;
68    end process marking;
69
70    determine_marked : process(s_marking)
71    begin
72      if (s_marking = 0) then
73        marked ⇐ '0';
74      else
75        marked ⇐ '1';
76      end if;
```

```vhdl
77      end process determine_marked;

78

79      marking_validation_evaluation : process(output_arcs_types, output_arcs_weights,
            s_marking)
80      begin
81        for i in 0 to output_arcs_number − 1 loop
82          if ( ( (((output_arcs_types(i) = arc_t(BASIC)) or (output_arcs_types(i) = arc_t(TEST)))
                and (s_marking >= output_arcs_weights(i)) )
83              or ( (output_arcs_types(i) = arc_t(INHIBITOR)) and (s_marking <
                    output_arcs_weights(i)) ) )
84          then
85            output_arcs_valid(i) ⇐ '1';
86          else
87            output_arcs_valid(i) ⇐ '0';
88          end if;
89        end loop;
90      end process marking_validation_evaluation;

91

92      priority_evaluation : process(output_arcs_types, output_arcs_weights,
            output_transitions_fired, s_marking)
93        variable v_saved_output_token_sum : local_weight_t;
94      begin
95        v_saved_output_token_sum := 0;

96

97        for i in 0 to output_arcs_number − 1 loop
98          if (s_marking >= v_saved_output_token_sum + output_arcs_weights(i)) then
99            priority_authorizations(i) ⇐ '1';
100         else
101           priority_authorizations(i) ⇐ '0';
102         end if;

103

104         if ((output_transitions_fired(i) = '1') and (output_arcs_types(i) = arc_t(BASIC)))
                then
105           v_saved_output_token_sum := v_saved_output_token_sum + output_arcs_weights(i);
106         end if;

107

108       end loop;
109     end process priority_evaluation;

110

111     reinit_transitions_time_evaluation : process(clock, reset_n)
112     begin
113       if (reset_n = '0') then
114         reinit_transitions_time ⇐ (others ⇒ '0');
115       elsif rising_edge(clock) then
116         for i in 0 to output_arcs_number − 1 loop
117           if ( (((output_arcs_types(i) = arc_t(BASIC)) or (output_arcs_types(i) = arc_t(TEST)))
118                 and (s_marking − s_output_token_sum < output_arcs_weights(i))
```

```
119            and (s_output_token_sum > 0))
120          or output_transitions_fired(i) = '1' )
121      then
122        reinit_transitions_time(i) ⇐ '1';
123      else
124        reinit_transitions_time(i) ⇐ '0';
125      end if;
126    end loop;
127  end if;
128  end process reinit_transitions_time_evaluation;
129
130 end place_architecture;
```

LISTING A.1: The `place` design in concrete VHDL syntax.

```
1  design place place_architecture
2
3  -- Generic clause
4  ((input_arcs_number, natural(0,NATMAX), 1),
5   (output_arcs_number, natural(0,NATMAX), 1),
6   (maximal_marking, natural(0,NATMAX), 1))
7
8  -- Port clause
9  ((in, initial_marking, natural(0, maximal_marking)),
10  (in, input_arcs_weights, array (natural(0, 255), 0, input_arcs_number−1)),
11  (in, output_arcs_types, array (natural(0, 2), 0, output_arcs_number−1)),
12  (in, output_arcs_weights, array (natural(0, 2), 0, output_arcs_number−1)),
13  (in, input_transitions_fired, array (boolean, 0, input_arcs_number−1)),
14  (in, output_transitions_fired, array (boolean, 0, output_arcs_number−1)),
15  (out, output_arcs_valid, array (boolean, 0, output_arcs_number−1)),
16  (out, priority_authorizations, array (boolean, 0, output_arcs_number−1)),
17  (out, reinit_transitions_time, array (boolean, 0, output_arcs_number−1)),
18  (out, marked, boolean))
19
20  -- Architecture declarative part
21  ((s_input_token_sum, natural(0, maximal_marking)),
22   (s_marking, natural(0, maximal_marking)),
23   (s_output_token_sum, natural(0, maximal_marking)))
24
25  -- Behavior
26  process (input_tokens_sum, (input_arcs_weights, input_transitions_fired),
27          ((v_internal_input_token_sum, natural(0, maximal_marking))),
28  (v_internal_input_token_sum := 0;
29   (for (i, 0, input_arcs_number − 1)
30      (if (input_transitions_fired(i) = true)
31         (v_internal_input_token_sum := v_internal_input_token_sum +
                input_arcs_weights(i))));
```

```
32   s_input_token_sum ⇐ v_internal_input_token_sum))

33

34   process (output_tokens_sum,
35          (output_arcs_types, output_arcs_weights, output_transitions_fired),
36          ((v_internal_output_token_sum, natural(0, maximal_marking))),
37   (v_internal_output_token_sum := 0;
38    (for (i, 0, output_arcs_number−1)
39         (if (output_transitions_fired(i) = true and output_arcs_types(i) = 0)
40             (v_internal_output_token_sum := v_internal_output_token_sum +
41                 output_arcs_weights(i))));
41   s_output_token_sum ⇐ v_internal_output_token_sum))

42

43   process (marking, (clk, initial_marking), ∅
44   (rst (s_marking ⇐ initial_marking)
45        (rising (s_marking ⇐ s_marking + (s_input_token_sum − s_output_token_sum))))

46

47   process (determine_marked, (s_marking), ∅, (marked ⇐ s_marking > 0))

48

49   process (marking_validation_evaluation,
50           (output_arcs_types, output_arcs_weights, s_marking), ∅,
51   (for (i, 0, output_arcs_number − 1)
52       (output_arcs_valid(i) ⇐
53        ((((output_arcs_types(i) = 0) or (output_arcs_types(i) = 1))
54           and (s_marking >= output_arcs_weights(i)))
55         or ((output_arcs_types(i) = 2) and (s_marking < output_arcs_weights(i)))))))

56

57   process(priority_evaluation,
58           (output_arcs_types, output_arcs_weights, output_transitions_fired, s_marking),
59           ((v_saved_output_token_sum, natural(0, maximal_marking))),
60   (v_saved_output_token_sum := 0;
61    (for (i, 0, output_arcs_number − 1)
62        (priority_authorizations(i) ⇐
63         (s_marking >= v_saved_output_token_sum + output_arcs_weights(i))));
64   (if ((output_transitions_fired(i) = true) and (output_arcs_types(i) = 0))
65       (v_saved_output_token_sum := v_saved_output_token_sum + output_arcs_weights(i)))))

66

67   procees(reinit_transitions_time_evaluation, (clk), ∅,
68   (rst
69   (for (i, 0, output_arcs_number−1) (reinit_transitions_time(i) ⇐ false))
70   (rising
71    (for (i, 0, output_arcs_number−1)
72         (reinit_transitions_time(i) ⇐
73          (((output_arcs_types(i) = 0) or (output_arcs_types(i) = 1))
74            and (s_marking − s_output_token_sum < output_arcs_weights(i))
75            and (s_output_token_sum > 0))
76           or (output_transitions_fired(i) = true)))))
```

LISTING A.2: The `place` design in $\mathcal{H}$-VHDL abstract syntax.

Appendix B

The transition design in concrete and abstract VHDL syntax

```vhdl
entity transition is
  generic(
    transition_type : transition_t := NOT_TEMPORAL;
    input_arcs_number : natural := 1;
    conditions_number : natural := 1;
    maximal_time_counter : natural := 1
    );
  port(
    clock : in std_logic;
    reset_n : in std_logic;
    input_conditions : in std_logic_vector(conditions_number1 downto 0);
    time_A_value : in natural range 0 to maximal_time_counter;
    time_B_value : in natural range 0 to maximal_time_counter;
    input_arcs_valid : in std_logic_vector(input_arcs_number1 downto 0);
    reinit_time : in std_logic_vector(input_arcs_number1 downto 0);
    priority_authorizations : in std_logic_vector(input_arcs_number1 downto 0);
    fired : out std_logic
    );
end transition;

architecture transition_architecture of transition is

  signal s_condition_combination : std_logic;
  signal s_enabled : std_logic;
  signal s_firable : std_logic;
  signal s_firing_condition : std_logic;
  signal s_priority_combination : std_logic;
  signal s_reinit_time_counter : std_logic;
  signal s_time_counter : natural range 0 to maximal_time_counter;

begin

```

```vhdl
33   condition_evaluation : process(input_conditions)
34     variable v_internal_condition : std_logic;
35   begin
36     v_internal_condition := '1';
37
38     for i in 0 to conditions_number - 1 loop
39       v_internal_condition := v_internal_condition and input_conditions(i);
40     end loop;
41
42     s_condition_combination <= v_internal_condition;
43   end process condition_evaluation;
44
45   enable_evaluation : process(input_arcs_valid)
46     variable v_internal_enabled : std_logic;
47   begin
48     v_internal_enabled := '1';
49
50     for i in 0 to input_arcs_number - 1 loop
51       v_internal_enabled := v_internal_enabled and input_arcs_valid(i);
52     end loop;
53
54     s_enabled <= v_internal_enabled;
55   end process enable_evaluation;
56
57   reinit_time_counter_evaluation : process(reinit_time, s_enabled)
58     variable v_internal_reinit_time_counter : std_logic;
59   begin
60     v_internal_reinit_time_counter := '0';
61
62     for i in 0 to input_arcs_number - 1 loop
63       v_internal_reinit_time_counter := v_internal_reinit_time_counter or reinit_time(i
             );
64     end loop;
65
66     s_reinit_time_counter <= v_internal_reinit_time_counter;
67   end process reinit_time_counter_evaluation;
68
69   time_counter : process(reset_n, clock)
70   begin
71     if (reset_n = '0') then
72       s_time_counter <= 0;
73     elsif falling_edge(clock) then
74       if ((s_enabled = '1') and (transition_type /= transition_t(NOT_TEMPORAL))) then
75         if (s_reinit_time_counter = '0') then
76           if (s_time_counter < maximal_time_counter) then
77             s_time_counter <= s_time_counter + 1;
78           end if;
```

```vhdl
79          else
80            s_time_counter ⇐ 1;
81          end if;
82        else
83          s_time_counter ⇐ 0;
84        end if;
85      end if;
86    end process time_counter;
87
88    firing_condition_evaluation : process (s_enabled, s_condition_combination,
          s_reinit_time_counter, s_time_counter)
89    begin
90      if ( (s_condition_combination = '1')
91          and (s_enabled = '1')
92          and ( (transition_type = transition_t(NOT_TEMPORAL))
93              or ( (transition_type = transition_t(TEMPORAL_A_B))
94                  and (s_reinit_time_counter = '0')
95                  and (s_time_counter >= (time_A_value1))
96                  and (s_time_counter < time_B_value))
97              or ( (s_reinit_time_counter = '0')
98                  and ( ((transition_type = transition_t(TEMPORAL_A_A))
99                        and (s_time_counter = (time_A_value1)))
100                     or ( (transition_type = transition_t(TEMPORAL_A_INFINITE))
101                         and (s_time_counter >= (time_A_value1)) ))
102              or ( (transition_type /= transition_t(NOT_TEMPORAL))
103                  and (s_reinit_time_counter = '1')
104                  and (time_A_value = 1) )
105                )
106      ) then
107        s_firing_condition ⇐ '1';
108      else
109        s_firing_condition ⇐ '0';
110      end if;
111    end process firing_condition_evaluation;
112
113    priority_authorization_evaluation : process(priority_authorizations)
114      variable v_priority_combination : std_logic;
115    begin
116      v_priority_combination := '1';
117
118      for i in 0 to input_arcs_number − 1 loop
119        v_priority_combination := v_priority_combination and priority_authorizations(i);
120      end loop;
121
122      s_priority_combination ⇐ v_priority_combination;
123    end process priority_authorization_evaluation;
124
```

```
125    firable : process(reset_n, clock)
126    begin
127      if (reset_n = '0') then
128        s_firable ⇐ '0';
129      elsif falling_edge(clock) then
130        s_firable ⇐ s_firing_condition;
131      end if;
132    end process firable;
133
134    fired_evaluation : process (s_firable, s_priority_combination)
135    begin
136      fired ⇐ s_firable and s_priority_combination;
137    end process fired_evaluation;
138
139  end transition_architecture;
```

LISTING B.1: The transition design in concrete VHDL syntax.

```
1   design transition transition_architecture
2
3   -- Generic clause
4   ((transition_type, natural(0,3), 0),
5    (input_arcs_number, natural(0,NATMAX), 1),
6    (conditions_number, natural(0,NATMAX), 1),
7    (maximal_time_counter, natural(0,NATMAX),1))
8
9   -- Port clause
10  ((in, input_conditions, array(boolean, 0, conditions_number−1),
11   (in, time_A_value, natural(0, maximal_time_counter)),
12   (in, time_B_value, natural(0, maximal_time_counter)),
13   (in, input_arcs_valid, array(boolean, 0, input_arcs_number−1)),
14   (in, reinit_time, array(boolean, 0, input_arcs_number−1)),
15   (in, priority_authorizations, array(boolean, 0, input_arcs_number−1)),
16   (out, fired, boolean))
17
18  -- Architecture declarative part
19  ((s_condition_combination, boolean),
20   (s_enabled, boolean),
21   (s_firable, boolean),
22   (s_firing, boolean),
23   (s_priority, boolean),
24   (s_reinit, boolean),
25   (s_time_counter, natural(0, maximal_time_counter)),
26
27  -- Behavior
28
29  process (condition_evaluation, (input_conditions), ((v_internal_condition, boolean)),
```

```
30  (v_internal_condition := true;
31   (for (i, 0, conditions_number−1)
32        (v_internal_condition := v_internal_condition and input_conditions(i)));
33    s_condition_combination ⇐ v_internal_condition))
34
35  process (enable_evaluation, (input_arcs_valid), ((v_internal_enabled, boolean)),
36  (v_internal_enabled := true;
37   (for (i, 0, input_arcs_number−1)
38        (v_internal_enabled := v_internal_enabled and input_arcs_valid(i)));
39    s_enabled ⇐ v_internal_enabled))
40
41  process (reinit_time_counter_evaluation, (reinit_time, s_enabled),
42           ((v_internal_reinit_time_counter, boolean)),
43  (v_internal_reinit_time_counter := false;
44   (for (i, 0, input_arcs_number−1)
45        (v_internal_reinit_time_counter := v_internal_reinit_time_counter or reinit_time(
                i)));
46   (s_reinit_time_counter ⇐ v_internal_reinit_time_counter)))
47
48  process (time_counter, (clk), ∅,
49  (rst
50   (s_time_counter ⇐ 0)
51   (falling
52    (if ((s_enabled = true) and (transition_type ≠ 0))
53        (if (s_reinit_time_counter = false)
54            (if (s_time_counter < maximal_time_counter)
55                (s_time_counter ⇐ s_time_counter + 1))
56            (s_time_counter ⇐ 1))
57        (s_time_counter ⇐ 0)))))
58
59  process (firing_condition_evaluation,
60           (s_enabled, s_condition_combination, s_reinit_time_counter, s_time_counter),
61           ∅,
62  (s_firing_condition ⇐
63   (s_condition_combination = true)
64   and (s_enabled = true)
65   and ( (transition_type = 0)
66         or ((s_reinit_time_counter = false) and
67             (((transition_type = 1) and (s_time_counter >= (time_A_value1))
68                                 and (s_time_counter < time_B_value))
69              or ((transition_type = 2) and (s_time_counter = (time_A_value1)))
70              or ((transition_type = 3) and (s_time_counter >= (time_A_value1)))))
71         or ((s_reinit_time_counter = true)
72             and (transition_type ≠ 0)
73             and (time_A_value = 1)))))
74
75  process (priority_authorization_evaluation, (priority_authorizations),
```

```
76          ((v_priority_combination, boolean)),
77   (v_priority_combination := true;
78    (for (i, 0, input_arcs_number−1)
79          (v_priority_combination := v_priority_combination and priority_authorizations(i)));
80    s_priority_combination ⇐ v_priority_combination))
81
82   process (firable, (clk), ∅,
83   (rst (s_firable ⇐ false)
84        (falling (s_firable ⇐ s_firing_condition))))
85
86   process (fired_evaluation, (s_firable, s_priority_combination), ∅,
87   (fired ⇐ s_firable and s_priority_combination))
```
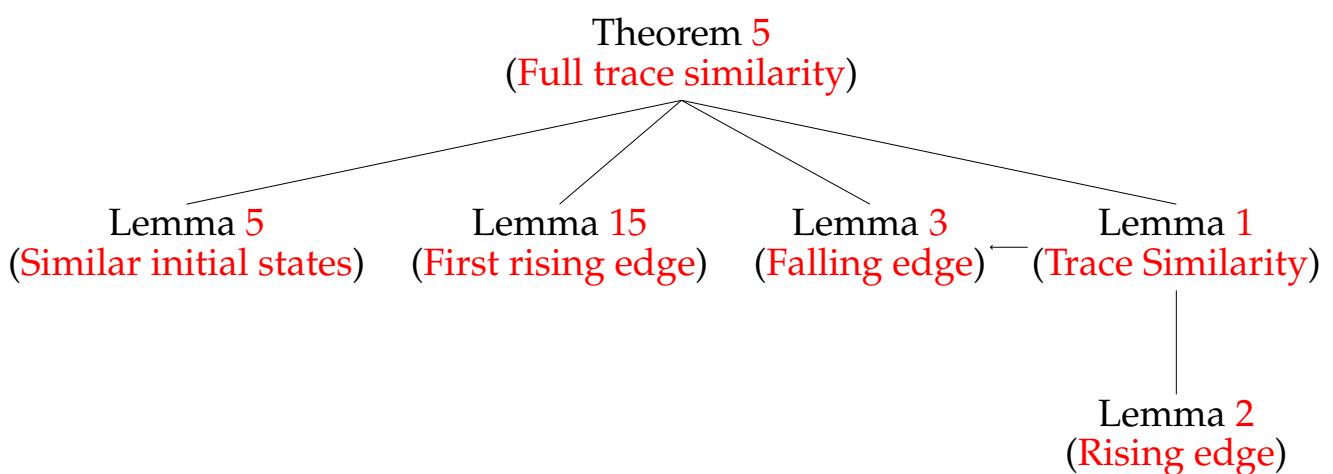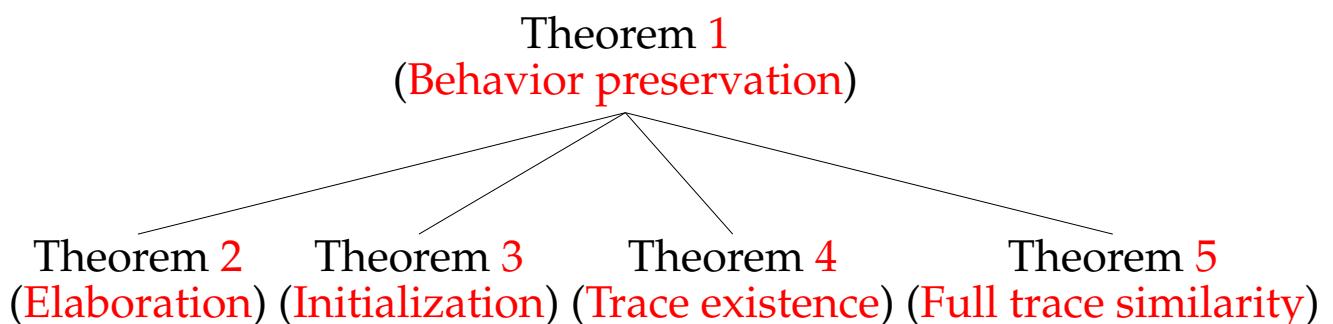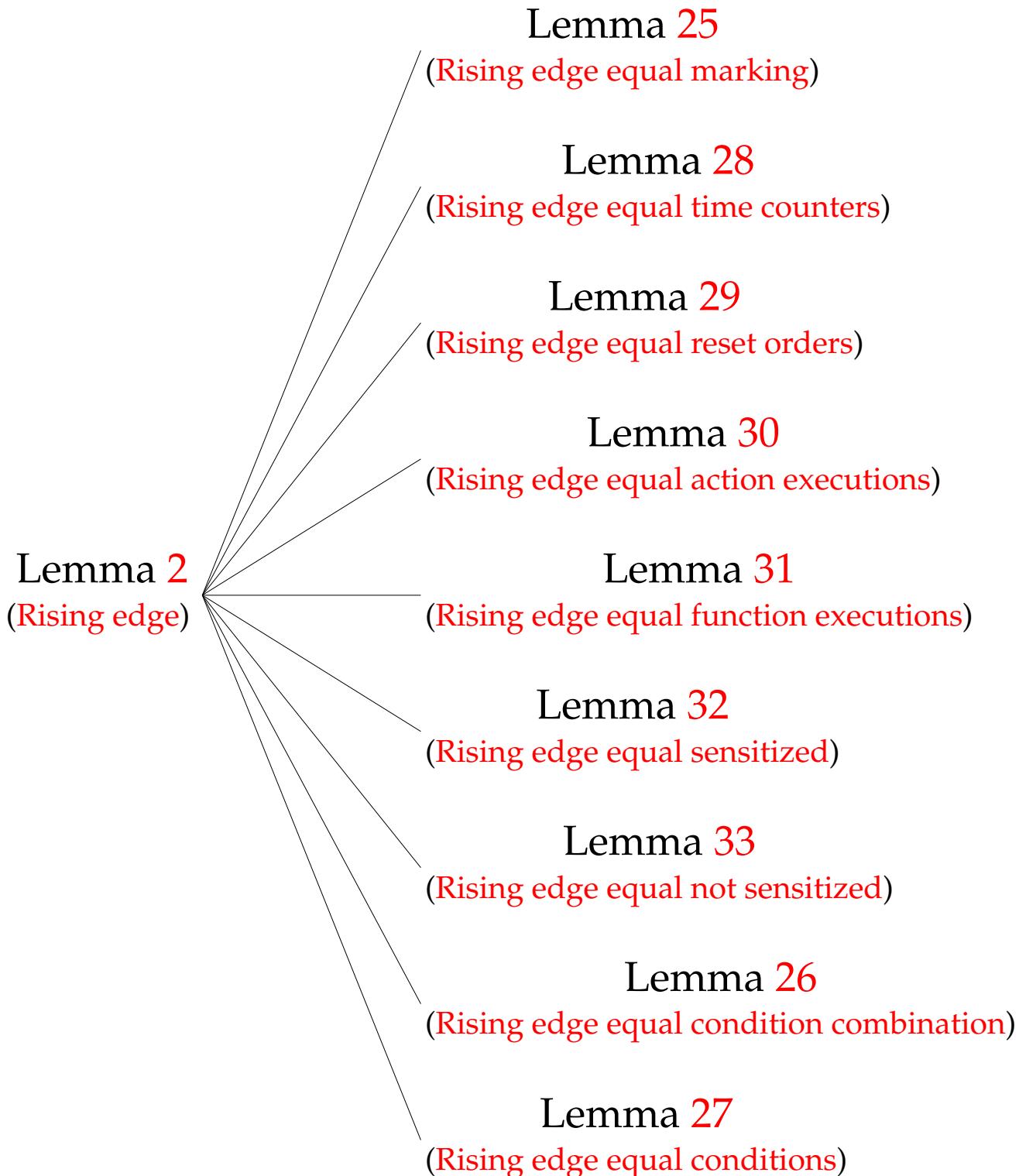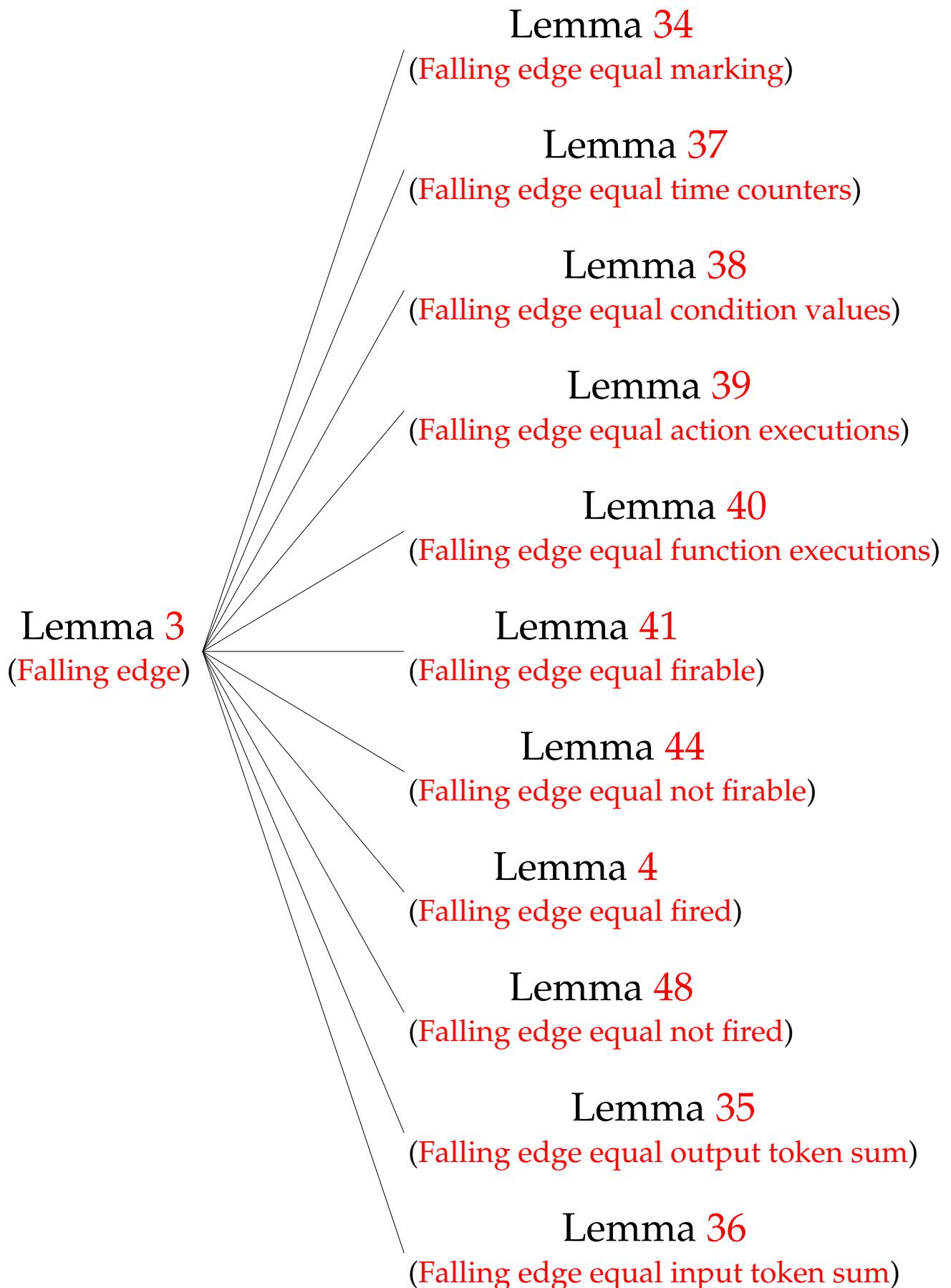
LISTING B.2: The transition design in $\mathcal{H}$-VHDL abstract syntax.

Appendix C

The semantic preservation theorem and its dependencies

Theorem 1
(Behavior preservation)

Theorem 2
(Elaboration)

Theorem 3
(Initialization)

Theorem 4
(Trace existence)

Theorem 5
(Full trace similarity)

Theorem 5
(Full trace similarity)

Lemma 5
(Similar initial states)

Lemma 15
(First rising edge)

Lemma 3
(Falling edge)

Lemma 1
(Trace Similarity)

Lemma 2
(Rising edge)

Lemma 25
(Rising edge equal marking)

Lemma 28
(Rising edge equal time counters)

Lemma 29
(Rising edge equal reset orders)

Lemma 30
(Rising edge equal action executions)

Lemma 2
(Rising edge)

Lemma 31
(Rising edge equal function executions)

Lemma 32
(Rising edge equal sensitized)

Lemma 33
(Rising edge equal not sensitized)

Lemma 26
(Rising edge equal condition combination)

Lemma 27
(Rising edge equal conditions)

Lemma 34
(Falling edge equal marking)

Lemma 37
(Falling edge equal time counters)

Lemma 38
(Falling edge equal condition values)

Lemma 39
(Falling edge equal action executions)

Lemma 40
(Falling edge equal function executions)

Lemma 3
(Falling edge)

Lemma 41
(Falling edge equal firable)

Lemma 44
(Falling edge equal not firable)

Lemma 4
(Falling edge equal fired)

Lemma 48
(Falling edge equal not fired)

Lemma 35
(Falling edge equal output token sum)

Lemma 36
(Falling edge equal input token sum)

Appendix D

Semantic preservation proof

Constants and signals reference			
Full name	*Alias*	*Category*	*Type*
input_arcs_number	ian	generic constant (T)	$\mathbb{N}$
transition_type	tt	generic constant (T)	$\{\texttt{not\_temp}, \texttt{temp\_a\_b},$ $\texttt{temp\_a\_a}, \texttt{temp\_a\_inf}\}$
conditions_number	cn	generic constant (T)	$\mathbb{N}$
maximal_time_counter	mtc	generic constant (T)	$\mathbb{N}$
time_A_value	A	input port (T)	$\mathbb{N}$
time_B_value	B	input port (T)	$\mathbb{N}$
input_conditions	ic	input port (T)	array of $\mathbb{B}$
reinit_time	rt	input port (T)	array of $\mathbb{B}$
input_arcs_valid	iav	input port (T)	array of $\mathbb{B}$
priority_authorizations	pauths	input port (T)	array of $\mathbb{B}$
fired	f	output port (T)	$\mathbb{B}$
s_condition_combination	scc	internal signal (T)	$\mathbb{B}$
s_reinit_time_counter	srtc	internal signal (T)	$\mathbb{B}$
s_priority_combination	spc	internal signal (T)	$\mathbb{B}$
s_firable	sfa	internal signal (T)	$\mathbb{B}$
s_enabled	se	internal signal (T)	$\mathbb{B}$
s_time_counter	stc	internal signal (T)	$\mathbb{N}$
s_firing_condition	sfc	internal signal (T)	$\mathbb{B}$
input_arcs_number	ian	generic constant (P)	$\mathbb{N}$
output_arcs_number	oan	generic constant (P)	$\mathbb{N}$
maximal_marking	mm	generic constant (P)	$\mathbb{N}$
initial_marking	im	input port (P)	$\mathbb{N}$
output_arcs_types	oat	input port (P)	array of $\{\texttt{basic}, \texttt{test}, \texttt{inhib}\}$
output_arcs_weights	oaw	input port (P)	array of $\mathbb{N}$
output_transitions_fired	otf	input port (P)	array of $\mathbb{B}$
input_arcs_weights	iaw	input port (P)	array of $\mathbb{N}$
input_transitions_fired	itf	input port (P)	array of $\mathbb{B}$
output_transitions_fired	otf	output port (P)	array of $\mathbb{B}$

`reinit_transitions_time`	rtt	output port (P)	array of $\mathbb{B}$
`priority_authorizations`	pauths	output port (P)	array of $\mathbb{B}$
`marked`	m	output port (P)	$\mathbb{B}$
`s_marking`	sm	internal signal (P)	$\mathbb{N}$
`s_output_token_sum`	sots	internal signal (P)	$\mathbb{N}$
`s_input_token_sum`	sits	internal signal (P)	$\mathbb{N}$

TABLE D.1: Constants and signals reference for the $\mathcal{H}$-VHDL transition and place designs. In the *Category* column, T (resp. P) indicates a generic constant, input port, output port or internal signal defined in the `transition` (resp. `place`) design.

D.1   Initial States

Definition 49 (Initial state hypotheses). *Given an $sitpn \in SITPN, b \in P \rightarrow \mathbb{N}, d \in design,$ $\gamma \in WM(sitpn, d), \Delta \in ElDesign, \sigma_e, \sigma_0 \in \Sigma,$ assume that:*

– *SITPN sitpn is transformed into the design d and yields the binder γ: $\lfloor sitpn \rfloor_b = (d, \gamma)$*

– *Δ is the elaborated version of d, σ_e is the default state of Δ, i.e. the state of Δ where all signals are initialized to their default value:*

$$\mathcal{D}_{\mathcal{H}}, \varnothing \vdash d \xrightarrow{\;elab\;} (\Delta, \sigma_e)$$

– *σ_0 is the initial state of Δ: $\Delta, \sigma_e \vdash d.cs \xrightarrow{\;init\;} \sigma_0$*

Lemma 5 (Similar initial states). *For all $sitpn \in SITPN, b \in P \rightarrow \mathbb{N}, d \in design, \gamma \in WM(sitpn, d), \Delta \in ElDesign, \sigma_e, \sigma_0 \in \Sigma$ that verify the hypotheses of Definition 49, then $\gamma \vdash s_0 \sim \sigma_0$.*

Proof.

By definition of the General state similarity relation, there are 6 points to prove.

1. $\forall p \in P, id_p \in Comps(\Delta)$ *s.t.* $\gamma(p) = id_p$, $s_0.M(p) = \sigma_0(id_p)(\texttt{s\_marking})$.

2. $\forall t \in T_i, id_t \in Comps(\Delta)$ *s.t.* $\gamma(t) = id_t$,
   $\big(u(I_s(t)) = \infty \wedge s_0.I(t) \leq l(I_s(t)) \Rightarrow s_0.I(t) = \sigma_0(id_t)(\texttt{s\_time\_counter})\big)$
   $\wedge \big(u(I_s(t)) = \infty \wedge s_0.I(t) > l(I_s(t)) \Rightarrow \sigma_0(id_t)(\texttt{s\_time\_counter}) = l(I_s(t))\big)$
   $\wedge \big(u(I_s(t)) \neq \infty \wedge s_0.I(t) > u(I_s(t)) \Rightarrow \sigma_0(id_t)(\texttt{s\_time\_counter}) = u(I_s(t))\big)$
   $\wedge \big(u(I_s(t)) \neq \infty \wedge s_0.I(t) \leq u(I_s(t)) \Rightarrow s_0.I(t) = \sigma_0(id_t)(\texttt{s\_time\_counter})\big)$.

3. $\forall t \in T_i, id_t \in Comps(\Delta)$ *s.t.* $\gamma(t) = id_t$, $s_0.reset_t(t) = \sigma_0(id_t)(\texttt{s\_reinit\_time\_counter})$.

4. $\forall c \in C, id_c \in Ins(\Delta)$ *s.t.* $\gamma(c) = id_c$, $s_0.cond(c) = \sigma_0(id_c)$.

5. $\forall a \in A, id_a \in Outs(\Delta)$ *s.t.* $\gamma(a) = id_a$, $s_0.ex(a) = \sigma_0(id_a)$.

6. $\forall f \in F, id_f \in Outs(\Delta)$ *s.t.* $\gamma(f) = id_f$, $s_0.ex(f) = \sigma_0(id_f)$.

– Apply the Initial states equal marking lemma to solve 1.

– Apply the Initial states equal time counters lemma to solve 2.

– Apply the Initial states equal reset orders lemma to solve 3.

– Apply the Initial states equal condition values lemma to solve 4.

– Apply the Initial states equal action executions lemma to solve 5.

– Apply the Initial states equal function executions lemma to solve 6.

$\square$

D.1.1 Initial states and marking

Lemma 6 (Initial states equal marking). *For all* $sitpn \in SITPN, b \in P \rightarrow \mathbb{N}, d \in design,$ $\gamma \in WM(sitpn, d), \Delta \in ElDesign, \sigma_e, \sigma_0 \in \Sigma$ *that verify the hypotheses of Definition 49, then* $\forall p \in P, id_p \in Comps(\Delta)$ *s.t.* $\gamma(p) = id_p$, $s_0.M(p) = \sigma_0(id_p)(\texttt{s\_marking})$.

Proof.

Given a $p \in P$ and an $id_p \in Comps(\Delta)$ s.t. $\gamma(p) = id_p$, let us show that

$\boxed{s_0.M(p) = \sigma_0(id_p)(\texttt{s\_marking}).}$

By construction and by definition of id_p, there exist g_p, i_p, o_p s.t. $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$.

By property of the $\mathcal{H}$-VHDL initialization relation, $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$, and through the examination of the $\texttt{marking}$ process defined in the place design architecture, we can deduce $\sigma_0(id_p)(\texttt{s\_marking}) = \sigma_0(id_p)(\texttt{initial\_marking})$.

Rewriting $\sigma_0(id_p)(\texttt{sm})$ as $\sigma_0(id_p)(\texttt{initial\_marking})$,

$\boxed{\sigma_0(id_p)(\texttt{initial\_marking}) = s_0.M(p).}$

By construction, $<\texttt{initial\_marking} \Rightarrow M_0(p)> \in i_p$.

By property of the $\mathcal{H}$-VHDL initialization relation, and $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$, then $\sigma_0(id_p)(\texttt{initial\_marking}) = M_0(p)$. Rewriting $\sigma_0(id_p)(\texttt{initial\_marking})$ as $M_0(p)$ in the current goal: $\boxed{M_0(p) = s_0.M(p).}$

By definition of s_0, we can rewrite $s_0.M(p)$ as $M_0(p)$ in the current goal, tautology.

$\square$

Lemma 7 (Null input token sum at initial state). *For all sitpn $\in$ SITPN, $b \in P \to \mathbb{N}$, $d \in design, \gamma \in WM(sitpn, d), \Delta \in ElDesign, \sigma_e, \sigma_0 \in \Sigma$ that verify the hypotheses of Definition 49, then $\forall p \in P, id_p \in Comps(\Delta)$ s.t. $\gamma(p) = id_p, \sigma_0(id_p)(\texttt{s\_input\_token\_sum}) = 0$.*

Proof.

Given a p and an id_p s.t. $\gamma(p) = id_p$, let us show that $\boxed{\sigma_0(id_p)(\texttt{s\_input\_token\_sum}) = 0.}$

By construction and by definition of id_p, there exist g_p, i_p, o_p s.t. $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$.

By property of the initialization relation, $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$, and through the examination of the $\texttt{input\_tokens\_sum}$ process defined in the place design architecture, we can deduce:

$$\sigma_0(id_p)(\texttt{sits}) = \sum_{i=0}^{\Delta(id_p)(\texttt{ian})-1} \begin{cases} \sigma_0(id_p)(\texttt{iaw})[i] \text{ if } \sigma_0(id_p)(\texttt{itf})[i] \\ 0 \text{ } otherwise \end{cases} \tag{D.1}$$

Rewriting the goal with Equation (D.1):

$\boxed{\displaystyle\sum_{i=0}^{\Delta(id_p)(\texttt{ian})-1} \begin{cases} \sigma_0(id_p)(\texttt{iaw})[i] \text{ if } \sigma_0(id_p)(\texttt{itf})[i] \\ 0 \text{ } otherwise \end{cases} = 0.}$

Let us perform case analysis on $input(p)$; there are two cases:

1. $input(p) = \varnothing$:

   By construction, we have $<\texttt{input\_arcs\_number} \Rightarrow 1> \in g_p$,
   $<\texttt{input\_transitions\_fired(0)} \Rightarrow \texttt{true}> \in i_p$,
   and $<\texttt{input\_arcs\_weights(0)} \Rightarrow 0> \in i_p$.

By property of the elaboration relation, $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$, and $<\texttt{input\_arcs\_number} \Rightarrow 1> \in g_p$, we can deduce $\Delta(id_p)(\texttt{ian}) = 1$.

By property of the initialization relation, $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$, $<\texttt{input\_-transitions\_fired(0)} \Rightarrow \texttt{true}> \in i_p$ and $<\texttt{input\_arcs\_weights(0)} \Rightarrow 0> \in i_p$, we can deduce $\sigma_0(id_p)(\texttt{itf})[0] = \texttt{true}$ and $\sigma_0(id_p)(\texttt{iaw})[0] = 0$.

Rewriting the goal with $\Delta(id_p)(\texttt{ian}) = 1$, $\sigma_0(id_p)(\texttt{itf})[0] = \texttt{true}$, $\sigma_0(id_p)(\texttt{iaw})[0] = 0$ and simplifying the goal, tautology.

2. $input(p) \neq \emptyset$:

   By construction, $<\texttt{input\_arcs\_number} \Rightarrow |input(p)|> \in g_p$, and by property of the elaboration relation, and $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$, we can deduce $\Delta(id_p)(\texttt{ian}) = |input(p)|$.

   Let us reason by induction on the sum term of the goal.

   – **BASE CASE**: The sum term equals 0, then tautology.

   – **INDUCTION CASE**:

   $$\sum_{i=1}^{\Delta(id_p)(\texttt{ian})-1} \begin{cases} \sigma_0(id_p)(\texttt{iaw})[i] \text{ if } \sigma_0(id_p)(\texttt{itf})[i] \\ 0 \ otherwise \end{cases} = 0$$

   $$\begin{cases} \sigma_0(id_p)(\texttt{iaw})[0] \text{ if } \sigma_0(id_p)(\texttt{itf})[0] \\ 0 \ otherwise \end{cases}$$
   $$+$$
   $$\sum_{i=1}^{\Delta(id_p)(\texttt{ian})-1} \begin{cases} \sigma_0(id_p)(\texttt{iaw})[i] \text{ if } \sigma_0(id_p)(\texttt{itf})[i] \\ 0 \ otherwise \end{cases} = 0$$

   Using the induction hypothesis to rewrite the goal:

   $$\begin{cases} \sigma_0(id_p)(\texttt{iaw})[0] \text{ if } \sigma_0(id_p)(\texttt{itf})[0] \\ 0 \ otherwise \end{cases} = 0$$

   Since $input(p) \neq \emptyset$, by construction, there exist an $id_t \in Comps(\Delta), g_t, i_t, o_t$ s.t. $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$, $id_{ft} \in Sigs(\Delta)$ s.t. $<\texttt{fired} \Rightarrow id_{ft}> \in o_t$ and $<\texttt{input\_transitions\_fired(0)} \Rightarrow \texttt{id}_{\texttt{ft}}> \in i_p$.

   By property of the initialization relation, $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$, $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$, $<\texttt{fired} \Rightarrow id_{ft}> \in o_t$ and $<\texttt{input\_transitions\_fired(0)} \Rightarrow \texttt{id}_{\texttt{ft}}> \in i_p$, we can deduce $\sigma_0(id_p)(\texttt{itf})[0] = \sigma_0(id_t)(\texttt{fired})$.

Rewriting the goal with $\sigma_0(id_p)(\texttt{itf})[0] = \sigma_0(id_t)(\texttt{fired})$:

$$\begin{cases} \sigma_0(id_p)(\texttt{iaw})[0] \text{ if } \sigma_0(id_t)(\texttt{fired}) \\ 0 \text{ } otherwise \end{cases} = 0$$

Appealing to Lemma 14, we can deduce $\sigma_0(id_t)(\texttt{fired}) = \texttt{false}$.
Rewriting the goal with $\sigma_0(id_t)(\texttt{fired}) = \texttt{false}$, and simplifying the goal, tautology.

□

Lemma 8 (Null output token sum at initial state). *For all sitpn $\in$ SITPN, $b \in P \to \mathbb{N}$, $d \in$ design, $\gamma \in WM(sitpn, d)$, $\Delta \in ElDesign, \sigma_e, \sigma_0 \in \Sigma$ that verify the hypotheses of Definition 49, then $\forall p \in P, id_p \in Comps(\Delta)$ s.t. $\gamma(p) = id_p$, $\sigma_0(id_p)(\texttt{s\_output\_token\_sum}) = 0$.*

Proof.

The proof is similar to the proof of Lemma 7.   □

D.1.2   Initial states and time counters

Lemma 9 (Initial states equal time counters). *For all sitpn $\in$ SITPN, $b \in P \to \mathbb{N}$, $d \in$ design, $\gamma \in WM(sitpn, d)$, $\Delta \in ElDesign, \sigma_e, \sigma_0 \in \Sigma$ that verify the hypotheses of Definition 49, then $\forall t \in T_i, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$,*
$u(I_s(t)) = \infty \land s_0.I(t) \leq l(I_s(t)) \Rightarrow s_0.I(t) = \sigma_0(id_t)(\texttt{s\_time\_counter}) \land$
$u(I_s(t)) = \infty \land s_0.I(t) > l(I_s(t)) \Rightarrow \sigma_0(id_t)(\texttt{s\_time\_counter}) = l(I_s(t)) \land$
$u(I_s(t)) \neq \infty \land s_0.I(t) > u(I_s(t)) \Rightarrow \sigma_0(id_t)(\texttt{s\_time\_counter}) = u(I_s(t)) \land$
$u(I_s(t)) \neq \infty \land s_0.I(t) \leq u(I_s(t)) \Rightarrow s_0.I(t) = \sigma_0(id_t)(\texttt{s\_time\_counter})$.

Proof.

Given a $t \in T_i$ and an $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, let us show that:

1. $\boxed{u(I_s(t)) = \infty \land s_0.I(t) \leq l(I_s(t)) \Rightarrow s_0.I(t) = \sigma_0(id_t)(\texttt{s\_time\_counter})}$

2. $\boxed{u(I_s(t)) = \infty \land s_0.I(t) > l(I_s(t)) \Rightarrow \sigma_0(id_t)(\texttt{s\_time\_counter}) = l(I_s(t))}$

3. $\boxed{u(I_s(t)) \neq \infty \land s_0.I(t) > u(I_s(t)) \Rightarrow \sigma_0(id_t)(\texttt{s\_time\_counter}) = u(I_s(t))}$

4. $\boxed{u(I_s(t)) \neq \infty \land s_0.I(t) \leq u(I_s(t)) \Rightarrow s_0.I(t) = \sigma_0(id_t)(\texttt{s\_time\_counter})}$

By construction and by definition of id_p, there exist g_p, i_p, o_p s.t. $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$.

Then, let us show the 4 previous points.

1. Assuming that $u(I_s(t)) = \infty \wedge s_0.I(t) \leq l(I_s(t))$, then let us show $\boxed{s_0.I(t) = \sigma_0(id_t)(\texttt{s\_time\_counter}).}$

   Rewriting $s_0.I(t)$ as 0, by definition of s_0, $\boxed{\sigma_0(id_t)(\texttt{s\_time\_counter}) = 0.}$

   By property of the $\mathcal{H}$-VHDL initialization relation, $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$, and through the examination of the $\texttt{time\_counter}$ process defined in the transition design architecture, we can deduce $\boxed{\sigma_0(id_t)(\texttt{s\_time\_counter}) = 0.}$

2. Assuming that $u(I_s(t)) = \infty$ and $s_0.I(t) > l(I_s(t))$, let us show $\boxed{\sigma_0(id_t)(\texttt{s\_time\_counter}) = l(I_s(t)).}$

   By definition, $l(I_s(t)) \in \mathbb{N}^*$ and $s_0.I(t) = 0$. Then, $\boxed{l(I_s(t)) < 0 \text{ is a contradiction.}}$

3. Assuming that $u(I_s(t)) \neq \infty$ and $s_0.I(t) > u(I_s(t))$, let us show $\boxed{\sigma_0(id_t)(\texttt{s\_time\_counter}) = u(I_s(t)).}$

   By definition, $u(I_s(t)) \in \mathbb{N}^*$ and $s_0.I(t) = 0$. Then, $\boxed{u(I_s(t)) < 0 \text{ is a contradiction.}}$

4. Assuming that $u(I_s(t)) \neq \infty$ and $s_0.I(t) \leq u(I_s(t))$, let us show $\boxed{s_0.I(t) = \sigma_0(id_t)(\texttt{s\_time\_counter}).}$

   Rewriting $s_0.I(t)$ as 0, by definition of s_0, $\boxed{\sigma_0(id_t)(\texttt{s\_time\_counter}) = 0.}$

   By property of the $\mathcal{H}$-VHDL initialization relation, $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$, and through the examination of the $\texttt{time\_counter}$ process defined in the transition design architecture, we can deduce $\boxed{\sigma_0(id_t)(\texttt{s\_time\_counter}) = 0.}$

   $\square$

D.1.3   Initial states and reset orders

Lemma 10 (Initial states equal reset orders). *For all $sitpn \in SITPN$, $b \in P \rightarrow \mathbb{N}$, $d \in design$, $\gamma \in WM(sitpn, d)$, $\Delta \in ElDesign, \sigma_e, \sigma_0 \in \Sigma$ that verify the hypotheses of Definition 49, then $\forall t \in T_i, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, $s_0.reset_t(t) = \sigma_0(id_t)(\texttt{s\_reinit\_time\_counter}).$*

Proof.

Given a $t \in T_i$ and an $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, let us show that

$\boxed{s_0.reset_t(t) = \sigma_0(id_t)(\texttt{s\_reinit\_time\_counter}).}$

Rewriting $s_0.reset_t(t)$ as $\texttt{false}$, by definition of s_0, $\boxed{\sigma_0(id_t)(\texttt{s\_reinit\_time\_counter}) = \texttt{false}.}$

By construction and by definition of id_t, there exist g_t, i_t, o_t s.t. $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$.

By property of the $\mathcal{H}$-VHDL initialization relation, $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$, and through the examination of the $\texttt{reinit\_time\_counter\_evaluation}$ process defined in the $\texttt{transition}$ design architecture

we can deduce $\sigma_0(id_t)(\texttt{s\_reinit\_time\_counter}) = \prod_{i=0}^{\Delta(id_t)(\texttt{ian})-1} \sigma_0(id_t)(\texttt{rt})[i].$

Rewriting $\sigma_0(id_t)(\texttt{s\_reinit\_time\_counter})$ as $\prod_{i=0}^{\Delta(id_t)(\texttt{ian})-1} \sigma_0(id_t)(\texttt{rt})[i],$

$\boxed{\prod_{i=0}^{\Delta(id_t)(\texttt{ian})-1} \sigma_0(id_t)(\texttt{rt})[i] = \texttt{false}.}$

For all $t \in T$ (resp. $p \in P$), let $input(t)$ (resp. $input(p)$) be the set of input places of t (resp. input transitions of p), and let $output(t)$ (resp. $output(p)$) be the set of output places of t (resp. output transitions of p).

Let us perform case analysis on $input(t)$; there are 2 cases:

– **CASE** $input(t) = \emptyset$.

   By construction, $<\texttt{input\_arcs\_number} \Rightarrow 1> \in g_t$, and by property of the elaboration relation, and $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$, we can deduce $\Delta(id_t)(\texttt{ian}) = 1$.

   By construction, $< \texttt{reinit\_time(0)} \Rightarrow \texttt{false} > \in i_t$, and by property of the initialization relation and $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$, we can deduce $\sigma_0(id_t)(\texttt{rt})[0] = \texttt{false}$.

   Rewriting $\Delta(id_t)(\texttt{ian})$ as 1 and $\sigma_0(id_t)(\texttt{rt})[0]$ as $\texttt{false}$, tautology.

– **CASE** $input(t) \neq \emptyset$.

   To prove the current goal, we can equivalently prove that
   $\boxed{\exists i \in [0, \Delta(id_t)(\texttt{ian}) - 1] \text{ s.t. } \sigma_0(id_t)(\texttt{rt})[i] = \texttt{false}.}$

   Since $input(t) \neq \emptyset$, $\exists p$ s.t. $p \in input(t)$. Let us take such a $p \in input(t)$.

   By construction, for all $p \in P$, there exist id_p s.t. $\gamma(p) = id_p$.

   By construction and by definition of id_p, there exist g_p, i_p, o_p s.t. $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$.

   By construction, there exist $i \in [0, |input(t)| - 1], j \in [0, |output(p)| - 1], id_{ji} \in Sigs(\Delta)$ s.t. $<\texttt{reinit\_transitions\_time(j)} \Rightarrow id_{ji}> \in o_p$ and $<\texttt{reinit\_time(i)} \Rightarrow id_{ji}> \in i_t$. Let us take such a i, j and id_{ji}.

By construction and $input(t) \neq \emptyset$, $<\texttt{input\_arcs\_number} \Rightarrow |input(t)|> \in g_t$.

By property of the $\mathcal{H}$-VHDL elaboration relation and $<\texttt{input\_arcs\_number} \Rightarrow |input(t)|> \in g_t$, we can deduce $\Delta(id_t)(\texttt{ian}) = |input(t)|$.

Since $\Delta(id_t)(\texttt{ian}) = |input(t)|$ and we have an $i \in [0, |input(t)| - 1]$, then, we have an $i \in [0, \Delta(id_t)(\texttt{ian}) - 1]$. Let us take that i to prove the goal.

Then, we must show $\boxed{\sigma_0(id_t)(\texttt{rt})[i] = \texttt{false.}}$

By property of the $\mathcal{H}$-VHDL initialization relation and $<\texttt{reinit\_time(i)} \Rightarrow id_{ji}> \in i_t$, we can deduce $\sigma_0(id_t)(\texttt{rt})[i] = \sigma_0(id_{ji})$.

Rewriting $\sigma_0(id_t)(\texttt{rt})[i]$ as $\sigma_0(id_{ji})$, $\boxed{\sigma_0(id_{ji}) = \texttt{false.}}$

By property of the $\mathcal{H}$-VHDL initialization relation and $<\texttt{reinit\_transitions\_time(j)} \Rightarrow id_{ji}> \in o_p$, we can deduce $\sigma_0(id_{ji}) = \sigma_0(id_p)(\texttt{rtt})[j]$.

Rewriting $\sigma_0(id_{ji})$ as $\sigma_0(id_p)(\texttt{rtt})[j]$, $\boxed{\sigma_0(id_p)(\texttt{rtt})[j] = \texttt{false.}}$

Since $t \in output(p)$, then we know that $output(p) \neq \emptyset$.

Then, by construction, $<\texttt{output\_arcs\_number} \Rightarrow |output(p)|> \in g_p$.

By property of the elaboration relation and $<\texttt{output\_arcs\_number} \Rightarrow |output(p)|> \in g_p$, we can deduce that $\Delta(id_p)(\texttt{oan}) = |output(p)|$.

Since $\Delta(id_p)(\texttt{oan}) = |output(p)|$ and $j \in [0, |output(p)| - 1]$, then $j \in [0, \Delta(id_p)(\texttt{oan}) - 1]$.

By property of the $\mathcal{H}$-VHDL initialization relation, $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$, through the examination of the $\texttt{reinit\_transitions\_time\_evaluation}$ process defined in the $\texttt{place}$ design architecture, and since $j \in [0, \Delta(id_p)(\texttt{oan}) - 1]$, $\boxed{\sigma_0(id_p)(\texttt{rtt})[j] = \texttt{false.}}$

$\square$

D.1.4   Initial states and condition values

Lemma 11 (Initial states equal condition values). *For all* $sitpn \in SITPN$, $b \in P \to \mathbb{N}$, $d \in design$, $\gamma \in WM(sitpn, d)$, $\Delta \in ElDesign$, $\sigma_e, \sigma_0 \in \Sigma$ *that verify the hypotheses of Definition 49, then* $\forall c \in \mathcal{C}, id_c \in Ins(\Delta)$ *s.t.* $\gamma(c) = id_c$, $s_0.cond(c) = \sigma_0(id_c)$.

Proof.

Given a $c \in \mathcal{C}$ and an $id_c \in Ins(\Delta)$ s.t. $\gamma(c) = id_c$, let us show that $\boxed{s_0.cond(c) = \sigma_0(id_c).}$

Rewriting $s_0.cond(c)$ as `false`, by definition of s_0, $\boxed{\sigma_0(id_c) = \texttt{false}.}$
By construction, id_c is an input port identifier of Boolean type in the $\mathcal{H}$-VHDL design d, and thus, by property of the $\mathcal{H}$-VHDL elaboration relation, we can deduce $\sigma_e(id_c) = \texttt{false}$.
By property of the $\mathcal{H}$-VHDL initialization relation and $id_c \in Ins(\Delta)$, we can deduce $\sigma_e(id_c) = \sigma_0(id_c)$.
Rewriting $\sigma_0(id_c)$ as $\sigma_e(id_c)$ and $\sigma_e(id_c)$ as `false`, tautology.

$\square$

D.1.5 Initial states and action executions

Lemma 12 (Initial states equal action executions). *For all* $sitpn \in SITPN$, $b \in P \to \mathbb{N}$, $d \in design$, $\gamma \in WM(sitpn, d)$, $\Delta \in ElDesign$, $\sigma_e, \sigma_0 \in \Sigma$ *that verify the hypotheses of Definition 49, then* $\forall a \in \mathcal{A}, id_a \in Outs(\Delta)$ *s.t.* $\gamma(a) = id_a$, $s_0.ex(a) = \sigma_0(id_a)$.

Proof.

Given a $a \in \mathcal{A}$ and an $id_a \in Outs(\Delta)$ s.t. $\gamma(a) = id_a$, let us show that $\boxed{s_0.ex(a) = \sigma_0(id_a).}$

Rewriting $s_0.ex(a)$ as `false`, by definition of s_0, $\boxed{\sigma_0(id_a) = \texttt{false}.}$
By construction, id_a is an output port identifier of Boolean type in the $\mathcal{H}$-VHDL design d. Moreover, we know that the output port identifier id_a is assigned to `false` in the generated `action` process during the initialization phase (i.e. the assignment is a part of a *reset* block). Thus, we can deduce that $\sigma_0(id_a) = \texttt{false}$.
Rewriting $\sigma_0(id_a)$ as `false`, tautology.

$\square$

D.1.6 Initial states and function executions

Lemma 13 (Initial states equal function executions). *For all* $sitpn \in SITPN$, $b \in P \to \mathbb{N}$, $d \in design$, $\gamma \in WM(sitpn, d)$, $\Delta \in ElDesign$, $\sigma_e, \sigma_0 \in \Sigma$ *that verify the hypotheses of Definition 49, then* $\forall f \in \mathcal{F}, id_f \in Outs(\Delta)$ *s.t.* $\gamma(f) = id_f$, $s_0.ex(f) = \sigma_0(id_f)$.

Proof.

Given a $f \in \mathcal{F}$ and an $id_f \in Outs(\Delta)$ s.t. $\gamma(f) = id_f$, let us show that $\boxed{s_0.ex(f) = \sigma_0(id_f).}$

Rewriting $s_0.ex(f)$ as `false`, by definition of s_0, $\boxed{\sigma_0(id_f) = \text{false}.}$

By construction, id_f is an output port identifier of Boolean type in the $\mathcal{H}$-VHDL design d, and thus, by property of the $\mathcal{H}$-VHDL elaboration relation, we can deduce $\sigma_e(id_f) = $ `false`.

By construction, and by property of the initialization relation, we know that the output port identifier id_f is assigned to `false` in the generated `function` process during the initialization phase (i.e. the assignment is a part of a *reset* block). Thus, we can deduce $\sigma_0(id_f) = \text{false}$.

Rewriting $\sigma_0(id_f)$ as `false`, tautology.

$\square$

D.1.7 Initial states and fired transitions

Lemma 14 (No fired at initial state). $\forall d \in design, \Delta \in ElDesign, \sigma_e, \sigma_0 \in \Sigma, id_t \in Comps(\Delta), g_t, i_t, o_t$ s.t. :

- $\mathcal{D}_{\mathcal{H}}, \varnothing \vdash d.cs \xrightarrow{elab} \sigma_0$

- $\Delta, \sigma_e \vdash d.cs \xrightarrow{init} \sigma_0$

- `comp(`id_t`, transition, `g_t, i_t, o_t`)` $\in d.cs$

then $\sigma_0(id_t)(\texttt{fired}) = \text{false}$.

Proof.

Assuming all the above hypotheses, let us show $\boxed{\sigma_0(id_t)(\texttt{fired}) = \text{false}.}$

By property of the initialization relation, `comp(`id_t`, transition, `g_t, i_t, o_t`)` $\in d.cs$, and through the examination of the `fired_evaluation` process defined in the transition design architecture, we can deduce:

$$\sigma_0(id_t)(\texttt{fired}) = \sigma_0(id_t)(\texttt{s\_firable}) . \sigma_0(id_t)(\texttt{s\_priority\_combination}) \qquad \text{(D.2)}$$

Rewriting the goal with Equation (D.2): $\boxed{\sigma_0(id_t)(\texttt{sfa}) . \sigma_0(id_t)(\texttt{spc}) = \text{false}.}$

By property of the initialization relation, `comp(`id_t`, transition, `g_t, i_t, o_t`)` $\in d.cs$, and through the examination of the `firable` process defined in the `transition` design architecture, we can deduce $\sigma_0(id_t)(\texttt{sfa}) = \text{false}$.

Rewriting the goal with $\sigma_0(id_t)(\mathtt{sfa}) = \mathtt{false}$ and simplifying the goal, tautology. □

D.2 First Rising Edge

Definition 50 (First rising edge hypotheses). *Given a sitpn* $\in$ *SITPN*, $b \in P \to \mathbb{N}$, $d \in$ *design*, $\gamma \in WM(sitpn, d)$, $\Delta \in ElDesign$, $\sigma_e, \sigma_0, \sigma_i, \sigma_\uparrow, \sigma \in \Sigma$, $E_c \in \mathbb{N} \to \mathcal{C} \to \mathbb{B}$, $E_p \in \mathbb{N} \times \{\uparrow, \downarrow\}) \to Ins(\Delta) \to value$, $\tau \in \mathbb{N}$, *assume that:*

– $\lfloor sitpn \rfloor_b = (d, \gamma)$ *and* $\mathcal{D}_\mathcal{H}, \varnothing \vdash d \xrightarrow{\ elab\ } (\Delta, \sigma_e)$ *and* $\gamma \vdash E_p \overset{env}{=\!=} E_c$

– σ_0 *is the initial state of* Δ: $\Delta, \sigma_e \vdash d.cs \xrightarrow{\ init\ } \sigma_0$

– $E_c, \tau \vdash s_0 \xrightarrow{\uparrow_0} s_0$

– $\mathtt{Inject}(\sigma_0, E_p, \tau, \sigma_i)$ *and* $\Delta, \sigma_i \vdash d.cs \xrightarrow{\uparrow} \sigma_\uparrow$ *and* $\Delta, \sigma_\uparrow \vdash d.cs \xrightarrow{\theta} \sigma$

Lemma 15 (First rising edge). *For all* $sitpn, b, d, \gamma, \Delta, \sigma_e, \sigma_0, \sigma_i, \sigma_\uparrow, \sigma, E_c, E_p, \tau$ *that verify the hypotheses of Definition 50, then* $\gamma, E_c, \tau \vdash s_0 \overset{\uparrow}{\approx} \sigma$.

Proof.

By definition of the Full post rising edge state similarity relation, there are 8 points to prove.

1. $\forall p \in P, id_p \in Comps(\Delta)$ s.t. $\gamma(p) = id_p$, $s_0.M(p) = \sigma(id_p)(\texttt{s\_marking})$.

2. $\forall t \in T_i, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$,
   $\big(u(I_s(t)) = \infty \wedge s_0.I(t) \leq l(I_s(t)) \Rightarrow s_0.I(t) = \sigma(id_t)(\texttt{s\_time\_counter})\big)$
   $\wedge\big(u(I_s(t)) = \infty \wedge s_0.I(t) > l(I_s(t)) \Rightarrow \sigma(id_t)(\texttt{s\_time\_counter}) = l(I_s(t))\big)$
   $\wedge\big(u(I_s(t)) \neq \infty \wedge s_0.I(t) > u(I_s(t)) \Rightarrow \sigma(id_t)(\texttt{s\_time\_counter}) = u(I_s(t))\big)$
   $\wedge\big(u(I_s(t)) \neq \infty \wedge s_0.I(t) \leq u(I_s(t)) \Rightarrow s_0.I(t) = \sigma(id_t)(\texttt{s\_time\_counter})\big)$.

3. $\forall t \in T_i, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, $s_0.reset_t(t) = \sigma(id_t)(\texttt{s\_reinit\_time\_counter})$.

4. $\forall a \in \mathcal{A}, id_a \in Outs(\Delta)$ s.t. $\gamma(a) = id_a$, $s_0.ex(a) = \sigma(id_a)$.

5. $\forall f \in \mathcal{F}, id_f \in Outs(\Delta)$ s.t. $\gamma(f) = id_f$, $s_0.ex(f) = \sigma(id_f)$.

6. $\forall t \in T, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, $t \in Sens(s_0.M) \Leftrightarrow \sigma(id_t)(\texttt{s_enabled}) = $ true.

7. $\forall t \in T, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, $t \notin Sens(s_0.M) \Leftrightarrow \sigma(id_t)(\texttt{s_enabled}) = $ false.

8. $\forall t \in T, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$,
   $$\sigma(id_t)(\texttt{s\_condition\_combination}) = \prod_{c \in conds(t)} \begin{cases} E_c(\tau, c) & if\ \mathbb{C}(t,c) = 1 \\ \texttt{not}(E_c(\tau, c)) & if\ \mathbb{C}(t,c) = -1 \end{cases}$$
   where $conds(t) = \{c \in \mathcal{C} \mid \mathbb{C}(t,c) = 1 \vee \mathbb{C}(t,c) = -1\}$.

9. $\forall c \in \mathcal{C}, id_c \in Ins(\Delta)$ s.t. $\gamma(c) = id_c$, $\sigma(id_c) = E_c(\tau, c)$.

– Apply the First rising edge equal marking lemma to solve 1.

– Apply the First rising edge equal time counters lemma to solve 2.

– Apply the First rising edge equal reset orders lemma to solve 3.

– Apply the First rising edge equal action executions lemma to solve 4.

– Apply the First rising edge equal function executions lemma to solve 5.

– Apply the First rising edge equal sensitized lemma to solve 6.

– Apply the First rising edge not equal sensitized lemma to solve 7.

– Apply the First rising edge equal condition combination lemma to solve 8.

– Apply the First rising edge equal conditions lemma to solve 9.

□

D.2.1   First rising edge and marking

Lemma 16 (First rising edge equal marking). *For all sitpn, $b, d, \gamma, \Delta, \sigma_e, \sigma_0, \sigma_i, \sigma_\uparrow, \sigma, E_c, E_p,$* *$\tau$ that verify the hypotheses of Definition 50, then $\forall p \in P, id_p \in Comps(\Delta)$ s.t. $\gamma(p) = id_p,$* *$s_0.M(p) = \sigma(id_p)(\texttt{s\_marking}).$*

Proof.

Given a p and an id_p s.t. $\gamma(p) = id_p$, let us show that $\boxed{s_0.M(p) = \sigma(id_p)(\texttt{s\_marking}).}$
By construction and by definition of id_p, there exist g_p, i_p, o_p s.t. $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs.$
By property of the $\texttt{Inject}$ relation, the $\mathcal{H}$-VHDL rising edge relation, the stabilize relation, $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$, and through the examination of the $\texttt{marking}$ process defined in the place design architecture, we can deduce:

$$\sigma(id_p)(\texttt{sm}) = \sigma_0(id_p)(\texttt{sm}) + \sigma_0(id_p)(\texttt{sits}) - \sigma_0(id_p)(\texttt{sots}) \tag{D.3}$$

Rewriting the goal with Equation (D.3):
$\boxed{s_0.M(p) = \sigma_0(id_p)(\texttt{sm}) + \sigma_0(id_p)(\texttt{sits}) - \sigma_0(id_p)(\texttt{sots}).}$
Appealing to Lemmas 7 and 8, we can deduce $\sigma_0(id_p)(\texttt{sits}) = 0$ and $\sigma_0(id_p)(\texttt{sots}) = 0$. Rewriting the goal with $\sigma_0(id_p)(\texttt{sits}) = 0$ and $\sigma_0(id_p)(\texttt{sots}) = 0$,
$\boxed{s_0.M(p) = \sigma_0(id_p)(\texttt{sm}).}$

Appealing to Lemma 6, $s_0.M(p) = \sigma_0(id_p)(\texttt{sm}).$ $\square$

D.2.2   First rising edge and time counters

Lemma 17 (First rising edge equal time counters). *For all sitpn, $b, d, \gamma, \Delta, \sigma_e, \sigma_0, \sigma_i, \sigma_\uparrow, \sigma,$* *$E_c, E_p, \tau$ that verify the hypotheses of Definition 50, then* *$\forall t \in T_i, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t,$* *$u(I_s(t)) = \infty \land s_0.I(t) \le l(I_s(t)) \Rightarrow s_0.I(t) = \sigma(id_t)(\texttt{s\_time\_counter}) \land$* *$u(I_s(t)) = \infty \land s_0.I(t) > l(I_s(t)) \Rightarrow \sigma(id_t)(\texttt{s\_time\_counter}) = l(I_s(t)) \land$* *$u(I_s(t)) \ne \infty \land s_0.I(t) > u(I_s(t)) \Rightarrow \sigma(id_t)(\texttt{s\_time\_counter}) = u(I_s(t)) \land$* *$u(I_s(t)) \ne \infty \land s_0.I(t) \le u(I_s(t)) \Rightarrow s_0.I(t) = \sigma(id_t)(\texttt{s\_time\_counter}).$*

Proof.

Given a $t \in T_i$ and an $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, let us show that:

1. $\boxed{u(I_s(t)) = \infty \land s_0.I(t) \le l(I_s(t)) \Rightarrow s_0.I(t) = \sigma(id_t)(\texttt{s\_time\_counter})}$

2. $\boxed{u(I_s(t)) = \infty \land s_0.I(t) > l(I_s(t)) \Rightarrow \sigma(id_t)(\texttt{s\_time\_counter}) = l(I_s(t))}$

3. $\boxed{u(I_s(t)) \neq \infty \land s_0.I(t) > u(I_s(t)) \Rightarrow \sigma(id_t)(\texttt{s\_time\_counter}) = u(I_s(t))}$

4. $\boxed{u(I_s(t)) \neq \infty \land s_0.I(t) \leq u(I_s(t)) \Rightarrow s_0.I(t) = \sigma(id_t)(\texttt{s\_time\_counter})}$

By construction and by definition of id_t, there exist g_t, i_t, o_t s.t. $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$.

Then, let us show the 4 previous points:

1. Assuming that $u(I_s(t)) = \infty$ and $s_0.I(t) \leq l(I_s(t))$, let us show
   $\boxed{s_0.I(t) = \sigma(id_t)(\texttt{stc}).}$

   By property of the $\texttt{Inject}$ relation, the $\mathcal{H}$-VHDL rising edge and stabilize relations, and $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$, we can deduce $\sigma(id_t)(\texttt{stc}) = \sigma_0(id_t)(\texttt{stc})$.

   Rewriting $\sigma(id_t)(\texttt{stc})$ as $\sigma_0(id_t)(\texttt{stc})$, $\boxed{s_0.I(t) = \sigma_0(id_t)(\texttt{stc}).}$

   Appealing to Lemma 9, $\boxed{s_0.I(t) = \sigma_0(id_t)(\texttt{stc}).}$

2. Assuming that $u(I_s(t)) = \infty$ and $s_0.I(t) > l(I_s(t))$, let us show
   $\boxed{\sigma(id_t)(\texttt{stc}) = l(I_s(t)).}$

   By definition, $l(I_s(t)) \in \mathbb{N}^*$ and $s_0.I(t) = 0$. Then, $\boxed{l(I_s(t)) < 0 \text{ is a contradiction.}}$

3. Assuming that $u(I_s(t)) \neq \infty$ and $s_0.I(t) > u(I_s(t))$, let us show
   $\boxed{\sigma(id_t)(\texttt{stc}) = u(I_s(t))}.$

   By definition, $u(I_s(t)) \in \mathbb{N}^*$ and $s_0.I(t) = 0$. Then, $\boxed{u(I_s(t)) < 0 \text{ is a contradiction.}}$

4. Assuming that $u(I_s(t)) \neq \infty$ and $s_0.I(t) \leq u(I_s(t))$, let us show
   $\boxed{s_0.I(t) = \sigma(id_t)(\texttt{stc}).}$

   By property of the $\texttt{Inject}$ relation, the $\mathcal{H}$-VHDL rising edge and stabilize relations, and $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$, we can deduce $\sigma(id_t)(\texttt{stc}) = \sigma_0(id_t)(\texttt{stc})$.

   Rewriting $\sigma(id_t)(\texttt{stc})$ as $\sigma_0(id_t)(\texttt{stc})$, $\boxed{s_0.I(t) = \sigma_0(id_t)(\texttt{stc}).}$

   Appealing to Lemma 9, $\boxed{s_0.I(t) = \sigma_0(id_t)(\texttt{stc}).}$

$\square$

D.2.3   First rising edge and reset orders

Lemma 18 (First rising edge equal reset orders). *For all $sitpn, b, d, \gamma, \Delta, \sigma_e, \sigma_0, \sigma_i, \sigma_\uparrow, \sigma, E_c,$*
E_p, τ that verify the hypotheses of Definition 50, then
$\forall t \in T, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, $s_0.reset_t(t) = \sigma(id_t)(\texttt{s\_reinit\_time\_counter})$.

Proof.

Given a $t \in T$ and an $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, let us show that
$\boxed{s_0.reset_t(t) = \sigma(id_t)(\texttt{srtc}).}$
By construction and by definition of id_t, there exist g_t, i_t, o_t s.t. $\texttt{comp}(id_t, \texttt{transition}, g_t,$
$i_t, o_t) \in d.cs$.
By property of the stabilize relation, $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$, and through
the examination of the $\texttt{reinit\_time\_counter\_evaluation}$ process defined in the transi-
tion design architecture, we can deduce:

$$\sigma(id_t)(\texttt{srtc}) = \sum_{i=0}^{\Delta(id_t)(\texttt{input\_arcs\_number})-1} \sigma(id_t)(\texttt{reinit\_time})[i] \qquad \text{(D.4)}$$

Rewriting the goal with Equation (D.4): $\boxed{s_0.reset_t(t) = \sum_{i=0}^{\Delta(id_t)(\texttt{ian})-1} \sigma(id_t)(\texttt{rt})[i].}$

Let us perform case analysis on $input(t)$; there are two cases:

– **CASE** $input(t) = \varnothing$:

  By construction, $<\texttt{input\_arcs\_number} \Rightarrow 1> \in g_t$, and by property of the $\mathcal{H}$-VHDL
  elaboration relation, we can deduce $\Delta(id_t)(\texttt{ian}) = 1$.

  By construction, $< \texttt{reinit\_time(0)} \Rightarrow \texttt{false} >\in i_t$, and by property of the $\mathcal{H}$-VHDL
  stabilize relation, $\sigma(id_t)(\texttt{rt})[0] = \texttt{false}$.

  Rewriting the goal with $\Delta(id_t)(\texttt{ian}) = 1$ and $\sigma(id_t)(\texttt{rt})[0] = \texttt{false}$,
  $\boxed{s_0.reset_t(t) = \texttt{false}.}$

  By definition of s_0, $\boxed{s_0.reset_t(t) = \texttt{false}.}$

– **CASE** $input(t) \neq \varnothing$:

  By construction, $<\texttt{input\_arcs\_number} \Rightarrow |input(t)|> \in g_t$, and by property of the
  $\mathcal{H}$-VHDL elaboration relation, we can deduce $\Delta(id_t)(\texttt{ian}) = |input(t)|$.

  Rewriting $\Delta(id_t)(\texttt{ian})$ as $|input(t)|$, $\boxed{s_0.reset_t(t) = \sum_{i=0}^{|input(t)|-1} \sigma(id_t)(\texttt{rt})[i].}$

By definition of s_0, $s_0.reset_t(t) = \mathtt{false}$. Rewriting $s_0.reset_t(t)$ as $\mathtt{false}$,

$$\boxed{\sum_{i=0}^{|input(t)|-1} \sigma(id_t)(\mathtt{rt})[i] = \mathtt{false}.}$$

Given a $i \in [0, |input(t)| - 1]$, let us show $\boxed{\sigma(id_t)(\mathtt{rt})[i] = \mathtt{false}.}$

By construction, and since $input(t) \neq \varnothing$, there exist a $p \in input(t)$, an $id_p \in Comps(\Delta)$ s.t. $\gamma(p) = id_p$, a g_p, an i_p, an o_p s.t. $\mathtt{comp}(id_p, \mathtt{place}, g_p, i_p, o_p) \in d.cs$, and there exist a $j \in [0, |output(p)| - 1]$ and an $id_{ji} \in Sigs(\Delta)$ s.t. $\mathtt{<reinit\_transition\_time(j)} \Rightarrow \mathtt{id_{ji}>} \in o_p$ and $\mathtt{<reinit\_time(i)} \Rightarrow \mathtt{id_{ji}>} \in i_t$.

By property of the stabilize relation, $\mathtt{<reinit\_transition\_time(j)} \Rightarrow id_{ji}> \in o_p$ and $\mathtt{<reinit\_time(i)} \Rightarrow id_{ji}> \in i_t$, we can deduce $\sigma(id_t)(\mathtt{rt})[i] = \sigma(id_{ji}) = \sigma(id_p)(\mathtt{rtt})[j]$.

Rewriting $\sigma(id_t)(\mathtt{rt})[i]$ as $\sigma(id_{ji})$ and $\sigma(id_{ji})$ as $\sigma(id_p)(\mathtt{rtt})[j]$, $\boxed{\sigma(id_p)(\mathtt{rtt})[j] = \mathtt{false}.}$

By property of the $\mathcal{H}$-VHDL rising edge and stabilize relations, $\mathtt{comp}(id_p, \mathtt{place}, g_p, i_p, o_p) \in d.cs$, and through the examination of the process defined in the place design architecture, we can deduce:

$$
\begin{aligned}
\sigma(id_p)(\mathtt{rtt})[j] =& ((\sigma_0(id_p)(\mathtt{oat})[j] = \mathtt{basic} + \sigma_0(id_p)(\mathtt{oat})[j] = \mathtt{test}) \\
&.(\sigma_0(id_p)(\mathtt{sm}) - \sigma_0(id_p)(\mathtt{sots}) < \sigma_0(id_p)(\mathtt{oaw})[j]) \\
&.(\sigma_0(id_p)(\mathtt{sots}) > 0)) \\
&+ (\sigma_0(id_p)(\mathtt{otf})[j])
\end{aligned}
\tag{D.5}
$$

Rewriting the goal with Equation (D.5),

$$
\boxed{
\begin{aligned}
\mathtt{false} =& ((\sigma_0(id_p)(\mathtt{oat})[j] = \mathtt{basic} + \sigma_0(id_p)(\mathtt{oat})[j] = \mathtt{test}) \\
&.(\sigma_0(id_p)(\mathtt{sm}) - \sigma_0(id_p)(\mathtt{sots}) < \sigma_0(id_p)(\mathtt{oaw})[j]) \\
&.(\sigma_0(id_p)(\mathtt{sots}) > 0)) \\
&+ (\sigma_0(id_p)(\mathtt{otf})[j])
\end{aligned}
}
$$

By construction, there exists an $id_{fj} \in Sigs(\Delta)$ s.t. $\mathtt{<fired} \Rightarrow \mathtt{id_{fj}>} \in o_t$ and $\mathtt{<output\_transitions\_fired(j)} \Rightarrow \mathtt{id_{fj}>} \in i_p$.

By property of the initialization relation, $\mathtt{<fired} \Rightarrow \mathtt{id_{fj}>} \in o_t$ and $\mathtt{<output\_transitions\_fired(j)} \Rightarrow \mathtt{id_{fj}>} \in i_p$, we can deduce $\sigma_0(id_p)(\mathtt{otf})[j] = \sigma_0(id_{fj}) = \sigma_0(id_t)(\mathtt{fired})$.

Appealing to Lemma 14, we can deduce $\sigma_0(id_t)(\mathtt{fired}) = \mathtt{false}$ and consequently $\sigma_0(id_p)(\mathtt{otf})[j] = \mathtt{false}$.

Rewriting $\sigma_0(id_p)(\texttt{otf})[j]$ as false and simplifying the goal,

$$
\begin{aligned}
false =& ((\sigma_0(id_p)(\texttt{oat})[j] = \texttt{BASIC} + \sigma_0(id_p)(\texttt{oat})[j] = \texttt{TEST}) \\
& .(\sigma_0(id_p)(\texttt{sm}) - \sigma_0(id_p)(\texttt{sots}) < \sigma_0(id_p)(\texttt{oaw})[j]) \\
& .(\sigma_0(id_p)(\texttt{sots}) > 0))
\end{aligned}
$$

Appealing to Lemma 8, we can deduce $\sigma_0(id_p)(\texttt{sots}) = 0$.

Rewriting $\sigma_0(id_p)(\texttt{sots})$ as 0 and simplifying the goal, tautology.

$\square$

D.2.4 First rising edge and action executions

Lemma 19 (First rising edge equal action executions). *For all* $sitpn, b, d, \gamma, \Delta,$ $\sigma_e, \sigma_0, \sigma_i, \sigma_\uparrow, \sigma, E_c, E_p, \tau$ *that verify the hypotheses of Definition 50, then* $\forall a \in \mathcal{A}, id_a \in Outs(\Delta)$ *s.t.* $\gamma(a) = id_a$, $s_0.ex(a) = \sigma(id_a)$.

Proof.

Given an $a \in \mathcal{A}$ and an $id_a \in Outs(\Delta)$ s.t. $\gamma(a) = id_a$, let us show that $s_0.ex(a) = \sigma(id_a)$. By construction, id_a is an output port identifier of Boolean type in the $\mathcal{H}$-VHDL design d. The generated action process assigns a value to the output port id_a only during the initialization phase or a falling edge phase.
By property of the Inject, $\mathcal{H}$-VHDL rising edge and stabilize relations, we can deduce $\sigma(id_a) = \sigma_0(id_a)$.
Rewriting $\sigma(id_a)$ as $\sigma_0(id_a)$, $s_0.ex(a) = \sigma_0(id_a)$. Appealing to Lemma 12, $s_0.ex(a) = \sigma_0(id_a)$.

$\square$

D.2.5 First rising edge and function executions

Lemma 20 (First rising edge equal function executions). *For all* $sitpn, b, d, \gamma, \Delta,$ $\sigma_e, \sigma_0, \sigma_i, \sigma_\uparrow, \sigma, E_c, E_p, \tau$ *that verify the hypotheses of Definition 50, then* $\forall f \in \mathcal{F}, id_f \in Outs(\Delta)$ *s.t.* $\gamma(f) = id_f$, $s_0.ex(f) = \sigma(id_f)$.

Proof.

Given an $f \in \mathcal{F}$ and an $id_f \in Outs(\Delta)$ s.t. $\gamma(f) = id_f$, let us show that $\boxed{s_0.ex(f) = \sigma(id_f).}$

Rewriting $s_0.ex(f)$ as $\texttt{false}$, by definition of s_0, $\boxed{\sigma(id_f) = \texttt{false}.}$

By construction, id_f is an output port identifier of Boolean type in the $\mathcal{H}$-VHDL design d. The generated $\texttt{function}$ process assigns a value to the output port id_f only during the initialization phase or during a rising edge phase.

By construction, the $\texttt{function}$ process is defined in the behavior of design d, i.e. $\texttt{ps}(\texttt{function}, \varnothing, sl, ss) \in d.cs$.

Let $trs(f)$ be the set of transitions associated to function f, i.e. $trs(f) = \{t \in T \mid \mathbb{F}(t, f) = true\}$.

Let us perform case analysis on $trs(f)$; there are two cases:

– **CASE** $trs(f) = \varnothing$:

By construction, $\texttt{id}_\texttt{f} \Leftarrow \texttt{false} \in ss_\uparrow$ where $ss_\uparrow$ is the part of the "$\texttt{function}$" process body executed during a rising edge phase (i.e. a rising edge block statement).

By property of the $\mathcal{H}$-VHDL rising edge and the stabilize relation, $\boxed{\sigma(id_f) = \texttt{false}.}$

– **CASE** $trs(f) \neq \varnothing$:

By construction, $\texttt{id}_\texttt{f} \Leftarrow \texttt{id}_{\texttt{ft}_0} + \cdots + \texttt{id}_{\texttt{ft}_\texttt{n}} \in ss_\uparrow$ where $ss_\uparrow$ is the part of the "$\texttt{function}$" process body executed during the rising edge phase, and $n = |trs(f)| - 1$, and for all $i \in [0, n-1]$, id_{ft_i} is an internal signal of design d.

By property of the $\texttt{Inject}$, the $\mathcal{H}$-VHDL rising edge and stabilize relations, we can deduce $\sigma(id_f) = \sigma_0(id_{ft_0}) + \cdots + \sigma_0(id_{ft_n})$.

Rewriting $\sigma(id_f)$ as $\sigma_0(id_{ft_0}) + \cdots + \sigma_0(id_{ft_n})$, $\boxed{\sigma_0(id_{ft_0}) + \cdots + \sigma_0(id_{ft_n}) = \texttt{false}.}$

By construction, for all id_{ft_i}, there exist a $t_i \in trs(f)$ and an id_{t_i} s.t. $\gamma(t_i) = id_{t_i}$.

By construction and by definition of id_{t_i}, there exist g_{t_i}, i_{t_i} and o_{t_i} s.t. $\texttt{comp}(id_{t_i}, \texttt{transition}, g_{t_i}, i_{t_i}, o_{t_i}) \in d.cs$.

By construction, we have $<\texttt{fired} \Rightarrow \texttt{id}_{\texttt{ft}_\texttt{i}}> \in o_{t_i}$, and by property of the initialization relation, we have $\sigma_0(id_{ft_i}) = \sigma_0(id_{t_i})(\texttt{fired})$.

Rewriting $\sigma_0(id_{ft_i})$ as $\sigma_0(id_{t_i})(\texttt{fired})$, $\boxed{\sigma_0(id_{t_0})(\texttt{fired}) + \cdots + \sigma_0(id_{t_n})(\texttt{fired}) = \texttt{false}.}$

Appealing to Lemma 14, we can deduce $\sigma_0(id_{t_i})(\texttt{fired}) = \texttt{false}$.

Rewriting all $\sigma_0(id_{t_i})(\texttt{fired})$ as $\texttt{false}$ and simplifying the goal, tautology.

$\square$

D.2.6   First rising edge and sensitization

Lemma 21 (First rising edge equal sensitized). *For all* $sitpn, b, d, \gamma, \Delta, \sigma_e, \sigma_0, \sigma_i, \sigma_\uparrow, \sigma, E_c,$ E_p, τ *that verify the hypotheses of Definition 50, then*
$\forall t \in T, id_t \in Comps(\Delta)$ *s.t.* $\gamma(t) = id_t, t \in Sens(s_0.M) \Leftrightarrow \sigma(id_t)(\texttt{s\_enabled}) = \texttt{true}.$

Proof.

See the proof of Lemma 32.   □

Lemma 22 (First rising edge not equal sensitized). *For all* $sitpn, b, d, \gamma, \Delta, \sigma_e, \sigma_0, \sigma_i, \sigma_\uparrow, \sigma,$ E_c, E_p, τ *that verify the hypotheses of Definition 50, then*
$\forall t \in T, id_t \in Comps(\Delta)$ *s.t.* $\gamma(t) = id_t, t \notin Sens(s_0.M) \Leftrightarrow \sigma(id_t)(\texttt{s\_enabled}) = \texttt{false}.$

Proof.

See the proof of Lemma 33.   □

D.2.7   First rising edge and conditions

Lemma 23 (First rising edge equal condition combination). *For all* $sitpn, b, d, \gamma, \Delta,$ $\sigma_e, \sigma_0, \sigma_i, \sigma_\uparrow, \sigma, E_c, E_p, \tau$ *that verify the hypotheses of Definition 50, then*
$\forall t \in T, id_t \in Comps(\Delta)$ *s.t.* $\gamma(t) = id_t,$

$$\sigma(id_t)(\texttt{s\_condition\_combination}) = \prod_{c \in conds(t)} \begin{cases} E_c(\tau, c) & \text{if } \mathbb{C}(t, c) = 1 \\ \texttt{not}(E_c(\tau, c)) & \text{if } \mathbb{C}(t, c) = -1 \end{cases}$$

where $conds(t) = \{c \in \mathcal{C} \mid \mathbb{C}(t, c) = 1 \vee \mathbb{C}(t, c) = -1\}.$

Proof.

See the proof of Lemma 26.   □

Lemma 24 (First rising edge equal conditions). *For all* $sitpn, b, d, \gamma, \Delta, \sigma_e, \sigma_0, \sigma_i, \sigma_\uparrow, \sigma, E_c,$ E_p, τ *that verify the hypotheses of Definition 50, then*
$\forall c \in \mathcal{C}, id_c \in Ins(\Delta)$ *s.t.* $\gamma(c) = id_c, \sigma(id_c) = E_c(\tau, c).$

Proof.

See the proof of Lemma 27.   □

D.3 Rising Edge

Definition 51 (Rising edge hypotheses). *Given an sitpn $\in$ SITPN, $b \in P \to \mathbb{N}$, $d \in$ design, $\gamma \in WM(sitpn, d)$, $E_c \in \mathbb{N} \to C \to \mathbb{B}$, $\Delta \in ElDesign$, $E_p \in \mathbb{N} \to Ins(\Delta) \to value$, $\tau \in \mathbb{N}$, $s, s' \in S(sitpn)$, $\sigma_e, \sigma, \sigma_i, \sigma_\uparrow, \sigma' \in \Sigma$, assume that:*

- $\lfloor sitpn \rfloor_b = (d, \gamma)$ *and* $\gamma \vdash E_p \overset{env}{=} E_c$ *and* $\mathcal{D}_\mathcal{H}, \emptyset \vdash \mathrm{d} \xrightarrow{elab} \Delta, \sigma_e$

- $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$

- $E_c, \tau \vdash s \overset{\uparrow}{\longrightarrow} s'$

- $\mathrm{Inject}(\sigma, E_p, \tau, \sigma_i)$ *and* $\mathcal{D}_\mathcal{H}, \Delta, \sigma_i \vdash \mathrm{d.cs} \overset{\uparrow}{\to} \sigma_\uparrow$ *and* $\mathcal{D}_\mathcal{H}, \Delta, \sigma_\uparrow \vdash \mathrm{d.cs} \overset{\rightsquigarrow}{\to} \sigma'$

- *State σ is a stable design state:* $\mathcal{D}_\mathcal{H}, \Delta, \sigma \vdash \mathrm{d.cs} \xrightarrow{comb} \sigma$

D.3.1 Rising edge and Marking

Lemma 25 (Rising edge equal marking). *For all sitpn, b, d, γ, E_c, E_p, τ, Δ, σ_e, s, s', σ, σ_i, $\sigma_\uparrow$, σ' that verify the hypotheses of Definition 51, then $\forall p, id_p$ s.t. $\gamma(p) = id_p$, $s'.M(p) = \sigma'(id_p)(\mathtt{s\_marking})$.*

Proof.

Given a $p \in P$, let us show $\boxed{s'.M(p) = \sigma'(id_p)(\mathtt{s\_marking}).}$

By construction and by definition of id_p, there exist g_p, i_p, o_p s.t. $\mathtt{comp}(id_p, \mathtt{place}, g_p, i_p, o_p) \in d.cs$.

By definition of the SITPN state transition relation on rising edge:

$$s'.M(p) = s.M(p) - \sum_{t \in Fired(s)} pre(p, t) + \sum_{t \in Fired(s)} post(t, p) \tag{D.6}$$

By property of the $\mathtt{Inject}$, the $\mathcal{H}$-VHDL rising edge and the stabilize relations, $\mathtt{comp}(id_p, \mathtt{place}, g_p, i_p, o_p) \in d.cs$, and through the examination of the $\mathtt{marking}$ process defined in the place design architecture, we can deduce:

$$\sigma'(id_p)(\mathtt{sm}) = \sigma(id_p)(\mathtt{sm}) - \sigma(id_p)(\mathtt{s\_output\_token\_sum}) \\ + \sigma(id_p)(\mathtt{s\_input\_token\_sum}) \tag{D.7}$$

Rewriting the goal with D.6 and D.7,

$$s.M(p) - \sum_{t \in Fired(s)} pre(p,t) + \sum_{t \in Fired(s)} post(t,p)$$
$$=$$
$$\sigma(id_p)(\mathtt{sm}) - \sigma(id_p)(\mathtt{sots}) + \sigma(id_p)(\mathtt{sits})$$

By definition of the Full post falling edge state similarity relation, we can deduce $s.M(p) = \sigma(id_p)(\mathtt{sm})$, $\sum_{t \in Fired(s)} pre(p,t) = \sigma(id_p)(\mathtt{sots})$ and $\sum_{t \in Fired(s)} post(t,p) = \sigma(id_p)(\mathtt{sits})$, and thus,

$$s.M(p) - \sum_{t \in Fired(s)} pre(p,t) + \sum_{t \in Fired(s)} post(t,p)$$
$$=$$
$$\sigma(id_p)(\mathtt{sm}) - \sigma(id_p)(\mathtt{sots}) + \sigma(id_p)(\mathtt{sits})$$

$\square$

D.3.2 Rising edge and conditions

Lemma 26 (Rising edge equal condition combination). *For all sitpn, b, d, γ, E_c, E_p, τ, Δ, σ_e, s, s', σ, σ_i, $\sigma_\uparrow$, σ' that verify the hypotheses of Definition 51, then $\forall t \in T, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$,*

$$\sigma'(id_t)(\mathtt{s\_condition\_combination}) = \prod_{c \in conds(t)} \begin{cases} E_c(\tau,c) & if\ \mathbb{C}(t,c) = 1 \\ \mathtt{not}(E_c(\tau,c)) & if\ \mathbb{C}(t,c) = -1 \end{cases}$$

where $conds(t) = \{c \in \mathcal{C} \mid \mathbb{C}(t,c) = 1 \vee \mathbb{C}(t,c) = -1\}$.

Proof.

Given a t and an id_t s.t. $\gamma(t) = id_t$, let us show

$$\sigma'(id_t)(\mathtt{s\_condition\_combination}) = \prod_{c \in conds(t)} \begin{cases} E_c(\tau,c) & if\ \mathbb{C}(t,c) = 1 \\ \mathtt{not}(E_c(\tau,c)) & if\ \mathbb{C}(t,c) = -1 \end{cases}.$$

By construction and by definition of id_t, there exist g_t, i_t, o_t s.t. $\mathtt{comp}(id_t, \mathtt{transition}, g_t, i_t, o_t) \in d.cs$.

By property of the $\mathcal{H}$-VHDL stabilize relation, $\mathtt{comp}(id_t, \mathtt{transition}, g_t, i_t, o_t) \in d.cs$, and through the examination of the $\mathtt{condition\_evaluation}$ process defined in the transition design architecture, we can deduce:

$$\sigma'(id_t)(\mathtt{scc}) = \prod_{i=0}^{\Delta(id_t)(\mathtt{conditions\_number})-1} \sigma'(id_t)(\mathtt{input\_conditions})[i] \qquad (D.8)$$

Rewriting the goal with D.8,

$$\boxed{\prod_{i=0}^{\Delta(id_t)(\mathtt{cn})-1} \sigma'(id_t)(\mathtt{ic})[i] = \prod_{c\in conds(t)} \begin{cases} E_c(\tau,c) & \textit{if } \mathbb{C}(t,c) = 1 \\ \mathtt{not}(E_c(\tau,c)) & \textit{if } \mathbb{C}(t,c) = -1 \end{cases}}.$$

Let us perform case analysis on $conds(t)$; there are two cases:

– **CASE** $conds(t) = \varnothing$: $\boxed{\prod_{i=0}^{\Delta(id_t)(\mathtt{cn})-1} \sigma'(id_t)(\mathtt{ic})[i] = \mathtt{true}.}$

By construction, $<\mathtt{cn} \Rightarrow 1> \in g_t$ and $<\mathtt{ic}(0) \Rightarrow \mathtt{true}> \in i_t$.

By property of the stabilize relation, $<\mathtt{cn} \Rightarrow 1> \in g_t$ and $<\mathtt{ic}(0) \Rightarrow \mathtt{true}> \in i_t$, we can deduce $\Delta(id_t)(\mathtt{cn}) = 1$ and $\sigma'(id_t)(\mathtt{ic})[0] = \mathtt{true}$.

Rewriting the goal with $\Delta(id_t)(\mathtt{cn}) = 1$ and $\sigma'(id_t)(\mathtt{ic})[0] = \mathtt{true}$, tautology.

– **CASE** $conds(t) \neq \varnothing$:
By construction, $<\mathtt{cn} \Rightarrow |conds(t)|> \in g_t$, and by property of the stabilize relation, we can deduce $\Delta(id_t)(\mathtt{cn}) = |conds(t)|$.

Rewriting the goal with $\Delta(id_t)(\mathtt{cn}) = |conds(t)|$:

$$\boxed{\prod_{i=0}^{|conds(t)|-1} \sigma'(id_t)(\mathtt{ic})[i] = \prod_{c\in conds(t)} \begin{cases} E_c(\tau,c) & \textit{if } \mathbb{C}(t,c) = 1 \\ \mathtt{not}(E_c(\tau,c)) & \textit{if } \mathbb{C}(t,c) = -1 \end{cases}}$$

There exists a mapping, given by the transformation function, between the set $conds(t)$ and the indexes of $[0, |conds(t)| - 1]$.

Let $\beta \in conds(t) \to [0, |conds(t)| - 1]$ be this mapping.

To prove the current goal, it suffices to prove that for all condition $c \in conds(t)$, we have

$$\boxed{\left(\begin{cases} E_c(\tau,c) & \textit{if } \mathbb{C}(t,c) = 1 \\ \mathtt{not}(E_c(\tau,c)) & \textit{if } \mathbb{C}(t,c) = -1 \end{cases} \right) = \sigma'(id_t)(\mathtt{ic})[\beta(c)]}$$

Given a $c \in conds(t)$, let us show the above goal.

By construction, for all $c \in conds(t)$, there exists an $id_c \in Ins(\Delta)$ such that

– $\gamma(c) = id_c$
– $\mathbb{C}(t,c) = 1$ implies $<\mathtt{ic}(\beta(c)) \Rightarrow id_c> \in i_t$
– $\mathbb{C}(t,c) = -1$ implies $<\mathtt{ic}(\beta(c)) \Rightarrow \mathtt{not}\ id_c> \in i_t$

Let us take such an id_c with the above properties.

By definition of $c \in conds(t)$, we have $\mathbb{C}(t,c) = 1 \vee \mathbb{C}(t,c) = -1$. Let us perform case analysis on $\mathbb{C}(t,c) = 1 \vee \mathbb{C}(t,c) = -1$:

- **CASE** $\mathbb{C}(t,c) = 1$:

  In that case, we must show: $\boxed{E_c(\tau,c) = \sigma'(id_t)(\mathtt{ic})[\beta(c)]}$

  By assumption, we have $<\mathtt{ic}(\beta(c)) \Rightarrow \mathtt{id_c}> \in i_t$ and by property of the stabilize relation, we can deduce $\sigma(id_t)(\mathtt{ic})[\beta(c)] = \sigma'(id_c)$.

  Rewriting the goal with $\sigma(id_t)(\mathtt{ic})[\beta(c)] = \sigma'(id_c)$:

  $\boxed{E_c(\tau,c) = \sigma'(id_c)}$

  By property of the $\mathtt{Inject}$ relation and $id_c \in Ins(\Delta)$, we can deduce $\sigma'(id_c) = E_p(\tau)(id_c)$.

  By property of $\gamma \vdash E_p \overset{env}{=} E_c$, we can deduce $E_p(\tau)(id_c) = E_c(\tau,c)$.

  Rewriting the goal with $\sigma'(id_c) = E_p(\tau)(id_c)$ and $E_p(\tau)(id_c) = E_c(\tau,c)$:

  $\boxed{E_c(\tau,c) = E_c(\tau,c)}$, then <mark>tautology.</mark>

- **CASE** $\mathbb{C}(t,c) = -1$:

  In that case, we must show: $\boxed{\mathtt{not}\ E_c(\tau,c) = \sigma'(id_t)(\mathtt{ic})[\beta(c)]}$

  By assumption, we have $<\mathtt{ic}(\beta(c)) \Rightarrow \mathtt{not\ id_c}> \in i_t$ and by property of the stabilize relation, we can deduce $\sigma(id_t)(\mathtt{ic})[\beta(c)] = \mathtt{not}\ \sigma'(id_c)$.

  Rewriting the goal with $\sigma(id_t)(\mathtt{ic})[\beta(c)] = \mathtt{not}\ \sigma'(id_c)$:

  $\boxed{\mathtt{not}\ E_c(\tau,c) = \mathtt{not}\ \sigma'(id_c)}$

  By property of the $\mathtt{Inject}$ relation and $id_c \in Ins(\Delta)$, we can deduce $\sigma'(id_c) = E_p(\tau)(id_c)$.

  By property of $\gamma \vdash E_p \overset{env}{=} E_c$, we can deduce $E_p(\tau)(id_c) = E_c(\tau,c)$.

  Rewriting the goal with $\sigma'(id_c) = E_p(\tau)(id_c)$ and $E_p(\tau)(id_c) = E_c(\tau,c)$:

  $\boxed{\mathtt{not}\ E_c(\tau,c) = \mathtt{not}\ E_c(\tau,c)}$, then <mark>tautology.</mark>

$\square$

Lemma 27 (Rising edge equal conditions). *For all $sit\,pn$, b, d, γ, E_c, E_p, τ, Δ, σ_e, s, s', σ, σ_i, $\sigma_\uparrow$, σ' that verify the hypotheses of Definition 51, then*
$\forall c \in \mathcal{C}, id_c \in Ins(\Delta)$ s.t. $\gamma(c) = id_c, \sigma'(id_c) = E_c(\tau,c)$.

Proof.

Given a $c \in \mathcal{C}$ and an $id_c \in Ins(\Delta)$ such that $\gamma(c) = id_c$, let us show

$$\sigma'(id_c) = E_c(\tau,c)$$

By property of the $\mathtt{Inject}$ relation and $id_c \in Ins(\Delta)$, we can deduce $\sigma'(id_c) = E_p(\tau)(id_c)$.
By property of $\gamma \vdash E_p \overset{env}{=} E_c$, we can deduce $E_p(\tau)(id_c) = E_c(\tau,c)$.

Rewriting the goal with $\sigma'(id_c) = E_p(\tau)(id_c)$ and $E_p(\tau)(id_c) = E_c(\tau, c)$, tautology . □

D.3.3 Rising edge and time counters

Lemma 28 (Rising edge equal time counters). *For all $sitpn$, b, d, γ, E_c, E_p, τ, Δ, σ_e, s, s', σ, σ_i, $\sigma_\uparrow$, σ' that verify the hypotheses of Definition 51, then*
$\forall t \in T_i, id_t \in Comps(\Delta) \ s.t. \ \gamma(t) = id_t,$
$\big(u(I_s(t)) = \infty \wedge s'.I(t) \leq l(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(\mathtt{s\_time\_counter})\big)$
$\wedge \big(u(I_s(t)) = \infty \wedge s'.I(t) > l(I_s(t)) \Rightarrow \sigma'(id_t)(\mathtt{s\_time\_counter}) = l(I_s(t))\big)$
$\wedge \big(u(I_s(t)) \neq \infty \wedge s'.I(t) > u(I_s(t)) \Rightarrow \sigma'(id_t)(\mathtt{s\_time\_counter}) = u(I_s(t))\big)$
$\wedge \big(u(I_s(t)) \neq \infty \wedge s'.I(t) \leq u(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(\mathtt{s\_time\_counter})\big).$

Proof.

Given a $t \in T_i$ and an $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, let us show

$\big(u(I_s(t)) = \infty \wedge s'.I(t) \leq l(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(\mathtt{s\_time\_counter})\big)$
$\wedge \big(u(I_s(t)) = \infty \wedge s'.I(t) > l(I_s(t)) \Rightarrow \sigma'(id_t)(\mathtt{s\_time\_counter}) = l(I_s(t))\big)$
$\wedge \big(u(I_s(t)) \neq \infty \wedge s'.I(t) > u(I_s(t)) \Rightarrow \sigma'(id_t)(\mathtt{s\_time\_counter}) = u(I_s(t))\big)$
$\wedge \big(u(I_s(t)) \neq \infty \wedge s'.I(t) \leq u(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(\mathtt{s\_time\_counter})\big)$

By construction and by definition of id_t, there exist g_t, i_t, o_t s.t. $\mathtt{comp}(id_t, \mathtt{transition}, g_t, i_t, o_t) \in d.cs$.

Then, there are 4 points to show:

1. $\boxed{u(I_s(t)) = \infty \wedge s'.I(t) \leq l(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(\mathtt{s\_time\_counter})}$

   Assuming that $u(I_s(t)) = \infty$ and $s'.I(t) \leq l(I_s(t))$, let us show
   $\boxed{s'.I(t) = \sigma'(id_t)(\mathtt{s\_time\_counter}).}$

   By property of the `Inject`, $\mathcal{H}$-VHDL rising edge and stabilize relations, $\mathtt{comp}(id_t, \mathtt{transition}, g_t, i_t, o_t) \in d.cs$, and through the examination of the `time_counter` process defined in the transition design architecture, we can deduce $\sigma'(id_t)(\mathtt{stc}) = \sigma(id_t)(\mathtt{stc})$.

   By property of $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$, we can deduce $s.I(t) = \sigma(id_t)(\mathtt{stc})$.

   Rewriting the goal with $\sigma'(id_t)(\mathtt{stc}) = \sigma(id_t)(\mathtt{stc})$ and $s.I(t) = \sigma(id_t)(\mathtt{stc})$, tautology.

2. $\boxed{u(I_s(t)) = \infty \wedge s'.I(t) > l(I_s(t)) \Rightarrow \sigma'(id_t)(\mathtt{s\_time\_counter}) = l(I_s(t).}$

   Proved in the same fashion as 1.

3. $\boxed{u(I_s(t)) \neq \infty \wedge s'.I(t) > u(I_s(t)) \Rightarrow \sigma'(id_t)(\texttt{s\_time\_counter}) = u(I_s(t)).}$

    Proved in the same fashion as 1.

4. $\boxed{u(I_s(t)) \neq \infty \wedge s'.I(t) \leq u(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(\texttt{s\_time\_counter})}$

    Proved in the same fashion as 1.

$\square$

D.3.4   Rising edge and reset orders

Lemma 29 (Rising edge equal reset orders). *For all sitpn, b, d, γ, E_c, E_p, τ, Δ, σ_e, s, s', σ, σ_i, $\sigma_\uparrow$, σ' that verify the hypotheses of Definition 51, then*
$\forall t \in T_i, id_t \in Comps(\Delta) \text{ s.t. } \gamma(t) = id_t, s'.reset_t(t) = \sigma'(id_t)(\texttt{s\_reinit\_time\_counter})$

Proof.

Given a $t \in T_i$ and an $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, let us show
$\boxed{s'.reset_t(t) = \sigma'(id_t)(\texttt{s\_reinit\_time\_counter}).}$
By construction and by definition of id_t, there exist g_t, i_t, o_t s.t. $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$.
By property of the $\mathcal{H}$-VHDL stabilize relation, $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$, and through the examination of the $\texttt{reinit\_time\_counter\_evaluation}$ process defined in the transition design architecture, we can deduce:

$$\sigma'(id_t)(\texttt{srtc}) = \sum_{i=0}^{\Delta(id_t)(\texttt{input\_arcs\_number})-1} \sigma'(id_t)(\texttt{reinit\_time})[i] \qquad (D.9)$$

Rewriting the goal with (D.9), $\boxed{s'.reset_t(t) = \sum_{i=0}^{\Delta(id_t)(\texttt{ian})-1} \sigma'(id_t)(\texttt{rt})[i].}$

Let us perform case analysis on $input(t)$; there are two cases:

– **CASE** $input(t) = \varnothing$:

  By construction, $<\texttt{input\_arcs\_number} \Rightarrow 1> \in g_t$, and by property of the elaboration relation, we can deduce $\Delta(id_t)(\texttt{ian}) = 1$.

  By construction, there exists an $id_{ft} \in Sigs(\Delta)$ s.t. $<\texttt{reinit\_time(0)} \Rightarrow \texttt{id}_{\texttt{ft}}> \in i_t$ and $<\texttt{fired} \Rightarrow \texttt{id}_{\texttt{ft}}> \in o_t$, and by property of the stabilize relation and $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$, we can deduce $\sigma'(id_t)(\texttt{rt})[0] = \sigma'(id_{ft}) = \sigma'(id_t)(\texttt{fired})$.

Rewriting the goal with $\Delta(id_t)(\text{ian}) = 1$ and $\sigma'(id_t)(\text{rt})[0] = \sigma'(id_{ft}) = \sigma'(id_t)(\text{fired})$: $\boxed{s'.reset_t(t) = \sigma'(id_t)(\text{fired}).}$

By property of the stabilize relation, $\text{comp}(id_t, \text{transition}, g_t, i_t, o_t) \in d.cs$, and through the examination of the fired_evaluation process, we can deduce:

$$\sigma'(id_t)(\text{fired}) = \sigma'(id_t)(\text{s\_firable}) \cdot \sigma'(id_t)(\text{s\_priority\_combination}) \qquad (\text{D.10})$$

Rewriting the goal with (D.10):
$\boxed{s'.reset_t(t) = \sigma'(id_t)(\text{s\_firable}) \cdot \sigma'(id_t)(\text{s\_priority\_combination}).}$

By property of the stabilize relation, $\text{comp}(id_t, \text{transition}, g_t, i_t, o_t) \in d.cs$, and through the examination of the $\text{priority\_authorization\_evaluation}$ process defined in the transition design architecture, we can deduce:

$$\sigma'(id_t)(\text{spc}) = \prod_{i=0}^{\Delta(id_t)(\text{ian})-1} \sigma'(id_t)(\text{priority\_authorizations})[i] \qquad (\text{D.11})$$

As $\Delta(id_t)(\text{ian}) = 1$, we can deduce $\prod_{i=0}^{\Delta(id_t)(\text{ian})-1} \sigma'(id_t)(\text{pauths})[i] = \sigma'(id_t)(\text{pauths})[0]$.

Rewriting the goal with (D.11) and $\prod_{i=0}^{\Delta(id_t)(\text{ian})-1} \sigma'(id_t)(\text{pauths})[i] = \sigma'(id_t)(\text{pauths})[0]$:

$\boxed{s'.reset_t(t) = \sigma'(id_t)(\text{s\_firable}) \cdot \sigma'(id_t)(\text{pauths})[0].}$

By construction, $<\text{priority\_authorizations}(0) \Rightarrow \text{true}> \in i_t$, and by property of the stabilize relation and $\text{comp}(id_t, \text{transition}, g_t, i_t, o_t) \in d.cs$, we can deduce $\sigma'(id_t)(\text{pauths})[0] = \text{true}$.

Rewriting the goal with $\sigma'(id_t)(\text{pauths})[0] = \text{true}$, and simplifying the equation:
$\boxed{s'.reset_t(t) = \sigma'(id_t)(\text{s\_firable}).}$

Let us perform case analysis on $t \in \textit{Fired}(s)$ or $t \notin \textit{Fired}(s)$:

– **CASE** $t \in \textit{Fired}(s)$:

  By property of $E_c, \tau \vdash s \xrightarrow{\uparrow} s'$ (Rule (8)), we can deduce $s'.reset_t(t) = \text{true}$.

  Rewriting the goal with $s'.reset_t(t) = \text{true}$: $\boxed{\sigma'(id_t)(\text{s\_firable}) = \text{true}.}$

  By property of the stabilize, the $\mathcal{H}$-VHDL rising edge and the Inject relations, $\text{comp}(id_t, \text{transition}, g_t, i_t, o_t) \in d.cs$, and through the examination of the firable process defined in the transition design architecture, we can deduce $\sigma(id_t)(\text{s\_firable}) = \sigma'(id_t)(\text{s\_firable})$.

  Rewriting the goal with $\sigma(id_t)(\text{s\_firable}) = \sigma'(id_t)(\text{s\_firable})$, we have
  $\boxed{\sigma(id_t)(\text{s\_firable}) = \text{true}.}$

By property of $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$, we can deduce $t \in \mathit{Firable}(s) \Leftrightarrow \sigma(id_t)(\texttt{sfa}) = \texttt{true}$.

Rewriting the goal with $t \in \mathit{Firable}(s) \Leftrightarrow \sigma(id_t)(\texttt{sfa}) = \texttt{true}$, $\boxed{t \in \mathit{Firable}(s).}$

By property of $t \in \mathit{Fired}(s)$, $t \in \mathit{Firable}(s)$.

- **CASE** $t \notin \mathit{Fired}(s)$:

  By property of $\mathit{input}(t) = \varnothing$, there does not exist any input place connected to t by a `basic` or `test` arc. Thus, by property of $E_c, \tau \vdash s \overset{\uparrow}{\longrightarrow} s'$ (Rule (8)), we can deduce $s'.\mathit{reset}_t(t) = \texttt{false}$.

  Rewriting the goal with $s'.\mathit{reset}_t(t) = \texttt{false}$: $\boxed{\sigma'(id_t)(\texttt{s\_firable}) = \texttt{false}.}$

  By property of the stabilize, the $\mathcal{H}$-VHDL rising edge and the `Inject` relations, $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$, and through the examination of the `firable` process defined in the transition design architecture, we can deduce $\sigma(id_t)(\texttt{sfa}) = \sigma'(id_t)(\texttt{sfa})$.

  Rewriting the goal with $\sigma(id_t)(\texttt{sfa}) = \sigma'(id_t)(\texttt{sfa})$, $\boxed{\sigma(id_t)(\texttt{sfa}) = \texttt{false}.}$

  By property of $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$, we can deduce $t \notin \mathit{Firable}(s) \Leftrightarrow \sigma(id_t)(\texttt{sfa}) = \texttt{false}$.

  By property of $t \notin \mathit{Fired}(s)$ and $\mathit{input}(t) = \varnothing$, $t \notin \mathit{Firable}(s)$.

- **CASE** $\mathit{input}(t) \neq \varnothing$:

  By construction, $<\texttt{input\_arcs\_number} \Rightarrow |\mathit{input}(t)|> \in g_t$, and by property of the elaboration relation, we can deduce $\Delta(id_t)(\texttt{ian}) = |\mathit{input}(t)|$.

  Rewriting the goal with $\Delta(id_t)(\texttt{ian}) = |\mathit{input}(t)|$,

  $$\boxed{s'.\mathit{reset}_t(t) = \sum_{i=0}^{|\mathit{input}(t)|-1} \sigma'(id_t)(\texttt{rt})[i].}$$

  Let us perform case analysis on $t \in \mathit{Fired}(s)$ or $t \notin \mathit{Fired}(s)$:

- **CASE** $t \in \mathit{Fired}(s)$:

  By property of $E_c, \tau \vdash s \overset{\uparrow}{\longrightarrow} s'$ (Rule (8)), we can deduce $s'.\mathit{reset}_t(t) = \texttt{true}$.

  Rewriting the goal with $s'.\mathit{reset}_t(t) = \texttt{true}$, $\boxed{\sum_{i=0}^{|\mathit{input}(t)|-1} \sigma'(id_t)(\texttt{rt})[i] = \texttt{true}.}$

  To prove the goal, let us show $\boxed{\exists i \in [0, |\mathit{input}(t)| - 1] \text{ s.t. } \sigma'(id_t)(\texttt{rt})[i] = \texttt{true}.}$

  By construction, and $\mathit{input}(t) \neq \varnothing$, there exist $p \in \mathit{input}(t)$ and $id_p \in \mathit{Comps}(\Delta)$ s.t. $\gamma(p) = id_p$.

  By construction and by definition of id_p, there exist g_p, i_p, o_p s.t. $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$.

By construction, there exist an $i \in [0, |input(t)| - 1]$, a $j \in [0, |output(p)| - 1]$ and $id_{ji} \in Sigs(\Delta)$ s.t. $<\texttt{reinit\_transition\_time(j)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in o_p$ and $<\texttt{reinit\_time(i)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in i_t$. Let us take such an i, j and id_{ji}, and let us use i to prove the goal: $\boxed{\sigma'(id_t)(\texttt{rt})[i] = \texttt{true.}}$

By property of the stabilize relation, $<\texttt{reinit\_transition\_time(j)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in o_p$ and $<\texttt{reinit\_time(i)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in i_t$, we can deduce $\sigma'(id_t)(\texttt{rt})[i] = \sigma'(id_{ji}) = \sigma'(id_p)(\texttt{rtt})[j]$.

Rewriting the goal with $\sigma'(id_t)(\texttt{rt})[i] = \sigma'(id_{ji}) = \sigma'(id_p)(\texttt{rtt})[j]$, $\boxed{\sigma'(id_p)(\texttt{rtt})[j] = \texttt{true.}}$

By property of the $\texttt{Inject}$, the $\mathcal{H}$-VHDL rising edge and the stabilize relations, $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$, and through the examination of the $\texttt{reinit\_-}$ $\texttt{transitions\_time\_evaluation}$ process defined in the place design architecture, we can deduce:

$$
\begin{aligned}
\sigma'(id_p)(\texttt{rtt})[j] =& ((\sigma(id_p)(\texttt{oat})[j] = \texttt{basic} + \sigma(id_p)(\texttt{oat})[j] = \texttt{test}) \\
& .(\sigma(id_p)(\texttt{sm}) - \sigma(id_p)(\texttt{sots}) < \sigma(id_p)(\texttt{oaw})[j]) \\
& .(\sigma(id_p)(\texttt{sots}) > 0)) \\
& + \sigma(id_p)(\texttt{otf})[j]
\end{aligned}
\tag{D.12}
$$

Rewriting the goal with (D.12),

$$
\begin{aligned}
\boxed{
\begin{aligned}
\texttt{true} =& ((\sigma(id_p)(\texttt{oat})[j] = \texttt{basic} + \sigma(id_p)(\texttt{oat})[j] = \texttt{test}) \\
& .(\sigma(id_p)(\texttt{sm}) - \sigma(id_p)(\texttt{sots}) < \sigma(id_p)(\texttt{oaw})[j]) \\
& .(\sigma(id_p)(\texttt{sots}) > 0)) \\
& + (\sigma(id_p)(\texttt{otf})[j])
\end{aligned}
}
\end{aligned}
$$

By construction, there exists $id_{ft} \in Sigs(\Delta)$ such that $<\texttt{output\_transitions\_fired(j)} \Rightarrow \texttt{id}_{\texttt{ft}}> \in i_p$ and $<\texttt{fired} \Rightarrow \texttt{id}_{\texttt{ft}}> \in o_t$. By property of state σ, which is a stable state, we have $\sigma(id_t)(\texttt{fired}) = \sigma(id_{ft}) = \sigma(id_p)(\texttt{otf})[j]$.

Rewriting the goal with $\sigma(id_t)(\texttt{fired}) = \sigma(id_{ft}) = \sigma(id_p)(\texttt{otf})[j]$,

$$
\begin{aligned}
\boxed{
\begin{aligned}
\texttt{true} =& ((\sigma(id_p)(\texttt{oat})[j] = \texttt{basic} + \sigma(id_p)(\texttt{oat})[j] = \texttt{test}) \\
& .(\sigma(id_p)(\texttt{sm}) - \sigma(id_p)(\texttt{sots}) < \sigma(id_p)(\texttt{oaw})[j]) \\
& .(\sigma(id_p)(\texttt{sots}) > 0)) \\
& + \sigma(id_t)(\texttt{fired})
\end{aligned}
}
\end{aligned}
$$

By property of $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$, we can deduce $t \in Fired(s) \Leftrightarrow \sigma(id_t)(\texttt{fired}) = \texttt{true}$.

Rewriting the goal with $t \in Fired(s) \Leftrightarrow \sigma(id_t)(\texttt{fired}) = \texttt{true}$ and simplify the goal, then tautology .

- **CASE** $t \notin Fired(s)$: Then, there are two cases that will determine the value of $s'.reset_t(t)$. Either there exists a place p with an output token sum greater than zero, that is connected to t by an `basic` or `test` arc, and such that the transient marking of p disables t; or such a place does not exist (the predicate is decidable).

  * **CASE** there exists such a place p as described above:
    Then, let us take such a place p and $\omega \in \mathbb{N}^*$ s.t.:

    1. $\sum\limits_{t_i \in Fired(s)} pre(p, t_i) > 0$

    2. $pre(p, t) = (\omega, \texttt{basic}) \vee pre(p, t) = (\omega, \texttt{test})$

    3. $s.M(p) - \sum\limits_{t_i \in Fired(s)} pre(p, t_i) < \omega$

    We will only consider the case where $pre(p, t) = (\omega, \texttt{basic})$; the proof is the similar when $pre(p, t) = (\omega, \texttt{test})$.

    Assuming that p exists, and by property of $E_c, \tau \vdash s \xrightarrow{\uparrow} s'$ (Rule (8)), we can deduce $s'.reset_t(t) = \texttt{true}$.

    Rewriting the goal with $s'.reset_t(t) = \texttt{true}$, $\boxed{\sum\limits_{i=0}^{|input(t)|-1} \sigma'(id_t)(\texttt{rt})[i] = \texttt{true}.}$

    To prove the goal, let us show $\boxed{\exists i \in [0, |input(t)| - 1] \text{ s.t. } \sigma'(id_t)(\texttt{rt})[i] = \texttt{true}.}$
    By construction, there exists $id_p \in Comps(\Delta)$ s.t. $\gamma(p) = id_p$.
    By construction and by definition of id_p, there exist g_p, i_p, o_p s.t. $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$.
    By construction, there exist an $i \in [0, |input(t)| - 1]$, a $j \in [0, |output(p)| - 1]$ and $id_{ji} \in Sigs(\Delta)$ s.t. $<\texttt{reinit\_transition\_time(j)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in o_p$ and $<\texttt{reinit\_time(i)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in i_t$. Let us take such an i, j and id_{ji}, and let us use i to prove the goal: $\boxed{\sigma'(id_t)(\texttt{rt})[i] = \texttt{true}.}$
    By property of the stabilize relation, $<\texttt{reinit\_transition\_time(j)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in o_p$ and $<\texttt{reinit\_time(i)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in i_t$, we have $\sigma'(id_t)(\texttt{rt})[i] = \sigma'(id_{ji}) = \sigma'(id_p)(\texttt{rtt})[j]$.
    Rewriting the goal with $\sigma'(id_t)(\texttt{rt})[i] = \sigma'(id_{ji}) = \sigma'(id_p)(\texttt{rtt})[j]$, we have
    $\boxed{\sigma'(id_p)(\texttt{rtt})[j] = \texttt{true}.}$
    By property of the `Inject`, the $\mathcal{H}$-VHDL rising edge and the stabilize relation, and through the examination of the `reinit_transitions_time_evaluation` process

defined in the place design architecture, we can deduce:

$$
\begin{aligned}
\sigma'(id_p)(\mathtt{rtt})[j] =& ((\sigma(id_p)(\mathtt{oat})[j] = \mathtt{basic} + \sigma(id_p)(\mathtt{oat})[j] = \mathtt{test}) \\
& .(\sigma(id_p)(\mathtt{sm}) - \sigma(id_p)(\mathtt{sots}) < \sigma(id_p)(\mathtt{oaw})[j]) \\
& .(\sigma(id_p)(\mathtt{sots}) > 0)) \\
& + \sigma(id_p)(\mathtt{otf})[j]
\end{aligned}
\tag{D.13}
$$

Rewriting the goal with (D.13),

$$
\boxed{
\begin{aligned}
\mathtt{true} =& ((\sigma(id_p)(\mathtt{oat})[j] = \mathtt{basic} + \sigma(id_p)(\mathtt{oat})[j] = \mathtt{test}) \\
& .(\sigma(id_p)(\mathtt{sm}) - \sigma(id_p)(\mathtt{sots}) < \sigma(id_p)(\mathtt{oaw})[j]) \\
& .(\sigma(id_p)(\mathtt{sots}) > 0)) \\
& + \sigma(id_p)(\mathtt{otf})[j]
\end{aligned}
}
$$

By construction, $<\mathtt{output\_arcs\_types(j)} \Rightarrow \mathtt{basic}> \in i_p$ and $<\mathtt{output\_arcs\_weights(j)} \Rightarrow \omega> \in i_p$.

By property of the stabilize relation and $\mathtt{comp}(id_p, \mathtt{place}, g_p, i_p, o_p) \in d.cs$, we can deduce $\sigma'(id_p)(\mathtt{oat})[j] = \mathtt{basic}$ and $\sigma'(id_p)(\mathtt{oaw})[j] = \omega$.

By property of $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$, we can deduce $\sigma(id_p)(\mathtt{sm}) = s.M(p)$ and $\sigma(id_p)(\mathtt{sots}) = \sum\limits_{t_i \in Fired(s)} pre(p, t_i)$.

Rewriting the goal with $\sigma'(id_p)(\mathtt{oat})[j] = \mathtt{basic}$, $\sigma'(id_p)(\mathtt{oaw})[j] = \omega$, $\sigma(id_p)(\mathtt{sm}) = s.M(p)$ and $\sigma(id_p)(\mathtt{sots}) = \sum\limits_{t_i \in Fired(s)} pre(p, t_i)$, and simplifying the goal:

$$
\boxed{
\begin{aligned}
((s.M(p) - \sum_{t_i \in Fired(s)} pre(p, t_i) < \omega) . (\sum_{t_i \in Fired(s)} pre(p, t_i) > 0)) + \sigma(id_t)(\mathtt{fired}) \\
= \\
\mathtt{true}
\end{aligned}
}
$$

We assumed that $s.M(p) - \sum\limits_{t_i \in Fired(s)} pre(p, t_i) < \omega$ and $\sum\limits_{t_i \in Fired(s)} pre(p, t_i) > 0$.

Thus, by assumption:

$$
\begin{aligned}
((s.M(p) - \sum_{t_i \in Fired(s)} pre(p, t_i) < \omega) . (\sum_{t_i \in Fired(s)} pre(p, t_i) > 0)) + \sigma(id_t)(\mathtt{fired}) \\
= \\
\mathtt{true}
\end{aligned}
$$

* **CASE** such a place does not exist:
Then, let us assume that, for all place $p \in P$

1. $\displaystyle\sum_{t_i \in Fired(s)} pre(p, t_i) = 0$

2. or $\forall \omega \in \mathbb{N}^*,\ pre(p, t) = (\omega, \texttt{basic}) \vee pre(p, t) = (\omega, \texttt{test}) \Rightarrow$
   $s.M(p) - \displaystyle\sum_{t_i \in Fired(s)} pre(p, t_i) \geq \omega.$

In that case, by property of $E_c, \tau \vdash s \xrightarrow{\uparrow} s'$ (Rule (8)), we can deduce $s'.reset_t(t) = \texttt{false}$.

Rewriting the goal with $s'.reset_t(t) = \texttt{false}$: $\boxed{\displaystyle\sum_{i=0}^{|input(t)|-1} \sigma'(id_t)(\texttt{rt})[i] = \texttt{false}.}$

To prove the goal, let us show $\boxed{\forall i \in [0, |input(t)| - 1],\ \sigma'(id_t)(\texttt{rt})[i] = \texttt{false}.}$

Given an $i \in [0, |input(t)| - 1]$, let us show $\boxed{\sigma'(id_t)(\texttt{rt})[i] = \texttt{false}.}$

By construction, there exist a $p \in input(t)$, an $id_p \in Comps(\Delta)$, g_p, i_p, o_p, a $j \in [0, |output(p)| - 1]$, an $id_{ji} \in Sigs(\Delta)$ s.t. $\gamma(p) = id_p$ and $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$ and $<\texttt{reinit\_transition\_time(j)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in o_p$ and $<\texttt{reinit\_time(i)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in i_t$. Let us take such a $p, id_p, g_p, i_p, o_p, j$ and id_{ji}.

By property of the stabilize relation, $<\texttt{reinit\_transition\_time(j)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in o_p$ and $<\texttt{reinit\_time(i)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in i_t$, we have $\sigma'(id_t)(\texttt{rt})[i] = \sigma'(id_{ji}) = \sigma'(id_p)(\texttt{rtt})[j]$.

Rewriting the goal with $\sigma'(id_t)(\texttt{rt})[i] = \sigma'(id_{ji}) = \sigma'(id_p)(\texttt{rtt})[j]$:

$\boxed{\sigma'(id_p)(\texttt{rtt})[j] = \texttt{false.}}$

By property of the $\texttt{Inject}$, the $\mathcal{H}$-VHDL rising edge and the stabilize relations, $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$, and through the examination of the $\texttt{reinit\_transitions\_time\_evaluation}$ process defined in the place design architecture, we can deduce:

$$
\begin{aligned}
\sigma'(id_p)(\texttt{rtt})[j] = &((\sigma(id_p)(\texttt{oat})[j] = \texttt{basic} + \sigma(id_p)(\texttt{oat})[j] = \texttt{test}) \\
&.(\sigma(id_p)(\texttt{sm}) - \sigma(id_p)(\texttt{sots}) < \sigma(id_p)(\texttt{oaw})[j]) \\
&.(\sigma(id_p)(\texttt{sots}) > 0)) \\
&+ \sigma(id_p)(\texttt{otf})[j]
\end{aligned}
\tag{D.14}
$$

Rewriting the goal with (D.14),

$$
\boxed{
\begin{aligned}
\texttt{false} = &((\sigma(id_p)(\texttt{oat})[j] = \texttt{basic} + \sigma(id_p)(\texttt{oat})[j] = \texttt{test}) \\
&.(\sigma(id_p)(\texttt{sm}) - \sigma(id_p)(\texttt{sots}) < \sigma(id_p)(\texttt{oaw})[j]) \\
&.(\sigma(id_p)(\texttt{sots}) > 0)) \\
&+ \sigma(id_p)(\texttt{otf})[j])
\end{aligned}
}
$$

By construction, there exists $id_{ft} \in Sigs(\Delta)$ such that
$<\texttt{output\_transitions\_fired(j)} \Rightarrow \texttt{id}_{\texttt{ft}}> \in i_p$ and $<\texttt{fired} \Rightarrow \texttt{id}_{\texttt{ft}}> \in o_t$. By
property of state σ as being a stable state, we have $\sigma(id_t)(\texttt{fired}) = \sigma(id_{ft}) = \sigma(id_p)(\texttt{otf})[j]$.
Rewriting the goal with $\sigma(id_t)(\texttt{fired}) = \sigma(id_{ft}) = \sigma(id_p)(\texttt{otf})[j]$:

$$
\begin{aligned}
\texttt{false} = &((\sigma(id_p)(\texttt{oat})[j] = \texttt{basic} + \sigma(id_p)(\texttt{oat})[j] = \texttt{test}) \\
&.(\sigma(id_p)(\texttt{sm}) - \sigma(id_p)(\texttt{sots}) < \sigma(id_p)(\texttt{oaw})[j]) \\
&.(\sigma(id_p)(\texttt{sots}) > 0)) \\
&+ \sigma(id_t)(\texttt{fired})
\end{aligned}
$$

By property of $\gamma \vdash s \stackrel{\downarrow}{\approx} \sigma$, we can deduce $t \notin Fired(s) \Leftrightarrow \sigma(id_t)(\texttt{fired}) = \texttt{false}$
Rewriting the goal with $t \notin Fired(s) \Leftrightarrow \sigma(id_t)(\texttt{fired}) = \texttt{false}$ and simplifying
the goal:

$$
\begin{aligned}
\texttt{false} = &((\sigma(id_p)(\texttt{oat})[j] = \texttt{basic} + \sigma(id_p)(\texttt{oat})[j] = \texttt{test}) \\
&.(\sigma(id_p)(\texttt{sm}) - \sigma(id_p)(\texttt{sots}) < \sigma(id_p)(\texttt{oaw})[j]) \\
&.(\sigma(id_p)(\texttt{sots}) > 0))
\end{aligned}
$$

Then, based on the assumptions made at the beginning of case, there are two cases:

1. **CASE** $\sum\limits_{t_i \in Fired(s)} pre(p, t_i) = 0$:

   By property of $\gamma \vdash s \stackrel{\downarrow}{\approx} \sigma$, we can deduce $\sum\limits_{t_i \in Fired(s)} pre(p, t_i) = \sigma(id_p)(\texttt{sots})$.

   Rewriting the goal with $\sum\limits_{t_i \in Fired(s)} pre(p, t_i) = \sigma(id_p)(\texttt{sots})$ and

   $\sum\limits_{t_i \in Fired(s)} pre(p, t_i) = 0$, and simplifying the goal: tautology.

2. **CASE** $\forall \omega \in \mathbb{N}^*, \, pre(p, t) = (\omega, \texttt{basic}) \vee pre(p, t) = (\omega, \texttt{test}) \Rightarrow$
   $s.M(p) - \sum\limits_{t_i \in Fired(s)} pre(p, t_i) \geq \omega$:

   Let us perform case analysis on $pre(p, t)$; there are two cases:

   (a) **CASE** $pre(p, t) = (\omega, \texttt{basic})$ or $pre(p, t) = (\omega, \texttt{basic})$:
   By construction, $<\texttt{output\_arcs\_weights(j)} \Rightarrow \omega> \in i_p$.
   By property of stable state σ and $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$, we can
   deduce $\sigma(id_p)(\texttt{oaw})[j] = \omega$.

   By property of $\gamma \vdash s \stackrel{\downarrow}{\approx} \sigma$, we can deduce $\sigma(id_p)(\texttt{sm}) = s.M(p)$ and
   $\sigma(id_p)(\texttt{sots}) = \sum\limits_{t_i \in Fired(s)} pre(p, t_i)$.

Rewriting the goal with $\sigma(id_p)(\texttt{oaw})[j] = \omega$, $\sigma(id_p)(\texttt{sm}) = s.M(p)$ and $\sigma(id_p)(\texttt{sots}) = \sum\limits_{t_i \in Fired(s)} pre(p, t_i)$:

$$
\begin{aligned}
\texttt{false} =(&(\sigma(id_p)(\texttt{oat})[j] = \texttt{basic} + \sigma(id_p)(\texttt{oat})[j] = \texttt{test}) \\
&.(s.M(p) - \sum\limits_{t_i \in Fired(s)} pre(p, t_i) < \omega) \\
&.(\sum\limits_{t_i \in Fired(s)} pre(p, t_i) > 0))
\end{aligned}
$$

We assumed that $s.M(p) - \sum\limits_{t_i \in Fired(s)} pre(p, t_i) \geq \omega$, and then we can deduce $s.M(p) - \sum\limits_{t_i \in Fired(s)} pre(p, t_i) < \omega = \texttt{false}$.

Rewriting the goal with $s.M(p) - \sum\limits_{t_i \in Fired(s)} pre(p, t_i) < \omega = \texttt{false}$, and simplifying the goal, tautology.

(b) **CASE** $pre(p, t) = (\omega, \texttt{inhib})$:

By construction, $<\texttt{output\_arcs\_types(j)} \Rightarrow \texttt{inhib}> \in i_p$.

By property of stable state σ and $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$, we can deduce $\sigma(id_p)(\texttt{oat})[j] = \texttt{inhib}$.

Rewriting the goal with $\sigma(id_p)(\texttt{oat})[j] = \texttt{inhib}$, and simplifying the goal, we have a tautology.

$\square$

D.3.5   Rising edge and action executions

Lemma 30 (Rising edge equal action executions). *For all sitpn, b, d, γ, E_c, E_p, τ, Δ, σ_e, s, s′, σ, σ_i, $\sigma_\uparrow$, σ' that verify the hypotheses of Definition 51, then*
$\forall a \in \mathcal{A}, id_a \in Outs(\Delta) \text{ s.t. } \gamma(a) = id_a, \ s'.ex(a) = \sigma'(id_a).$

Proof.

Given an $a \in \mathcal{A}$ and an $id_a \in Outs(\Delta)$ s.t. $\gamma(a) = id_a$, let us show $s'.ex(a) = \sigma'(id_a).$

By property of $E_c, \tau \vdash s \xrightarrow{\uparrow} s'$, we can deduce $s.ex(a) = s'.ex(a)$.

By construction, id_a is an output port identifier of Boolean type in the $\mathcal{H}$-VHDL design d. The generated "$\texttt{action}$" process is responsible for the assignment of the *id a* only during the initialization phase or during a falling edge phase.

By property of the $\mathcal{H}$-VHDL `Inject`, rising edge, stabilize relations, and the ''`action`'' process, we can deduce $\sigma(id_a) = \sigma'(id_a)$.

Rewriting the goal with $s.ex(a) = s'.ex(a)$ and $\sigma(id_a) = \sigma'(id_a)$, $\boxed{s.ex(a) = \sigma(id_a).}$

By property of $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$, $\boxed{s.ex(a) = \sigma(id_a).}$ $\qquad\qquad\qquad\square$

D.3.6 Rising edge and function executions

Lemma 31 (Rising edge equal function executions). *For all sitpn, b, d, γ, E_c, E_p, τ, Δ, σ_e, s, s', σ, σ_i, $\sigma_\uparrow$, σ' that verify the hypotheses of Definition 51, then*
$\forall f \in \mathcal{F}, id_f \in Outs(\Delta) \text{ s.t. } \gamma(f) = id_f, \ s'.ex(f) = \sigma'(id_f).$

Proof.

Given an $f \in \mathcal{F}$ and an $id_f \in Outs(\Delta)$ s.t. $\gamma(f) = id_f$, let us show $\boxed{s'.ex(f) = \sigma'(id_f).}$

By property of $E_c, \tau \vdash s \overset{\uparrow}{\longrightarrow} s'$ (Rule (9)):

$$s'.ex(f) = \sum_{t \in Fired(s)} \mathbb{F}(t, f) \qquad\qquad (D.15)$$

By construction, id_f is an output port identifier of Boolean type in the $\mathcal{H}$-VHDL design d. The generated `function` process assigns a value to the output port id_f only during the initialization phase or during a rising edge phase.

By construction, the `function` process is defined in the behavior of design d, i.e. `ps(function,∅,sl,ss)` $\in d.cs$.

Let $trs(f)$ be the set of transitions associated to function f, i.e. $trs(f) = \{t \in T \mid \mathbb{F}(t, f) = $ `true`$\}$.

Let us perform case analysis on $trs(f)$; there are two cases:

– **CASE** $trs(f) = \emptyset$:

By construction, `id`$_f$ $\Leftarrow$ `false` $\in ss_\uparrow$ where $ss_\uparrow$ is the part of the `function` process body executed during a rising edge phase.

By property of the $\mathcal{H}$-VHDL rising edge, the stabilize relations and `ps(function,∅,sl,ss)` $\in d.cs$, we can deduce $\sigma'(id_f) = $ `false`.

By property of $\sum_{t \in Fired(s)} \mathbb{F}(t, f)$ and $trs(f) = \emptyset$, we can deduce $\sum_{t \in Fired(s)} \mathbb{F}(t, f) = $ `false`.

Rewriting the goal with (D.15), $\sigma'(id_f) = $ `false` and $\sum_{t \in Fired(s)} \mathbb{F}(t, f) = $ `false`:

tautology.

– **CASE** $trs(f) \neq \emptyset$:

By construction, $\mathtt{id_f} \Leftarrow \mathtt{id_{ft_0}} + \cdots + \mathtt{id_{ft_n}} \in ss_\uparrow$, where $id_{ft_i} \in Sigs(\Delta)$, $ss_\uparrow$ is the part of the $\mathtt{function}$ process body executed during a rising edge phase, and $n = |trs(f)| - 1$.

By property of the $\mathtt{Inject}$, the $\mathcal{H}$-VHDL rising edge, the stabilize relations, and $\mathtt{ps(function, \emptyset, sl, ss)} \in d.cs$, we can deduce:

$$\sigma'(id_f) = \sigma(id_{ft_0}) + \cdots + \sigma(id_{ft_n}) \tag{D.16}$$

Rewriting the goal with (D.15) and (D.16), $\boxed{\displaystyle\sum_{t \in Fired(s)} \mathbb{F}(t, f) = \sigma(id_{ft_0}) + \cdots + \sigma(id_{ft_n}).}$

Let us reason on the value of $\sigma(id_{ft_0}) + \cdots + \sigma(id_{ft_n})$; there are two cases:

– **CASE** $\sigma(id_{ft_0}) + \cdots + \sigma(id_{ft_n}) = \mathtt{true}$:

Then, we can rewrite the goal as follows: $\boxed{\displaystyle\sum_{t \in Fired(s)} \mathbb{F}(t, f) = \mathtt{true}.}$

To prove the above goal, let us show $\boxed{\exists t \in Fired(s) \text{ s.t. } \mathbb{F}(t, f) = \mathtt{true}.}$

From $\sigma(id_{ft_0}) + \cdots + \sigma(id_{ft_n}) = \mathtt{true}$, we can deduce $\exists id_{ft_i} \text{ s.t. } \sigma(id_{ft_i}) = \mathtt{true}$. Let us take such an id_{ft_i}.

By construction, there exist a $t \in trs(f)$, an $id_t \in Comps(\Delta)$, g_t, i_t, o_t such that:

* $\gamma(t) = id_t$
* $\mathtt{comp}(id_t, \mathtt{transition}, g_t, i_t, o_t) \in d.cs$
* $<\mathtt{fired} \Rightarrow \mathtt{id_{ft_i}}> \in o_t$

By property of σ as being a stable design state, and $\mathtt{comp}(id_t, \mathtt{transition}, g_t, i_t, o_t) \in d.cs$, we can deduce $\sigma(id_t)(\mathtt{fired}) = \sigma(id_{ft_i})$, and thus that $\sigma(id_t)(\mathtt{fired}) = \mathtt{true}$.

By property of $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$, we can deduce $t \in Fired(s)$.

Let us use t to prove the goal: $\boxed{\mathbb{F}(t, f) = \mathtt{true}.}$

By definition of $t \in trs(f)$, $\boxed{\mathbb{F}(t, f) = \mathtt{true}.}$

– **CASE** $\sigma(id_{ft_0}) + \cdots + \sigma(id_{ft_n}) = \mathtt{false}$:

Then, we can rewrite the goal as follows: $\boxed{\displaystyle\sum_{t \in Fired(s)} \mathbb{F}(t, f) = \mathtt{false}.}$

To prove the above goal, let us show $\boxed{\forall t \in Fired(s) \text{ s.t. } \mathbb{F}(t, f) = \mathtt{false}.}$

Given a $t \in Fired(s)$, let us show $\boxed{\mathbb{F}(t, f) = \mathtt{false}.}$

Let us perform case analysis on $\mathbb{F}(t, f)$; there are 2 cases:

* **CASE** $\boxed{\mathbb{F}(t, f) = \mathtt{false}.}$

* **CASE** $\mathbb{F}(t, f) = $ true:

By construction, there exist an $id_t \in Comps(\Delta)$, g_t, i_t, o_t and $id_{ft_i} \in Sigs(\Delta)$ such that:

· $\gamma(t) = id_t$

· comp(id_t, transition, g_t, i_t, o_t) $\in d.cs$

· <fired $\Rightarrow$ id$_{\text{ft}_i}$> $\in o_t$

By property of stable design state σ and comp(id_t, transition, g_t, i_t, o_t) $\in d.cs$, we can deduce $\sigma(id_t)(\text{fired}) = \sigma(id_{ft_i})$.

By property of $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$, we can deduce $t \in Fired(s) \Leftrightarrow \sigma(id_t)(\text{fired}) = $ true. Since $t \in Fired(s)$, we can deduce $\sigma(id_t)(\text{fired}) = $ true, and from $\sigma(id_t)(\text{fired}) = \sigma(id_{ft_i})$, we can deduce $\sigma(id_{ft_i}) = $ true.

Then, $\sigma(id_{ft_i}) = $ true contradicts $\sigma(id_{ft_0}) + \cdots + \sigma(id_{ft_n}) = $ false.

$\square$

D.3.7 Rising edge and sensitization

Lemma 32 (Rising edge equal sensitized). *For all sitpn, b, d, γ, E_c, E_p, τ, Δ, σ_e, s, s', σ, σ_i, $\sigma_\uparrow$, σ' that verify the hypotheses of Definition 51, then*
$\forall t \in T, id_t \in Comps(\Delta)$ *s.t.* $\gamma(t) = id_t, t \in Sens(s'.M) \Leftrightarrow \sigma'(id_t)(\text{s_enabled}) = $ true.

Proof.

Given a $t \in T$ and an $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, let us show
$\boxed{t \in Sens(s'.M) \Leftrightarrow \sigma'(id_t)(\text{s\_enabled}) = \text{true.}}$

By construction and by definition of id_t, there exist g_t, i_t, o_t s.t. comp(id_t, transition, g_t, i_t, o_t) $\in d.cs$.. Then, the proof is in two parts:

1. Assuming that $t \in Sens(s'.M)$, let us show $\boxed{\sigma'(id_t)(\text{s\_enabled}) = \text{true.}}$

   By property of the stabilize relation, comp(id_t, transition, g_t, i_t, o_t) $\in d.cs$, and through the examination of the enable_evaluation process defined in the transition design architecture:

   $$\sigma'(id_t)(\text{se}) = \prod_{i=0}^{\Delta(id_t)(\text{ian})-1} \sigma'(id_t)(\text{input\_arcs\_valid})[i] \qquad (\text{D.17})$$

Rewriting the goal with (D.17), $\boxed{\prod_{i=0}^{\Delta(id_t)(\mathtt{ian})-1} \sigma'(id_t)(\mathtt{iav})[i] = \mathtt{true}.}$

To prove the goal, let us show that $\boxed{\forall i \in [0, \Delta(id_t)(\mathtt{ian}) - 1],\ \sigma'(id_t)(\mathtt{iav})[i] = \mathtt{true}.}$

Given an $i \in [0, \Delta(id_t)(\mathtt{ian}) - 1]$, let us show $\boxed{\sigma'(id_t)(\mathtt{iav})[i] = \mathtt{true}.}$

Let us perform case analysis on $input(t)$.

- **CASE** $input(t) = \varnothing$:

  By construction, $<\mathtt{input\_arcs\_number} \Rightarrow 1> \in g_t$ and
  $<\mathtt{input\_arcs\_valid(0)} \Rightarrow \mathtt{true}> \in i_t$.
  By property of the elaboration and stabilize relations and $\mathtt{comp}(id_t, \mathtt{transition}, g_t, i_t, o_t) \in d.cs$, we can deduce $\Delta(id_t)(\mathtt{ian}) = 1$ and $\sigma'(id_t)(\mathtt{iav})[0] = \mathtt{true}$.
  Thanks to $\Delta(id_t)(\mathtt{ian}) = 1$, we can deduce that $i = 0$.
  Rewriting the goal with $\sigma'(id_t)(\mathtt{iav})[0] = \mathtt{true}$, tautology.

- **CASE** $input(t) \neq \varnothing$:

  By construction, $<\mathtt{input\_arcs\_number} \Rightarrow |input(t)|> \in g_t$.
  By property of the elaboration relation and $\mathtt{comp}(id_t, \mathtt{transition}, g_t, i_t, o_t) \in d.cs$, we can deduce $\Delta(id_t)(\mathtt{ian}) = |input(t)|$.
  Thanks to $\Delta(id_t)(\mathtt{ian}) = |input(t)|$, we know that $i \in [0, |input(t)| - 1]$.
  By construction, there exist a $p \in input(t)$, $id_p \in Comps(\Delta)$, g_p, i_p, o_p, $j \in [0, |output(p)| - 1]$ and $id_{ji} \in Sigs(\Delta)$ s.t. $\gamma(p) = id_p$ and
  $\mathtt{comp}(id_p, \mathtt{place}, g_p, i_p, o_p) \in d.cs$ and $<\mathtt{output\_arcs\_valid(j)} \Rightarrow id_{ji}> \in o_p$
  and $<\mathtt{input\_arcs\_valid(i)} \Rightarrow \mathtt{id_{ji}}> \in i_t$.
  By property of the stabilize relation, $\mathtt{comp}(id_t, \mathtt{transition}, g_t, i_t, o_t) \in d.cs$ and
  $\mathtt{comp}(id_p, \mathtt{place}, g_p, i_p, o_p) \in d.cs$, we can deduce $\sigma'(id_t)(\mathtt{iav})[i] = \sigma'(id_{ji}) = \sigma'(id_p)(\mathtt{oav})[j]$.
  Rewriting the goal with $\sigma'(id_t)(\mathtt{iav})[i] = \sigma'(id_{ji}) = \sigma'(id_p)(\mathtt{oav})[j]$:
  $\boxed{\sigma'(id_p)(\mathtt{oav})[j] = \mathtt{true}.}$

  By property of the stabilize relation, $\mathtt{comp}(id_p, \mathtt{place}, g_p, i_p, o_p) \in d.cs$, and through the examination of the $\mathtt{marking\_validation\_evaluation}$ process defined in the place design architecture, we can deduce:

$$
\begin{aligned}
\sigma'(id_p)(\mathtt{oav})[j] = &\big((\sigma'(id_p)(\mathtt{oat})[j] = \mathtt{basic} + \sigma'(id_p)(\mathtt{oat})[j] = \mathtt{test}) \\
&\cdot \sigma'(id_p)(\mathtt{sm}) \geq \sigma'(id_p)(\mathtt{oaw})[j]\big) \\
&+ \big(\sigma'(id_p)(\mathtt{oat})[j] = \mathtt{inhib} \cdot \sigma'(id_p)(\mathtt{sm}) < \sigma'(id_p)(\mathtt{oaw})[j]\big)
\end{aligned}
$$
$$\tag{D.18}$$

Rewriting the goal with (D.18),

$$\texttt{true} = \big((\sigma'(id_p)(\texttt{oat})[j] = \texttt{basic} + \sigma'(id_p)(\texttt{oat})[j] = \texttt{test})$$
$$. \; \sigma'(id_p)(\texttt{sm}) \geq \sigma'(id_p)(\texttt{oaw})[j] \big)$$
$$+ \; \big(\sigma'(id_p)(\texttt{oat})[j] = \texttt{inhib} . \; \sigma'(id_p)(\texttt{sm}) < \sigma'(id_p)(\texttt{oaw})[j] \big)$$

Let us perform case analysis on $pre(p, t)$; there are 3 cases:

– **CASE** $pre(p, t) = (\omega, \texttt{basic})$:

By construction, $<\texttt{output\_arcs\_types(j)} \Rightarrow \texttt{basic}> \in i_p$ and
$<\texttt{output\_arcs\_weights(j)} \Rightarrow \omega> \in i_p$.
By property of the stabilize relation and $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$,
we can deduce $\sigma'(id_p)(\texttt{oat})[j] = \texttt{basic}$ and $\sigma'(id_p)(\texttt{oaw})[j] = \omega$.
Rewriting the goal with $\sigma'(id_p)(\texttt{oat})[j] = \texttt{basic}$ and $\sigma'(id_p)(\texttt{oaw})[j] = \omega$, and
simplifying the goal:
$\boxed{\sigma'(id_p)(\texttt{sm}) \geq \omega = \texttt{true}.}$
Appealing to Lemma 25, we can deduce $s'.M(p) = \sigma'(id_p)(\texttt{sm})$.
Rewriting the goal with $s'.M(p) = \sigma'(id_p)(\texttt{sm})$: $\boxed{s'.M(p) \geq \omega = \texttt{true}.}$
By definition of $t \in Sens(s'.M)$, $\boxed{s'.M(p) \geq \omega = \texttt{true}.}$

– **CASE** $pre(p, t) = (\omega, \texttt{test})$: same as above.
– **CASE** $pre(p, t) = (\omega, \texttt{inhib})$:
By construction, $<\texttt{output\_arcs\_types(j)} \Rightarrow \texttt{inhib}> \in i_p$ and
$<\texttt{output\_arcs\_weights(j)} \Rightarrow \omega> \in i_p$.
By property of the stabilize relation and $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$, we
can deduce $\sigma'(id_p)(\texttt{oat})[j] = \texttt{inhib}$ and $\sigma'(id_p)(\texttt{oaw})[j] = \omega$.
Rewriting the goal with $\sigma'(id_p)(\texttt{oat})[j] = \texttt{inhib}$ and $\sigma'(id_p)(\texttt{oaw})[j] = \omega$, and
simplifying the goal: $\boxed{\sigma'(id_p)(\texttt{sm}) < \omega = \texttt{true}.}$
Appealing to Lemma 25, we can deduce $s'.M(p) = \sigma'(id_p)(\texttt{sm})$.
Rewriting the goal with $s'.M(p) = \sigma'(id_p)(\texttt{sm})$: $\boxed{s'.M(p) < \omega = \texttt{true}.}$
By definition of $t \in Sens(s'.M)$, $\boxed{s'.M(p) < \omega = \texttt{true}.}$

2. Assuming that $\sigma'(id_t)(\texttt{s\_enabled}) = \texttt{true}$, let us show $\boxed{t \in Sens(s'.M).}$

By definition of $t \in Sens(s'.M)$, let us show

$$\forall p \in P, \omega \in \mathbb{N}^*, \; \big(pre(p, t) = (\omega, \texttt{basic}) \vee pre(p, t) = (\omega, \texttt{test}) \Rightarrow s'.M(p) \geq \omega \big) \wedge \big(pre(p, t) = (\omega, \texttt{inhib}) \Rightarrow s'.M(p) < \omega \big)$$

Given a $p \in P$ and an $\omega \in \mathbb{N}^*$, let us show

$\boxed{pre(p,t) = (\omega, \texttt{basic}) \vee pre(p,t) = (\omega, \texttt{test}) \Rightarrow s'.M(p) \geq \omega}$ and

$\boxed{pre(p,t) = (\omega, \texttt{inhib}) \Rightarrow s'.M(p) < \omega.}$

(a) Assuming $pre(p,t) = (\omega, \texttt{basic}) \vee pre(p,t) = (\omega, \texttt{test})$, let us show $\boxed{s'.M(p) \geq \omega.}$

The proceeding is the same for $pre(p,t) = (\omega, \texttt{basic})$ and $pre(p,t) = (\omega, \texttt{test})$. Therefore, we will only cover the case where $pre(p,t) = (\omega, \texttt{basic})$.

By property of the stabilize relation and $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$, equation (D.17) holds.

Rewriting $\sigma'(id_t)(\texttt{se}) = \texttt{true}$ with (D.17), we can deduce:

$$\prod_{i=0}^{\Delta(id_t)(\texttt{ian})-1} \sigma'(id_t)(\texttt{iav})[i] = \texttt{true}.$$

Then, we can deduce that $\forall i \in [0, \Delta(id_t)(\texttt{ian}) - 1]$, $\sigma'(id_t)(\texttt{iav})[i] = \texttt{true}$.

By construction, there exist an $id_p \in Comps(\Delta)$, $g_p, i_p, o_p, i \in [0, |input(t)| - 1]$, $j \in [0, |output(p)| - 1]$ and $id_{ji} \in Sigs(\Delta)$ s.t. $\gamma(p) = id_p$ and $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$ and $<\texttt{output\_arcs\_valid(j)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in o_p$ and $<\texttt{input\_arcs\_valid(i)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in i_t$. Let us take such an $id_p \in Comps(\Delta)$, $g_p, i_p, o_p, i \in [0, |input(t)| - 1]$, $j \in [0, |output(p)| - 1]$ and $id_{ji} \in Sigs(\Delta)$.

By construction, $<\texttt{input\_arcs\_number} \Rightarrow |input(t)|> \in g_t$.

By property of the elaboration relation and $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$, we can deduce $\Delta(id_t)(\texttt{ian}) = |input(t)|$.

Thanks to $\Delta(id_t)(\texttt{ian}) = |input(t)|$, we can deduce that $\forall i \in [0, |input(t)| - 1]$, $\sigma'(id_t)(\texttt{iav})[i] = \texttt{true}$.

Having such an $i \in [0, |input(t)| - 1]$, we can deduce that $\sigma'(id_t)(\texttt{iav})[i] = \texttt{true}$.

By property of the stabilize relation, $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$ and $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$, we can deduce $\sigma'(id_t)(\texttt{iav})[i] = \sigma'(id_{ji}) = \sigma'(id_p)(\texttt{oav})[j]$.

Thanks to $\sigma'(id_t)(\texttt{iav})[i] = \sigma'(id_{ji}) = \sigma'(id_p)(\texttt{oav})[j]$, we have $\sigma'(id_p)(\texttt{oav})[j] = \texttt{true}$.

By property of the stabilize relation and $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$, equation (D.18) holds. Thanks to (D.18), we can deduce that:

$$\begin{aligned}
\texttt{true} = &\big((\sigma'(id_p)(\texttt{oat})[j] = \texttt{basic} + \sigma'(id_p)(\texttt{oat})[j] = \texttt{test}) \\
&\, . \, \sigma'(id_p)(\texttt{sm}) \geq \sigma'(id_p)(\texttt{oaw})[j]\big) \\
&+ \big(\sigma'(id_p)(\texttt{oat})[j] = \texttt{inhib} \, . \, \sigma'(id_p)(\texttt{sm}) < \sigma'(id_p)(\texttt{oaw})[j]\big)
\end{aligned} \qquad \text{(D.19)}$$

By construction, $<\texttt{output\_arcs\_types(j)} \Rightarrow \texttt{basic}> \in i_p$ and $<\texttt{output\_arcs\_weights(j)} \Rightarrow \omega> \in i_p$.

By property of the stabilize relation and $\text{comp}(id_p, \text{place}, g_p, i_p, o_p) \in d.cs$, we can deduce $\sigma'(id_p)(\text{oat})[j] = \text{basic}$ and $\sigma'(id_p)(\text{oaw})[j] = \omega$.

Thanks to $\sigma'(id_p)(\text{oat})[j] = \text{basic}$, $\sigma'(id_p)(\text{oaw})[j] = \omega$, and simplifying Equation (D.19), we can deduce $\sigma'(id_p)(\text{sm}) \geq \omega = \text{true}$.

Appealing to Lemma 25, $s'.M(p) \geq \omega$.

(b) Assuming $pre(p,t) = (\omega, \text{inhib})$, let us show $s'.M(p) < \omega$.

The proceeding is the same as in the preceding case. Here, we will start the proof where the two cases are diverging, i.e:

By construction, $<\text{output\_arcs\_types(j)} \Rightarrow \text{inhib}> \in i_p$ and $<\text{output\_arcs\_weights(j)} \Rightarrow \omega> \in i_p$.

By property of the stabilize relation and $\text{comp}(id_p, \text{place}, g_p, i_p, o_p) \in d.cs$, we can deduce $\sigma'(id_p)(\text{oat})[j] = \text{inhib}$ and $\sigma'(id_p)(\text{oaw})[j] = \omega$.

Thanks to $\sigma'(id_p)(\text{oat})[j] = \text{inhib}$ and $\sigma'(id_p)(\text{oaw})[j] = \omega$, and simplifying Equation (D.19), we can deduce $\sigma'(id_p)(\text{sm}) < \omega = \text{true}$.

Appealing to Lemma 25, $s'.M(p) < \omega$.

$\square$

Lemma 33 (Rising edge equal not sensitized). *For all sitpn, b, d, γ, E_c, E_p, τ, Δ, σ_e, s, s', σ, σ_i, $\sigma_\uparrow$, σ' that verify the hypotheses of Definition 51, then*
$$\forall t \in T, id_t \in Comps(\Delta) \text{ s.t. } \gamma(t) = id_t, t \notin Sens(s'.M) \Leftrightarrow \sigma'(id_t)(\texttt{s\_enabled}) = \texttt{false}.$$

Proof.

Proving the above lemma is trivial by appealing to Lemma 32 and by reasoning on contrapositives. $\square$

D.4 Falling Edge

D.4.1 Falling edge and marking

Lemma 34 (Falling edge equal marking). *For all sitpn, b, d, γ, Δ, σ_e, E_c, E_p, τ, s, s', σ, $\sigma_\downarrow$, σ' that verify the hypotheses of Definition 45, then $\forall p \in P, id_p \in Comps(\Delta)$ s.t. $\gamma(p) = id_p$, $s'.M(p) = \sigma'(id_p)(\texttt{s\_marking})$.*

Proof.

Given a $p \in P$ and an $id \in Comps(\Delta)$ s.t. $\gamma(p) = id_p$, let us show

$$\boxed{s'.M(p) = \sigma'(id_p)(\texttt{s\_marking}).}$$

By definition of $E_c, \tau \vdash sitpn, s \xrightarrow{\downarrow} s'$, we can deduce $s.M(p) = s'.M(p)$.
By property of the $\mathcal{H}$-VHDL falling edge relation, the stabilize relation and $\texttt{comp}(id_p,$ $\texttt{place}, g_p, i_p, o_p) \in d.cs$, and through the examination of the $\texttt{marking}$ process defined in the $\texttt{place}$ design architecture, we can deduce $\sigma'(id_p)(\texttt{s\_marking}) = \sigma(id_p)(\texttt{s\_marking})$.
Rewriting the goal with $s.M(p) = s'.M(p)$ and $\sigma'(id_p)(\texttt{sm}) = \sigma(id_p)(\texttt{sm})$:

$$\boxed{s.M(p) = \sigma(id_p)(\texttt{sm}).}$$

By definition of $\gamma, E_c, \tau \vdash s \overset{\downarrow}{\approx} \sigma$: $\boxed{s.M(p) = \sigma(id_p)(\texttt{sm}).}$

$\square$

Lemma 35 (Falling edge equal output token sum). *For all sitpn, b, d, γ, Δ, σ_e, E_c, E_p, τ, s, s', σ, $\sigma_\downarrow$, σ' that verify the hypotheses of Definition 45, then $\forall p, id_p$ s.t. $\gamma(p) = id_p$,*
$$\sum_{t \in Fired(s')} pre(p,t) = \sigma'(id_p)(\texttt{s\_output\_token\_sum}).$$

Proof.

Given a $p \in P$ and an $id_p \in Comps(\Delta)$, let us show

$$\boxed{\sum_{t \in Fired(s')} pre(p,t) = \sigma'(id_p)(\texttt{s\_output\_token\_sum}).}$$

By construction and by definition of id_p, there exist g_p, i_p, o_p s.t. $\texttt{comp}(id_p, \texttt{place}, g_p, i_p,$ $o_p) \in d.cs$.
By property of the stabilize relation, $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$, and through the examination of the $\texttt{output\_tokens\_sum}$ process defined in the place design architecture:

$$\sigma'(id_p)(\texttt{sots}) = \sum_{i=0}^{\Delta(id_p)(\texttt{oan})-1} \begin{cases} \sigma'(id_p)(\texttt{oaw})[i] \text{ if } (\sigma'(id_p)(\texttt{otf})[i] \\ \qquad\qquad\qquad . \, \sigma'(id_p)(\texttt{oat})[i] = \texttt{basic}) \\ 0 \text{ } otherwise \end{cases} \quad (D.20)$$

Rewriting the goal with (D.20):

$$\boxed{\sum_{t \in Fired(s')} pre(p,t) = \sum_{i=0}^{\Delta(id_p)(\texttt{oan})-1} \begin{cases} \sigma'(id_p)(\texttt{oaw})[i] \text{ if } (\sigma'(id_p)(\texttt{otf})[i] \\ \qquad\qquad\qquad . \, \sigma'(id_p)(\texttt{oat})[i] = \texttt{basic}) \\ 0 \text{ } otherwise \end{cases}}$$

Let us unfold the definition of the left sum term:

$$\sum_{t \in Fired(s')} \begin{cases} \omega \text{ if } pre(p,t) = (\omega, \texttt{basic}) \\ 0 \text{ } otherwise \end{cases}$$
$$=$$
$$\sum_{i=0}^{\Delta(id_p)(\texttt{oan})-1} \begin{cases} \sigma'(id_p)(\texttt{oaw})[i] \text{ if } (\sigma'(id_p)(\texttt{otf})[i] \\ \qquad\qquad\qquad . \, \sigma'(id_p)(\texttt{oat})[i] = \texttt{basic}) \\ 0 \text{ } otherwise \end{cases}$$

To ease the reading, let us define functions $f \in Fired(s') \to \mathbb{N}$ and $g \in [0, |output(p)| - 1] \to \mathbb{N}$ s.t. $f(t) = \begin{cases} \omega \text{ if } pre(p,t) = (\omega, \texttt{basic}) \\ 0 \text{ } otherwise \end{cases}$

and $g(i) = \begin{cases} \sigma'(id_p)(\texttt{oaw})[i] \text{ if } (\sigma'(id_p)(\texttt{otf})[i] \\ \qquad\qquad\qquad . \, \sigma'(id_p)(\texttt{oat})[i] = \texttt{basic}) \\ 0 \text{ } otherwise \end{cases}$

Then, the goal is: $\boxed{\displaystyle\sum_{t \in Fired(s')} f(t) = \sum_{i=0}^{\Delta(id_p)(\texttt{oan})-1} g(i)}$

Let us perform case analysis on $output(p)$; there are two cases:

– **CASE** $output(p) = \varnothing$:

  By construction, $<\texttt{output\_arcs\_number} \Rightarrow 1> \in g_p$, $<\texttt{output\_arcs\_types(0)} \Rightarrow$ $\texttt{basic}> \in i_p$, $<\texttt{output\_transitions\_fired(0)} \Rightarrow \texttt{true}> \in i_p$, and $<\texttt{output\_arcs\_}$ $\texttt{weights(0)} \Rightarrow 0> \in i_p$.

  By property of the elaboration relation and $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$, we can deduce $\Delta(id_p)(\texttt{oan}) = 1$.

  By property of the stabilize relation and $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$, we can deduce $\sigma'(id_p)(\texttt{oat})[0] = \texttt{basic}$, $\sigma'(id_p)(\texttt{otf})[0] = \texttt{true}$ and $\sigma'(id_p)(\texttt{oaw})[0] = 0$.

  By property of $output(p) = \varnothing$, we can deduce
  $$\sum_{t \in Fired(s')} \begin{cases} \omega \text{ if } pre(p,t) = (\omega, \texttt{basic}) \\ 0 \text{ } otherwise \end{cases} = 0$$
  Rewriting the goal with $\Delta(id_p)(\texttt{oan}) = 1$, $\sigma'(id_p)(\texttt{oat})[0] = \texttt{basic}$, $\sigma'(id_p)(\texttt{otf})[0] =$ $\texttt{true}$, $\sigma'(id_p)(\texttt{oaw})[0] = 0$ and $\displaystyle\sum_{t \in Fired(s')} \begin{cases} \omega \text{ if } pre(p,t) = (\omega, \texttt{basic}) \\ 0 \text{ } otherwise \end{cases} = 0$,

  tautology.

– **CASE** $output(p) \neq \varnothing$:

  By construction, $<\texttt{oan} \Rightarrow |output(p)|> \in g_p$, and by property of the elaboration relation, we can deduce $\Delta(id_p)(\texttt{oan}) = |output(p)|$.

Rewriting the goal with $\Delta(id_p)(\texttt{oan}) = |output(p)|$:
$$\sum_{t \in Fired(s')} f(t) = \sum_{i=0}^{|output(p)|-1} g(i).$$

There exists a mapping, given by the transformation function, between the set $output(p)$ and $[0, |output(p)| - 1]$.

Let $\beta \in output(p) \rightarrow [0, |output(p)| - 1]$ be that mapping.

To prove the current goal, it suffices to show that, for all $t \in Fired(s')$, if $t \in output(p)$ then $f(t) = g(\beta(t))$, and $f(t) = 0$ otherwise.

Given a $t \in Fired(s')$, there are two points to prove:

1. Assuming that $t \in output(p)$, show $f(t) = g(\beta(t))$.
2. Assuming that $t \notin output(p)$, show $f(t) = 0$.

1. Assuming that $t \in output(p)$, let us show $\boxed{f(t) = g(\beta(t)).}$

   Replacing the terms $f(t)$ and $g(\beta(t))$ by their full definition, let us show

   $$\begin{cases} \omega \text{ if } pre(p,t) = (\omega, \texttt{basic}) \\ 0 \text{ } otherwise \end{cases}$$
   $$=$$
   $$\begin{cases} \sigma'(id_p)(\texttt{oaw})[\beta(t)] \text{ if } (\sigma'(id_p)(\texttt{otf})[\beta(t)] \\ \qquad\qquad\qquad . \, \sigma'(id_p)(\texttt{oat})[\beta(t)] = \texttt{basic}) \\ 0 \text{ } otherwise \end{cases}$$

   As $t \in output(p)$, there exist a weight $\omega \in \mathbb{N}$ and an arc type $a \in \{\texttt{basic}, \texttt{test}, \texttt{inhib}\}$ such that $pre(p,t) = (\omega, a)$.

   By construction, we have:

   – $<\texttt{oat}(\beta(t)) \Rightarrow a> \in i_p$
   – $<\texttt{oaw}(\beta(t)) \Rightarrow \omega> \in i_p$

   By property of the stabilize relation and $<\texttt{oat}(\beta(t)) \Rightarrow a> \in i_p$, we have $\sigma'(id_p)(\texttt{oat})[\beta(t)] = a$.

   Let us perform case analysis of the value of a; there are two cases:

   – **CASE** $a = \texttt{inhib}$ or $a = \texttt{test}$:
     In that case, $pre(p,t) \neq (\omega, \texttt{basic})$ and $\sigma'(id_p)(\texttt{oat})[\beta(t)] \neq \texttt{basic}$.
     Thus, the goal can be rewritten as follows: $\boxed{0 = 0}$, tautology.

   – **CASE** $a = \texttt{basic}$:
     In that case, $pre(p,t) = (\omega, \texttt{basic})$ and $\sigma'(id_p)(\texttt{oat})[\beta(t)] = \texttt{basic}$.
     Thus, the goal can be rewritten as follows:

$$\omega = \begin{cases} \sigma'(id_p)(\texttt{oaw})[\beta(t)] \text{ if } \sigma'(id_p)(\texttt{otf})[\beta(t)] \\ 0 \text{ } otherwise \end{cases}$$

By property of the stabilize relation and $<\texttt{oaw}(\beta(t)) \Rightarrow \omega> \in i_p$, we have $\sigma'(id_p)(\texttt{oaw})[\beta(t)] = \omega$. Thus, the goal can be rewritten as follows:

$$\omega = \begin{cases} \omega \text{ if } \sigma'(id_p)(\texttt{otf})[\beta(t)] \\ 0 \text{ } otherwise \end{cases}$$

By construction, there exists an $id_{ft} \in Sigs(\Delta)$ such that:

∗ $<\texttt{fired} \Rightarrow id_{ft}> \in o_t$

∗ $<\texttt{otf}(\beta(t)) \Rightarrow \texttt{id}_{ft}> \in i_p$

Let us take an $id_{ft} \in Sigs(\Delta)$ that verifies the above properties.

By property of the stabilize relation, $<\texttt{fired} \Rightarrow \texttt{id}_{ft}> \in o_t$ and $<\texttt{otf}(\beta(t)) \Rightarrow \texttt{id}_{ft}> \in i_p$, we can deduce $\sigma'(id_p)(\texttt{otf})[\beta(t)] = \sigma'(id_{ft}) = \sigma'(id_t)(\texttt{fired})$.

Thus, the goal can be rewritten as follows:

$$\omega = \begin{cases} \omega \text{ if } \sigma'(id_t)(\texttt{fired}) \\ 0 \text{ } otherwise \end{cases}$$

Appealing to Lemma 4, from $t \in Fired(s')$, we can deduce $\sigma'(id_t)(\texttt{fired}) = \texttt{true}$.

Thus, the goal can be rewritten as follows: $\boxed{\omega = \omega}$, tautology.

2. Assuming that $t \notin output(p)$, let us show $\boxed{f(t) = 0.}$

   Replacing the term $f(t)$ by its full definition, let us show

   $$\begin{cases} \omega \text{ if } pre(p,t) = (\omega, \texttt{basic}) \\ 0 \text{ } otherwise \end{cases} = 0$$

   As $t \notin output(p)$, then $pre(p,t) \neq (\omega, \texttt{basic})$, and we can rewrite the goal as follows: $\boxed{0 = 0}$, tautology.

   $\square$

Lemma 36 (Falling edge equal input token sum). *For all sitpn, b, d, γ, Δ, σ_e, E_c, E_p, τ, s, s', σ, $\sigma_\downarrow$, σ' that verify the hypotheses of Definition 45, then $\forall p, id_p$ s.t. $\gamma(p) = id_p$,*
$$\sum_{t \in Fired(s')} post(t, p) = \sigma'_p(\texttt{s\_input\_token\_sum}).$$

Proof.

Given a $p \in P$ and an $id_p \in Comps(\Delta)$, let us show

$$\boxed{\sum_{t \in Fired(s')} post(t, p) = \sigma'(id_p)(\text{s\_input\_token\_sum}).}$$

By construction and by definition of id_p, there exist g_p, i_p, o_p s.t. $\text{comp}(id_p, \text{place}, g_p, i_p, o_p) \in d.cs$.

By property of the stabilize relation, $\text{comp}(id_p, \text{place}, g_p, i_p, o_p) \in d.cs$, and through the examination of the input_tokens_sum process defined in the place design architecture:

$$\sigma'(id_p)(\text{sits}) = \sum_{i=0}^{\Delta(id_p)(\text{ian})-1} \begin{cases} \sigma'(id_p)(\text{iaw})[i] \text{ if } \sigma'(id_p)(\text{itf})[i] \\ 0 \text{ otherwise} \end{cases} \tag{D.21}$$

Rewriting the goal with (D.21):

$$\boxed{\sum_{t \in Fired(s')} post(t, p) = \sum_{i=0}^{\Delta(id_p)(\text{ian})-1} \begin{cases} \sigma'(id_p)(\text{iaw})[i] \text{ if } \sigma'(id_p)(\text{otf})[i] \\ 0 \text{ otherwise} \end{cases}}$$

Let us unfold the definition of the left sum term:

$$\boxed{\begin{aligned} \sum_{t \in Fired(s')} \begin{cases} \omega \text{ if } post(t, p) = \omega \\ 0 \text{ otherwise} \end{cases} \\ = \\ \sum_{i=0}^{\Delta(id_p)(\text{ian})-1} \begin{cases} \sigma'(id_p)(\text{iaw})[i] \text{ if } \sigma'(id_p)(\text{itf})[i] \\ 0 \text{ otherwise} \end{cases} \end{aligned}}$$

Let us perform case analysis on $input(p)$; there are two cases:

– **CASE** $input(p) = \emptyset$:

By construction, $<\text{input\_arcs\_number} \Rightarrow 1> \in g_p$, $<\text{input\_transitions\_fired(0)} \Rightarrow \text{true}> \in i_p$, and $<\text{input\_arcs\_weights(0)} \Rightarrow 0> \in i_p$.

By property of the elaboration relation and $\text{comp}(id_p, \text{place}, g_p, i_p, o_p) \in d.cs$, we can deduce $\Delta(id_p)(\text{ian}) = 1$.

By property of the stabilize relation and $\text{comp}(id_p, \text{place}, g_p, i_p, o_p) \in d.cs$, we can deduce $\sigma'(id_p)(\text{itf})[0] = \text{true}$ and $\sigma'(id_p)(\text{iaw})[0] = 0$.

By property of $input(p) = \emptyset$, we can deduce $\sum_{t \in Fired(s')} \begin{cases} \omega \text{ if } post(t, p) = \omega \\ 0 \text{ otherwise} \end{cases} = 0$.

Rewriting the goal with $\Delta(id_p)(\texttt{ian}) = 1$, $\sigma'(id_p)(\texttt{itf})[0] = \texttt{true}$, $\sigma'(id_p)(\texttt{iaw})[0] = 0$,

and $\displaystyle\sum_{t \in Fired(s')} \begin{cases} \omega \text{ if } post(t, p) = \omega \\ 0 \text{ otherwise} \end{cases} = 0$, and simplifying the goal: tautology.

- **CASE** $input(p) \neq \emptyset$:

  By construction, $<\texttt{ian} \Rightarrow |input(p)|> \in g_p$, and by property of the elaboration relation, we can deduce $\Delta(id_p)(\texttt{ian}) = |input(p)|$.

  To ease the reading, let us define functions $f \in Fired(s') \rightarrow \mathbb{N}$ and $g \in [0, |input(p)| - 1] \rightarrow \mathbb{N}$ s.t. $f(t) = \begin{cases} \omega \text{ if } post(t, p) = \omega \\ 0 \text{ otherwise} \end{cases}$ and $g(i) = \begin{cases} \sigma'(id_p)(\texttt{iaw})[i] \text{ if } \sigma'(id_p)(\texttt{itf})[i] \\ 0 \text{ otherwise} \end{cases}$

  Then, the goal is: $\boxed{\displaystyle\sum_{t \in Fired(s')} f(t) = \sum_{i=0}^{\Delta(id_p)(\texttt{ian})-1} g(i)}$

  Rewriting the goal with $\Delta(id_p)(\texttt{ian}) = |input(p)|$: $\boxed{\displaystyle\sum_{t \in Fired(s')} f(t) = \sum_{i=0}^{|input(p)|-1} g(i).}$

  There exists a mapping, given by the transformation function, between the set $input(p)$ and $[0, |input(p)| - 1]$.

  Let $\beta \in input(p) \rightarrow [0, |input(p)| - 1]$ be that mapping.

  To prove the current goal, it suffices to show that, for all $t \in Fired(s')$, if $t \in input(p)$ then $f(t) = g(\beta(t))$, and $f(t) = 0$ otherwise.

  Given a $t \in Fired(s')$, there are two points to prove:

  1. Assuming that $t \in input(p)$, show $f(t) = g(\beta(t))$.
  2. Assuming that $t \notin input(p)$, show $f(t) = 0$.

  1. Assuming that $t \in input(p)$, let us show $\boxed{f(t) = g(\beta(t)).}$

     Replacing the terms $f(t)$ and $g(\beta(t))$ by their full definition, let us show

     $\boxed{\begin{cases} \omega \text{ if } post(t, p) = \omega \\ 0 \text{ otherwise} \end{cases} \\ = \\ \begin{cases} \sigma'(id_p)(\texttt{iaw})[\beta(t)] \text{ if } \sigma'(id_p)(\texttt{itf})[\beta(t)] \\ 0 \text{ otherwise} \end{cases}}$

As $t \in input(p)$, there exist a weight $\omega \in \mathbb{N}^*$ such that $post(t,p) = \omega$. Let us take such an ω. Thus, the goal can be rewritten as follows:

$$\omega = \begin{cases} \sigma'(id_p)(\mathtt{iaw})[\beta(t)] \text{ if } \sigma'(id_p)(\mathtt{itf})[\beta(t)] \\ 0 \text{ } otherwise \end{cases}$$

By construction, we have $<\mathtt{iaw}(\beta(t)) \Rightarrow \omega> \in i_p$, and by property of the stabilize relation, we can deduce $\sigma'(id_p)(\mathtt{iaw})[\beta(t)] = \omega$. Thus, the goal can be rewritten as follows:

$$\omega = \begin{cases} \omega \text{ if } \sigma'(id_p)(\mathtt{itf})[\beta(t)] \\ 0 \text{ } otherwise \end{cases}$$

By construction, there exists an $id_{ft} \in Sigs(\Delta)$ such that:

– $<\mathtt{fired} \Rightarrow id_{ft}> \in o_t$

– $<\mathtt{itf}(\beta(t)) \Rightarrow \mathtt{id_{ft}}> \in i_p$

Let us take an $id_{ft} \in Sigs(\Delta)$ that verifies the above properties.

By property of the stabilize relation, $<\mathtt{fired} \Rightarrow \mathtt{id_{ft}}> \in o_t$ and $<\mathtt{itf}(\beta(t)) \Rightarrow \mathtt{id_{ft}}> \in i_p$, we can deduce $\sigma'(id_p)(\mathtt{itf})[\beta(t)] = \sigma'(id_{ft}) = \sigma'(id_t)(\mathtt{fired})$.

Thus, the goal can be rewritten as follows:

$$\omega = \begin{cases} \omega \text{ if } \sigma'(id_t)(\mathtt{fired}) \\ 0 \text{ } otherwise \end{cases}$$

Appealing to Lemma 4, from $t \in Fired(s')$, we can deduce $\sigma'(id_t)(\mathtt{fired}) = \mathtt{true}$.

Thus, the goal can be rewritten as follows: $\boxed{\omega = \omega}$, tautology.

2. Assuming that $t \notin input(p)$, let us show $\boxed{f(t) = 0.}$

    Replacing the term $f(t)$ by its full definition, let us show

$$\begin{cases} \omega \text{ if } post(t,p) = \omega \\ 0 \text{ } otherwise \end{cases} = 0$$

As $t \notin output(p)$, then $post(t,p) \neq \omega$, and we can rewrite the goal as follows: $\boxed{0 = 0}$, tautology.

$\square$

D.4.2 Falling edge and time counters

Lemma 37 (Falling edge equal time counters). *For all sitpn, b, d, γ, Δ, σ_e, E_c, E_p, τ, s, s', σ, $\sigma_{\downarrow}$, σ' that verify the hypotheses of Definition 45, then $\forall t \in T_i, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$,*
$$\left(u(I_s(t)) = \infty \wedge s'.I(t) \leq l(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(\mathtt{s\_time\_counter})\right)$$
$$\wedge\left(u(I_s(t)) = \infty \wedge s'.I(t) > l(I_s(t)) \Rightarrow \sigma'(id_t)(\mathtt{s\_time\_counter}) = l(I_s(t))\right)$$
$$\wedge\left(u(I_s(t)) \neq \infty \wedge s'.I(t) > u(I_s(t)) \Rightarrow \sigma'(id_t)(\mathtt{s\_time\_counter}) = u(I_s(t))\right)$$
$$\wedge\left(u(I_s(t)) \neq \infty \wedge s'.I(t) \leq u(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(\mathtt{s\_time\_counter})\right).$$

Proof.

Given a $t \in T_i$ and an $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, let us show

$$\left(u(I_s(t)) = \infty \wedge s'.I(t) \leq l(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(\mathtt{s\_time\_counter})\right)$$
$$\wedge\left(u(I_s(t)) = \infty \wedge s'.I(t) > l(I_s(t)) \Rightarrow \sigma'(id_t)(\mathtt{s\_time\_counter}) = l(I_s(t))\right)$$
$$\wedge\left(u(I_s(t)) \neq \infty \wedge s'.I(t) > u(I_s(t)) \Rightarrow \sigma'(id_t)(\mathtt{s\_time\_counter}) = u(I_s(t))\right)$$
$$\wedge\left(u(I_s(t)) \neq \infty \wedge s'.I(t) \leq u(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(\mathtt{s\_time\_counter})\right)$$

By construction and by definition of id_t, there exist g_t, i_t, o_t s.t. $\mathtt{comp}(id_t, \mathtt{transition}, g_t, i_t, o_t) \in d.cs$.

By property of the elaboration, $\mathcal{H}$-VHDL rising edge and stabilize relations, $\mathtt{comp}(id_t, \mathtt{transition}, g_t, i_t, o_t) \in d.cs$, and through the examination of the $\mathtt{time\_counter}$ process defined in the transition design architecture, we can deduce:

$$\sigma(id_t)(\mathtt{se}) = \mathtt{true} \wedge \Delta(id_t)(\mathtt{tt}) \neq \mathtt{NOT\_TEMPORAL} \wedge \sigma(id_t)(\mathtt{srtc}) = \mathtt{false}$$
$$\wedge\sigma(id_t)(\mathtt{stc}) < \Delta(id_t)(\mathtt{mtc}) \Rightarrow \sigma'(id_t)(\mathtt{stc}) = \sigma(id_t)(\mathtt{stc}) + 1 \tag{D.22}$$

$$\sigma(id_t)(\mathtt{se}) = \mathtt{true} \wedge \Delta(id_t)(\mathtt{tt}) \neq \mathtt{NOT\_TEMPORAL} \wedge \sigma(id_t)(\mathtt{srtc}) = \mathtt{false}$$
$$\wedge\sigma(id_t)(\mathtt{stc}) \geq \Delta(id_t)(\mathtt{mtc}) \Rightarrow \sigma'(id_t)(\mathtt{stc}) = \sigma(id_t)(\mathtt{stc}) \tag{D.23}$$

$$\sigma(id_t)(\mathtt{se}) = \mathtt{true} \wedge \Delta(id_t)(\mathtt{tt}) \neq \mathtt{NOT\_TEMPORAL}$$
$$\wedge\sigma(id_t)(\mathtt{srtc}) = \mathtt{true} \Rightarrow \sigma'(id_t)(\mathtt{stc}) = 1 \tag{D.24}$$

$$\sigma(id_t)(\mathtt{se}) = \mathtt{false} \vee \Delta(id_t)(\mathtt{tt}) = \mathtt{NOT\_TEMPORAL} \Rightarrow \sigma'(id_t)(\mathtt{stc}) = 0 \tag{D.25}$$

Then, there are 4 points to show:

1. $u(I_s(t)) = \infty \wedge s'.I(t) \leq l(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(\mathtt{s\_time\_counter})$

Assuming $u(I_s(t)) = \infty$ and $s'.I(t) \leq l(I_s(t))$, let us show

$$\boxed{s'.I(t) = \sigma'(id_t)(\texttt{s\_time\_counter}).}$$

Let us perform case analysis on $t \in Sens(s.M)$; there are two cases:

(a) **CASE** $t \notin Sens(s.M)$:

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we can deduce $\sigma(id_t)(\texttt{se}) = \texttt{false}$.

Appealing to (D.25) and $\sigma(id_t)(\texttt{se}) = \texttt{false}$, we can deduce $\sigma'(id_t)(\texttt{stc}) = 0$.

By definition of $E_c, \tau \vdash s \overset{\downarrow}{\rightarrow} s'$ (Rule (3)), we can deduce $s'.I(t) = 0$.

Rewriting the goal with $\sigma'(id_t)(\texttt{stc}) = 0$ and $s'.I(t) = 0$: tautology.

(b) **CASE** $t \in Sens(s.M)$:

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we can deduce $\sigma(id_t)(\texttt{se}) = \texttt{true}$.

By construction, and as $u(I_s(t)) = \infty$, we have $<\texttt{tt} \Rightarrow \texttt{TEMP\_A\_INF}> \in g_t$. By property of the elaboration relation, we have $\Delta(id_t)(\texttt{tt}) = \texttt{TEMP\_A\_INF}$.

Let us perform case analysis on $s.reset_t(t)$; there are two cases:

  i. **CASE** $s.reset_t(t) = \texttt{true}$:

  By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, $\sigma(id_t)(\texttt{srtc}) = \texttt{true}$.

  Appealing to (D.24), $\sigma(id_t)(\texttt{se}) = \texttt{true}$, $\Delta(id_t)(\texttt{tt}) = \texttt{TEMP\_A\_INF}$ and $\sigma(id_t)(\texttt{srtc}) = \texttt{true}$, we can deduce $\sigma'(id_t)(\texttt{stc}) = 1$.

  By definition of $E_c, \tau \vdash s \overset{\downarrow}{\rightarrow} s'$ (Rule (3)), we can deduce $s'.I(t) = 1$.

  Rewriting the goal with $\sigma'(id_t)(\texttt{stc}) = 1$ and $s'.I(t) = 1$: tautology.

  ii. **CASE** $s.reset_t(t) = \texttt{false}$:

  By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $\sigma(id_t)(\texttt{srtc}) = \texttt{false}$.

  As $u(I_s(t)) = \infty$, there exists an $a \in \mathbb{N}^*$ s.t. $I_s(t) = [a, \infty]$. Let us take such an $a \in \mathbb{N}^*$. By construction, $<\texttt{maximal\_time\_counter} \Rightarrow a> \in g_t$, and by property of the elaboration relation, we have $\Delta(id_t)(\texttt{mtc}) = a$.

  By definition of $E_c, \tau \vdash s \overset{\downarrow}{\rightarrow} s'$ (Rule (4)), and knowing that $t \in Sens(s.M)$, $s.reset_t(t) = \texttt{false}$ and $u(I_s(t)) = \infty$, we can deduce $s'.I(t) = s.I(t) + 1$.

  Rewriting the goal with $s'.I(t) = s.I(t) + 1$: $\boxed{s.I(t) + 1 = \sigma'(id_t)(\texttt{stc}).}$

  We assumed that $s'.I(t) \leq l(I_s(t))$, and as $s'.I(t) = s.I(t) + 1$, then $s.I(t) + 1 \leq l(I_s(t))$, then $s.I(t) < l(I_s(t))$, then $s.I(t) < a$ since $a = l(I_s(t))$.

  By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, and knowing that $s.I(t) < l(I_s(t))$ and $u(I_s(t)) = \infty$, we can deduce $s.I(t) = \sigma(id_t)(\texttt{stc})$.

  Appealing to $\Delta(id_t)(\texttt{mtc}) = a$, $s.I(t) = \sigma(id_t)(\texttt{stc})$ and $s.I(t) < a$, we can deduce $\sigma(id_t)(\texttt{stc}) < \Delta(id_t)(\texttt{mtc})$.

  Appealing to (D.22), $\sigma(id_t)(\texttt{stc}) < \Delta(id_t)(\texttt{mtc})$, $\sigma(id_t)(\texttt{srtc}) = \texttt{false}$ and $\sigma(id_t)(\texttt{se}) = \texttt{true}$, we can deduce: $\sigma'(id_t)(\texttt{stc}) = \sigma(id_t)(\texttt{stc}) + 1$.

Rewriting the goal with $\sigma'(id_t)(\texttt{stc}) = \sigma(id_t)(\texttt{stc}) + 1$ and $s.I(t) = \sigma(id_t)(\texttt{stc})$: tautology.

2. $\boxed{u(I_s(t)) = \infty \wedge s'.I(t) > l(I_s(t)) \Rightarrow \sigma'(id_t)(\texttt{s\_time\_counter}) = l(I_s(t)).}$

Assuming that $u(I_s(t)) = \infty$ and $s'.I(t) > l(I_s(t))$, let us show
$\boxed{\sigma'(id_t)(\texttt{s\_time\_counter}) = l(I_s(t)).}$

As $u(I_s(t)) = \infty$, there exists an $a \in \mathbb{N}^*$ s.t. $I_s(t) = [a, \infty]$. Let us take such an $a \in \mathbb{N}^*$.

By construction, $<\texttt{maximal\_time\_counter} \Rightarrow a> \in g_t$, and $<\texttt{transition\_type} \Rightarrow \texttt{TEMP\_A\_INF}> \in g_t$ by property of the elaboration relation, we can deduce $\Delta(id_t)(\texttt{mtc}) = a$ and $\Delta(id_t)(\texttt{tt}) = \texttt{TEMP\_A\_INF}$.

Let us perform case analysis on $t \in Sens(s.M)$:

(a) **CASE** $t \notin Sens(s.M)$:

By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$ (Rule (6)), and knowing that $t \in Sens(s.M)$, we can deduce $s'.I(t) = 0$. Since $l(I_s(t)) \in \mathbb{N}^*$, then $l(I_s(t)) > 0$.

Contradicts $s'.I(t) > l(I_s(t))$.

(b) **CASE** $t \in Sens(s.M)$:

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\sim} \sigma$ and $t \in Sens(s.M)$, we can deduce $\sigma(id_t)(\texttt{se}) = \texttt{true}$.

Let us perform case analysis on $s.reset_t(t)$; there are two cases:

i. **CASE** $s.reset_t(t) = \texttt{true}$:

By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$: $s'.I(t) = 1$.
We assumed that $s'.I(t) > l(I_s(t))$, then $1 > l(I_s(t))$.
Contradicts $l(I_s(t)) > 0$.

ii. **CASE** $s.reset_t(t) = \texttt{false}$:

By property of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$ and $s.reset_t(t) = \texttt{false}$, we can deduce $\sigma(id_t)(\texttt{srtc}) = \texttt{false}$.

By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$ (Rule (4)), and knowing that $s'.I(t) > l(I_s(t))$, we can deduce

$$s'.I(t) = s.I(t) + 1 \Rightarrow s.I(t) + 1 > l(I_s(t))$$
$$\Rightarrow s.I(t) \geq l(I_s(t))$$

Let us perform case analysis on $s.I(t) \geq l(I_s(t))$:
A. **CASE** $s.I(t) > l(I_s(t))$: $\boxed{\sigma'(id_t)(\texttt{stc}) = l(I_s(t)).}$

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we can deduce $\sigma(id_t)(\texttt{stc}) = l(I_s(t))$.

Appealing to (D.23), we can deduce $\sigma'(id_t)(\texttt{stc}) = \sigma(id_t)(\texttt{stc})$.
Rewriting the goal with $\sigma'(id_t)(\texttt{stc}) = \sigma(id_t)(\texttt{stc})$ and $\sigma(id_t)(\texttt{stc}) = l(I_s(t))$: tautology.

 B. **CASE** $s.I(t) = l(I_s(t))$: $\boxed{\sigma'(id_t)(\texttt{stc}) = l(I_s(t)).}$

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we can deduce $s.I(t) = \sigma(id_t)(\texttt{stc})$.
Appealing to (D.23), we can deduce $\sigma'(id_t)(\texttt{stc}) = \sigma(id_t)(\texttt{stc})$.
Rewriting the goal with $\sigma'(id_t)(\texttt{stc}) = \sigma(id_t)(\texttt{stc})$, $s.I(t) = \sigma(id_t)(\texttt{stc})$ and $s.I(t) = l(I_s(t))$: tautology.

3. $\boxed{u(I_s(t)) \neq \infty \wedge s'.I(t) > u(I_s(t)) \Rightarrow \sigma'(id_t)(\texttt{s\_time\_counter}) = u(I_s(t)).}$

Assuming that $u(I_s(t)) \neq \infty$ and $s'.I(t) > u(I_s(t))$, let us show
$\boxed{\sigma'(id_t)(\texttt{s\_time\_counter}) = u(I_s(t)).}$

As $u(I_s(t)) \neq \infty$, there exists an $a \in \mathbb{N}^*$, and a $b \in \mathbb{N}^*$ s.t. $I_s(t) = [a, b]$. Let us take such an a and b.

By construction, $<\texttt{maximal\_time\_counter} \Rightarrow b> \in g_t$ and there exists $tt \in \{\texttt{TEMP\_A\_A}, \texttt{TEMP\_A\_B}\}$ s.t. $<\texttt{transition\_type} \Rightarrow tt> \in g_t$.

By property of the elaboration relation and $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$, we can deduce $\Delta(id_t)(\texttt{mtc}) = b = u(I_s(t))$ and $\Delta(id_t)(\texttt{tt}) \neq \texttt{NOT\_TEMP}$.

Let us perform case analysis on $t \in Sens(s.M)$:

(a) **CASE** $t \notin Sens(s.M)$:

 By definition of $E_c, \tau \vdash s \overset{\downarrow}{\rightarrow} s'$ (Rule (6)), and knowing that $t \in Sens(s.M)$, then $s'.I(t) = 0$. Since $u(I_s(t)) \in \mathbb{N}^*$, then $u(I_s(t)) > 0$.
 Contradicts $s'.I(t) > u(I_s(t))$.

(b) **CASE** $t \in Sens(s.M)$:

 By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$ and $t \in Sens(s.M)$, we can deduce $\sigma(id_t)(\texttt{se}) = \texttt{true}$.

 Let us perform case analysis on $s.reset_t(t)$; there are two cases:

 i. **CASE** $s.reset_t(t) = \texttt{true}$:

  By definition of $E_c, \tau \vdash s \overset{\downarrow}{\rightarrow} s'$ (Rule (3)), we can deduce $s'.I(t) = 1$.
  We assumed that $s'.I(t) > u(I_s(t))$, then we can deduce $1 > u(I_s(t))$.
  Contradicts $u(I_s(t)) > 0$.

 ii. **CASE** $s.reset_t(t) = \texttt{false}$:

  By property of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$ and $s.reset_t(t) = \texttt{false}$, we can deduce $\sigma(id_t)(\texttt{srtc}) = \texttt{false}$.
  Let us perform case analysis on $s.I(t) > u(I_s(t))$ or $s.I(t) \leq u(I_s(t))$:

A. **CASE** $s.I(t) > u(I_s(t))$: $\boxed{\sigma'(id_t)(\texttt{stc}) = u(I_s(t)).}$

By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$(Rule (5)), we can deduce $s'.I(t) = s.I(t)$.

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we can deduce $\sigma(id_t)(\texttt{stc}) = u(I_s(t))$.
Appealing to (D.23), we have $\sigma'(id_t)(\texttt{stc}) = \sigma(id_t)(\texttt{stc})$.
Rewriting the goal with $\sigma'(id_t)(\texttt{stc}) = \sigma(id_t)(\texttt{stc})$ and $\sigma(id_t)(\texttt{stc}) = u(I_s(t))$: tautology.

B. **CASE** $s.I(t) \leq u(I_s(t))$: $\boxed{\sigma'(id_t)(\texttt{stc}) = u(I_s(t)).}$

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we can deduce $s.I(t) = \sigma(id_t)(\texttt{stc})$.
Let us perform case analysis on $s.I(t) \leq u(I_s(t))$; there are two cases:

– **CASE** $s.I(t) = u(I_s(t))$:

Appealing to $\Delta(id_t)(\texttt{mtc}) = b = u(I_s(t))$, $s.I(t) = \sigma(id_t)(\texttt{stc})$ and $s.I(t) = u(I_s(t))$, we can deduce $\Delta(id_t)(\texttt{mtc}) \leq \sigma(id_t)(\texttt{stc})$.
Appealing to $\Delta(id_t)(\texttt{mtc}) \leq \sigma(id_t)(\texttt{stc})$ and (D.23), we can deduce $\sigma'(id_t)(\texttt{stc}) = \sigma(id_t)(\texttt{stc})$.
Rewriting the goal with $\sigma'(id_t)(\texttt{stc}) = \sigma(id_t)(\texttt{stc})$, $s.I(t) = \sigma(id_t)(\texttt{stc})$ and $s.I(t) = u(I_s(t))$: tautology.

– **CASE** $s.I(t) < u(I_s(t))$:

By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$(Rule (4)), we can deduce $s'.I(t) = s.I(t) + 1$.
From $s'.I(t) = s.I(t) + 1$ and $s.I(t) < u(I_s(t))$, we can deduce $s'.I(t) \leq u(I_s(t))$; contradicts $s'.I(t) > u(I_s(t))$.

4. $\boxed{u(I_s(t)) \neq \infty \wedge s'.I(t) \leq u(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(\texttt{s\_time\_counter}).}$

Assuming that $u(I_s(t)) \neq \infty$ and $s'.I(t) \leq u(I_s(t))$, let us show $\boxed{s'.I(t) = \sigma'(id_t)(\texttt{s\_time\_counter}).}$

As $u(I_s(t)) \neq \infty$, there exists an $a \in \mathbb{N}^*$, and a $b \in \mathbb{N}^*$ s.t. $I_s(t) = [a, b]$. Let us take such an a and b.

By construction, $<\texttt{maximal\_time\_counter} \Rightarrow b> \in g_t$ and there exists $tt \in \{\texttt{TEMP\_-A\_A}, \texttt{TEMP\_A\_B}\}$ s.t. $<\texttt{transition\_type} \Rightarrow tt> \in g_t$; by property of the elaboration relation, we can deduce $\Delta(id_t)(\texttt{mtc}) = b = u(I_s(t))$ and $\Delta(id_t)(\texttt{tt}) \neq \texttt{NOT\_TEMP}$.

Let us perform case analysis on $t \in Sens(s.M)$:

(a) **CASE** $t \notin Sens(s.M)$:

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $\sigma(id_t)(\texttt{se}) = \texttt{false}$.

Appealing (D.25) and $\sigma(id_t)(\texttt{se}) = \texttt{false}$, we have $\sigma'(id_t)(\texttt{stc}) = 0$.

By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$(Rule (6)), we have $s'.I(t) = 0$.

Rewriting the goal with $\sigma'(id_t)(\texttt{stc}) = 0$ and $s'.I(t) = 0$: tautology.

(b) **CASE** $t \in Sens(s.M)$:

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $\sigma(id_t)(\texttt{se}) = \texttt{true}$.

Let us perform case analysis on $s.reset_t(t)$:

i. **CASE** $s.reset_t(t) = \texttt{true}$:

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $\sigma(id_t)(\texttt{srtc}) = \texttt{true}$.
Appealing to (D.24), $\Delta(id_t)(\texttt{tt}) \neq \texttt{NOT\_TEMP}$, $\sigma(id_t)(\texttt{se}) = \texttt{true}$ and $\sigma(id_t)(\texttt{srtc}) = \texttt{true}$, we have $\sigma'(id_t)(\texttt{stc}) = 1$.
By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$(Rule (3)), we have $s'.I(t) = 1$.
Rewriting the goal with $\sigma'(id_t)(\texttt{stc}) = 1$ and $s'.I(t) = 1$, tautology.

ii. **CASE** $s.reset_t(t) = \texttt{false}$:

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $\sigma(id_t)(\texttt{srtc}) = \texttt{false}$.
Let us perform case analysis on $s.I(t) > u(I_s(t))$ or $s.I(t) \leq u(I_s(t))$:
A. **CASE** $s.I(t) > u(I_s(t))$:

By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$, we have $s.I(t) = s'.I(t)$, and thus, $s'.I(t) > u(I_s(t))$. Contradicts $s'.I(t) \leq u(I_s(t))$.

B. **CASE** $s.I(t) \leq u(I_s(t))$:

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $s.I(t) = \sigma(id_t)(\texttt{stc})$.
– **CASE** $s.I(t) < u(I_s(t))$:
From $s.I(t) < u(I_s(t))$, $s.I(t) = \sigma(id_t)(\texttt{stc})$ and
$\Delta(id_t)(\texttt{mtc}) = b = u(I_s(t))$, we can deduce $\sigma(id_t)(\texttt{stc}) < \Delta(id_t)(\texttt{mtc})$.
From (D.22), $\sigma(id_t)(\texttt{se}) = \texttt{true}$, $\Delta(id_t)(\texttt{tt}) \neq \texttt{NOT\_TEMP}$, $\sigma(id_t)(\texttt{srtc}) = \texttt{false}$ and $\sigma(id_t)(\texttt{stc}) < \Delta(id_t)(\texttt{mtc})$, we can deduce
$\sigma'(id_t)(\texttt{stc}) = \sigma(id_t)(\texttt{stc}) + 1$.
By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$(Rule (4)), we can deduce $s'.I(t) = s.I(t) + 1$.
Rewriting the goal with $\sigma'(id_t)(\texttt{stc}) = \sigma(id_t)(\texttt{stc}) + 1$ and $s'.I(t) = s.I(t) + 1$, tautology.

– **CASE** $s.I(t) = u(I_s(t))$:

By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$(Rule (4)), we know that $s'.I(t) = s.I(t) + 1$. We assumed that $s'.I(t) \leq u(I_s(t))$; thus, $s.I(t) + 1 \leq u(I_s(t))$. Contradicts $s.I(t) = u(I_s(t))$.

$\square$

D.4.3 Falling edge and condition values

Lemma 38 (Falling edge equal condition values). *For all sitpn, b, d, γ, Δ, σ_e, E_c, E_p, τ, s, s', σ, $\sigma_\downarrow$, σ' that verify the hypotheses of Definition 45, then $\forall c \in \mathcal{C}, id_c \in Ins(\Delta)$ s.t. $\gamma(c) = id_c$, $s'.cond(c) = \sigma'(id_c)$.*

Proof.

Given a $c \in \mathcal{C}$ and an $id_c \in Ins(\Delta)$ s.t. $\gamma(c) = id_c$, let us show $s'.cond(c) = \sigma'(id_c)$.

By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$(Rule (1)), we have $s'.cond(c) = E_c(\tau, c)$.

By definition of $\gamma, E_c, \tau \vdash s \overset{\downarrow}{\approx} \sigma$, we have $\sigma(id_c) = E_c(\tau, c)$
By property of the $\mathcal{H}$-VHDL falling edge, the stabilize relations and $id_c \in Ins(\Delta)$, we have $\sigma'(id_c) = \sigma(id_c) = E_c(\tau, c)$.
Rewriting the goal with $s'.cond(c) = E_c(\tau, c)$ and $\sigma'(id_c) = E_c(\tau, c)$, tautology.

$\square$

D.4.4 Falling edge and action executions

Lemma 39 (Falling edge equal action executions). *For all sitpn, b, d, γ, Δ, σ_e, E_c, E_p, τ, s, s', σ, $\sigma_\downarrow$, σ' that verify the hypotheses of Definition 45, then $\forall a \in \mathcal{A}, id_a \in Outs(\Delta)$ s.t. $\gamma(a) = id_a$, $s'.ex(a) = \sigma'(id_a)$.*

Proof.

Given an $a \in \mathcal{A}$ and an $id_a \in Outs(\Delta)$ s.t. $\gamma(a) = id_a$, let us show $s'.ex(a) = \sigma'(id_a)$.

By property of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$(Rule (2)):

$$s'.ex(a) = \sum_{p \in marked(s.M)} \mathbb{A}(p, a) \tag{D.26}$$

By construction, the generated `action` process is a part of design d's behavior, i.e. there exist an $sl \subseteq Sigs(\Delta)$ and an $ss_a \in ss$ s.t. $\mathtt{ps(action}, \varnothing, sl, ss) \in d.cs$.

By construction id_a is only assigned in the body of the `action` process during the initialization or a falling edge phase.

Let $pls(a)$ be the set of actions associated to action a, i.e. $pls(a) = \{p \in P \mid \mathbb{A}(p,a) = true\}$. Then, depending on $pls(a)$, there are two cases of assignment of output port id_a:

– **CASE** $pls(a) = \varnothing$:

  By construction, $\mathtt{id_a} \Leftarrow \mathtt{false} \in ss_{a\downarrow}$ where $ss_{a\downarrow}$ is the part of the ''`action`'' process body executed during a falling edge phase.

  By property of the $\mathcal{H}$-VHDL falling edge relation, the stabilize relation and $\mathtt{ps(action}, \varnothing, sl, ss_a) \in d.cs$, we can deduce $\sigma'(id_a) = false$.

  By property of $\sum\limits_{p \in marked(s.M)} \mathbb{A}(p,a)$ and $pls(a) = \varnothing$, we can deduce $\sum\limits_{p \in marked(s.M)} \mathbb{A}(p,a) = \mathtt{false}$.

  Rewriting the goal with (D.26), $\sigma'(id_a) = false$ and $\sum\limits_{p \in marked(s.M)} \mathbb{A}(p,a) = \mathtt{false}$,

  tautology.

– **CASE** $pls(a) \neq \varnothing$:

  By construction, $\mathtt{id_a} \Leftarrow \mathtt{id_{mp_0}} + \cdots + \mathtt{id_{mp_n}} \in ss_{a\downarrow}$, where $id_{mp_i} \in Sigs(\Delta)$, $ss_{a\downarrow}$ is the part of the `action` process body executed during the falling edge phase, and $n = |pls(a)| - 1$.

  By property of the $\mathcal{H}$-VHDL falling edge relation, the stabilize relation, and $\mathtt{ps(action}, \varnothing, sl, ss) \in d.cs$:

  $$\sigma'(id_a) = \sigma(id_{mp_0}) + \cdots + \sigma(id_{mp_n}) \tag{D.27}$$

  Rewriting the goal with (D.26) and (D.27):

  $$\boxed{\sum\limits_{p \in marked(s.M)} \mathbb{A}(p,a) = \sigma(id_{mp_0}) + \cdots + \sigma(id_{mp_n}).}$$

  Let us reason on the value of $\sigma(id_{mp_0}) + \cdots + \sigma(id_{mp_n})$; there are two cases:

  – **CASE** $\sigma(id_{mp_0}) + \cdots + \sigma(id_{mp_n}) = \mathtt{true}$:

    Then, we can rewrite the goal as follows: $\boxed{\sum\limits_{p \in marked(s.M)} \mathbb{A}(p,a) = \mathtt{true}.}$

    To prove the above goal, let us show $\boxed{\exists p \in marked(s.M) \text{ s.t. } \mathbb{A}(p,a) = \mathtt{true}.}$

    From $\sigma(id_{mp_0}) + \cdots + \sigma(id_{mp_n}) = \mathtt{true}$, we can deduce that $\exists id_{mp_i} \text{ s.t. } \sigma(id_{mp_i}) = \mathtt{true}$. Let us take an id_{mp_i} s.t. $\sigma(id_{mp_i}) = \mathtt{true}$.

    By construction, there exist a $p \in pls(a)$, an $id_p \in Comps(\Delta)$, g_p, i_p and o_p such that:

* $\gamma(p) = id_p$
* $\mathtt{comp}(id_p, \mathtt{place}, g_p, i_p, o_p) \in d.cs$
* $\mathtt{<marked \Rightarrow id_{mp_i}>} \in o_p$

Let us take such a p, id_p, g_p, i_p and o_p.

By property of stable σ and $\mathtt{comp}(id_p, \mathtt{place}, g_p, i_p, o_p) \in d.cs$, we can deduce $\sigma(id_{mp_i}) = \sigma(id_p)(\mathtt{marked})$.

By property of stable σ, $\mathtt{comp}(id_p, \mathtt{place}, g_p, i_p, o_p) \in d.cs$, and through the examination of the $\mathtt{determine\_marked}$ process defined in the place design architecture, we can deduce:

$$\sigma(id_p)(\mathtt{marked}) = \sigma(id_p)(\mathtt{sm}) > 0 \tag{D.28}$$

From $\sigma(id_{mp_i}) = \sigma(id_p)(\mathtt{marked})$, (D.28) and $\sigma(id_{mp_i}) = \mathtt{true}$, we can deduce that $\sigma(id_p)(\mathtt{marked}) = \mathtt{true}$ and $(\sigma(id_p)(\mathtt{sm}) > 0) = \mathtt{true}$.

By property of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $s.M(p) = \sigma(id_p)(\mathtt{sm})$.

From $s.M(p) = \sigma(id_p)(\mathtt{sm})$ and $(\sigma(id_p)(\mathtt{sm}) > 0) = \mathtt{true}$, we can deduce $p \in marked(s.M)$, i.e. $s.M(p) > 0$.

Let us use p to prove the goal: $\boxed{\mathbb{A}(p, a) = \mathtt{true.}}$

By definition of $p \in pls(a)$, $\boxed{\mathbb{A}(p, a) = \mathtt{true.}}$

- **CASE** $\sigma(id_{mp_0}) + \cdots + \sigma(id_{mp_n}) = \mathtt{false}$:

  Then, we can rewrite the goal as follows: $\boxed{\sum_{p \in marked(s.M)} \mathbb{A}(p, a) = \mathtt{false.}}$

  To prove the above goal, let us show $\boxed{\forall p \in marked(s.M) \text{ s.t. } \mathbb{A}(p, a) = \mathtt{false.}}$

  Given a $p \in marked(s.M)$, let us show $\boxed{\mathbb{A}(p, a) = \mathtt{false.}}$

  Let us perform case analysis on $\mathbb{A}(p, a)$; there are 2 cases:

  * **CASE** $\boxed{\mathbb{A}(p, a) = \mathtt{false.}}$

  * **CASE** $\mathbb{A}(p, a) = \mathtt{true}$:
    By construction, there exist an $id_p \in Comps(\Delta)$, g_{tp}, i_p, o_p and $id_{mp_i} \in Sigs(\Delta)$ such that:
    · $\gamma(p) = id_p$
    · $\mathtt{comp}(id_p, \mathtt{place}, g_p, i_p, o_p) \in d.cs$
    · $\mathtt{<marked \Rightarrow id_{mp_i}>} \in o_p$
    Let us take such a id_p, g_p, i_p, o_p and id_{mp_i}.
    By property of stable σ, $\mathtt{comp}(id_p, \mathtt{place}, g_p, i_p, o_p) \in d.cs$, and $\mathtt{<marked \Rightarrow id_{mp_i}>} \in o_p$, we can deduce $\sigma(id_{mp_i}) = \sigma(id_p)(\mathtt{marked})$.
    By property of stable σ, $\mathtt{comp}(id_p, \mathtt{place}, g_p, i_p, o_p) \in d.cs$, and through the examination of the $\mathtt{determine\_marked}$ process defined in the place design architecture,

we can deduce:
$$\sigma(id_p)(\texttt{marked}) = (\sigma(id_p)(\texttt{sm}) > 0) \tag{D.29}$$

From $\sigma(id_{mp_0}) + \cdots + \sigma(id_{mp_n}) = \texttt{false}$, we can deduce $\sigma(id_{mp_i}) = \texttt{false}$.
From $\sigma(id_p)(\texttt{marked}) = \texttt{false}$, we can deduce $(\sigma(id_p)(\texttt{sm}) > 0) = \texttt{false}$.

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $s.M(p) = \sigma(id_p)(\texttt{sm})$, and thus, we can deduce that $s.M(p) = 0$ (equivalent to $(s.M(p) > 0) = \texttt{false}$).
Contradicts $\boxed{p \in marked(s.M)}$ (i.e, $s.M(p) > 0$).

$\square$

D.4.5   Falling edge and function executions

Lemma 40 (Falling edge equal function executions). *For all sitpn, $b, d, \gamma, \Delta, \sigma_e, E_c, E_p, \tau, s$,*
$s', \sigma, \sigma_\downarrow, \sigma'$ that verify the hypotheses of Definition 45, then $\forall f \in \mathcal{F}, id_f \in Outs(\Delta)$ s.t. $\gamma(f) =$
id_f, $s'.ex(f) = \sigma'(id_f)$.

Proof.

Given an $f \in \mathcal{F}$ and an $id_f \in Outs(\Delta)$ s.t. $\gamma(f) = id_f$, let us show $\boxed{s'.ex(f) = \sigma'(id_f).}$

By property of $E_c, \tau \vdash s \overset{\downarrow}{\rightarrow} s'$, we can deduce $s.ex(f) = s'.ex(f)$.
By construction, id_f is an output port identifier of Boolean type in the $\mathcal{H}$-VHDL design d assigned by the `function` process only during the initialization or during a rising edge phase.
By property of the $\mathcal{H}$-VHDL rising edge, stabilize relations, and the `function` process, we can deduce $\sigma(id_f) = \sigma'(id_f)$.
Rewriting the goal with $s.ex(f) = s'.ex(f)$ and $\sigma(id_f) = \sigma'(id_f)$, $\boxed{s.ex(f) = \sigma(id_f).}$

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, $s.ex(f) = \sigma(id_f).$ $\square$

D.4.6   Falling edge and firable transitions

Lemma 41 (Falling edge equal firable). *For all sitpn, $b, d, \gamma, \Delta, \sigma_e, E_c, E_p, \tau, s, s', \sigma, \sigma_\downarrow$,*
σ' that verify the hypotheses of Definition 45, then $\forall t \in T, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$,
$t \in Firable(s') \Leftrightarrow \sigma'(id_t)(\texttt{s\_firable}) = \texttt{true}$.

Proof.

Given a $t \in T$ and $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, let us show that
$$t \in Firable(s') \Leftrightarrow \sigma'(id_t)(\texttt{s\_firable}) = \texttt{true.}$$

The proof is in two parts:

1. Assuming that $t \in Firable(s')$, let us show $\sigma'(id_t)(\texttt{s\_firable}) = \texttt{true.}$

    Appealing to Lemma 42: $\sigma'(id_t)(\texttt{s\_firable}) = \texttt{true.}$

2. Assuming that $\sigma'(id_t)(\texttt{s\_firable}) = \texttt{true}$, let us show $t \in Firable(s')$.

    Appealing to Lemma 43: $t \in Firable(s')$.

$\square$

Lemma 42 (Falling edge equal firable 1). *For all sitpn, b, d, γ, Δ, σ_e, E_c, E_p, τ, s, s', σ, $\sigma_\downarrow$, σ' that verify the hypotheses of Definition 45, then $\forall t \in T, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, $t \in Firable(s') \Rightarrow \sigma'(id_t)(s\_firable) = \texttt{true.}$*

Proof.

Given a $t \in T$ and $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, and assuming that $t \in Firable(s')$, let us show $\sigma'(id_t)(\texttt{s\_firable}) = \texttt{true.}$

By construction and by definition of id_t, there exist g_t, i_t, o_t s.t. $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$.

By property of the $\mathcal{H}$-VHDL falling edge relation, the stabilize relation, $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$, and through the examination of the $\texttt{firable}$ process defined in the transition design architecture, we can deduce:

$$\sigma'(id_t)(\texttt{sfa}) = \sigma(id_t)(\texttt{se}) \,.\, \sigma(id_t)(\texttt{scc}) \,.\, \texttt{checktc}(\Delta(id_t), \sigma(id_t)) \qquad \text{(D.30)}$$

Term $\texttt{checktc}(\Delta(id_t), \sigma(id_t))$ is defined as follows:

$$\texttt{checktc}(\Delta(id_t), \sigma(id_t))$$
$$=$$

$$\left(\texttt{not } \sigma(id_t)(\texttt{srtc}) \, . \right.$$
$$\left[(\Delta(id_t)(\texttt{tt}) = \texttt{TEMP\_A\_B} \, . \, (\sigma(id_t)(\texttt{stc}) \geq \sigma(id_t)(\texttt{A}) - 1)\right.$$
$$\left. . \, (\sigma(id_t)(\texttt{stc}) \leq \sigma(id_t)(\texttt{B}) - 1))\right. \tag{D.31}$$
$$+ (\Delta(id_t)(\texttt{tt}) = \texttt{TEMP\_A\_A} \, . \, (\sigma(id_t)(\texttt{stc}) = \sigma(id_t)(\texttt{A}) - 1))$$
$$\left.+ (\Delta(id_t)(\texttt{tt}) = \texttt{TEMP\_A\_INF} \, . \, (\sigma(id_t)(\texttt{stc}) \geq \sigma(id_t)(\texttt{A}) - 1))]\right)$$
$$+ \big(\sigma(id_t)(\texttt{srtc}) \, . \, \Delta(id_t)(\texttt{tt}) \neq \texttt{NOT\_TEMP} \, . \, \sigma(id_t)(\texttt{A}) = 1\big)$$
$$+ \Delta(id_t)(\texttt{tt}) = \texttt{NOT\_TEMP}$$

Rewriting the goal with (D.30): $\boxed{\sigma(id_t)(\texttt{se}) \, . \, \sigma(id_t)(\texttt{scc}) \, . \, \texttt{checktc}(\Delta(id_t), \sigma(id_t)) = \texttt{true}.}$
Then, there are three points to prove:

1. $\boxed{\sigma(id_t)(\texttt{se}) = \texttt{true}}$:

    From $t \in \textit{Firable}(s')$, we can deduce $t \in \textit{Sens}(s'.M)$. By definition of $E_c, \tau \vdash s \overset{\downarrow}{\to} s'$, we have $s.M = s'.M$, and thus, we can deduce $t \in \textit{Sens}(s.M)$.

    By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we know that $t \in \textit{Sens}(s.M)$ implies $\sigma(id_t)(\texttt{se}) = \texttt{true}.$

2. $\boxed{\sigma(id_t)(\texttt{scc}) = \texttt{true}}$:

    By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$:

    $$\sigma(id_t)(\texttt{scc}) = \prod_{c \in conds(t)} \begin{cases} E_c(\tau, c) & \textit{if } \mathbb{C}(t, c) = 1 \\ \texttt{not}(E_c(\tau, c)) & \textit{if } \mathbb{C}(t, c) = -1 \end{cases} \tag{D.32}$$

    where $conds(t) = \{c \in \mathcal{C} \mid \mathbb{C}(t, c) = 1 \vee \mathbb{C}(t, c) = -1\}$.

    Rewriting the goal with (D.32): $\boxed{\prod_{c \in conds(t)} \begin{cases} E_c(\tau, c) & \textit{if } \mathbb{C}(t, c) = 1 \\ \texttt{not}(E_c(\tau, c)) & \textit{if } \mathbb{C}(t, c) = -1 \end{cases} = \texttt{true}.}$

    To ease the reading, let us define $f(c) = \begin{cases} E_c(\tau, c) & \textit{if } \mathbb{C}(t, c) = 1 \\ \texttt{not}(E_c(\tau, c)) & \textit{if } \mathbb{C}(t, c) = -1 \end{cases}.$

    Let us reason by induction on the left term of the goal:

- **BASE CASE**: `true = true`.

- **INDUCTION CASE**:

$$\prod_{c' \in conds(t) \setminus \{c\}} f(c') = \texttt{true}$$

$$f(c) \cdot \prod_{c' \in conds(t) \setminus \{c\}} f(c') = \texttt{true}.$$

Rewriting the goal with the induction hypothesis, simplifying the goal, and unfolding the definition of $f(c)$: $\begin{cases} E_c(\tau, c) & if \ \mathbb{C}(t,c) = 1 \\ \texttt{not}(E_c(\tau, c)) & if \ \mathbb{C}(t,c) = -1 \end{cases} = \texttt{true}.$

As $c \in conds(t)$, let us perform case analysis on $\mathbb{C}(t,c) = 1 \lor \mathbb{C}(t,c) = -1$:

(a) **CASE** $\mathbb{C}(t,c) = 1$: $\boxed{E_c(\tau, c) = \texttt{true}.}$

By definition of $t \in Firable(s')$, we can deduce that $s'.cond(c) = \texttt{true}$. By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$ (Rule (1)), we have $s'.cond(c) = E_c(\tau, c)$. Thus, $E_c(\tau, c) = \texttt{true}.$

(b) $\mathbb{C}(t,c) = -1$: $\boxed{\texttt{not } E_c(\tau, c) = \texttt{true}.}$

By definition of $t \in Firable(s')$, we can deduce that $s'.cond(c) = \texttt{false}$. By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$ (Rule (1)), we have $s'.cond(c) = E_c(\tau, c)$. Thus, $\texttt{not } E_c(\tau, c) = \texttt{true}.$

3. $\boxed{\texttt{checktc}(\Delta(id_t), \sigma(id_t)) = \texttt{true}}$:

By definition of $t \in Firable(s')$, we have $t \notin T_i \lor s'.I(t) \in I_s(t)$. Let us perform case analysis on $t \notin T_i \lor s'.I(t) \in I_s(t)$:

(a) **CASE** $t \notin T_i$: $\boxed{\texttt{checktc}(\Delta(id_t), \sigma(id_t)) = \texttt{true}}$

By construction, $<\texttt{transition\_type} \Rightarrow \texttt{NOT\_TEMP}> \in g_t$, and by property of the elaboration relation, we have $\Delta(id_t)(\texttt{tt}) = \texttt{NOT\_TEMP}$.
From $\Delta(id_t)(\texttt{tt}) = \texttt{NOT\_TEMP}$, and by definition of $\texttt{checktc}(\Delta(id_t), \sigma(id_t))$, we can deduce $\texttt{checktc}(\Delta(id_t), \sigma(id_t)) = \texttt{true}.$

(b) **CASE** $s'.I(t) \in I_s(t)$: $\boxed{\texttt{checktc}(\Delta(id_t), \sigma(id_t)) = \texttt{true}}$

From $s'.I(t) \in I_s(t)$, we can deduce that $t \in T_i$. Thus, by construction, there exists $tt \in \{\texttt{TEMP\_A\_B}, \texttt{TEMP\_A\_A}, \texttt{TEMP\_A\_INF}\}$ s.t. $<\texttt{transition\_type} \Rightarrow tt> \in g_t$. By property of the elaboration relation, we have $\Delta(id_t)(\texttt{tt}) = tt$, and thus, we know $\Delta(id_t)(\texttt{tt}) \neq \texttt{NOT\_TEMP}$. Therefore, we can simplify the term $\texttt{checktc}(\Delta(id_t), \sigma(id_t))$ as follows:

$$\texttt{checktc}(\Delta(id_t), \sigma(id_t))$$
$$=$$

$$\begin{aligned}
\Bigg(&\texttt{not } \sigma(id_t)(\texttt{srtc}) \, . \\
&\quad \big[\big(\Delta(id_t)(\texttt{tt}) = \texttt{TEMP\_A\_B} \, . \, (\sigma(id_t)(\texttt{stc}) \geq \sigma(id_t)(\texttt{A}) - 1) \\
&\qquad\qquad\qquad\qquad\qquad\qquad . \, (\sigma(id_t)(\texttt{stc}) \leq \sigma(id_t)(\texttt{B}) - 1)) \\
&\quad + (\Delta(id_t)(\texttt{tt}) = \texttt{TEMP\_A\_A} \, . \\
&\qquad (\sigma(id_t)(\texttt{stc}) = \sigma(id_t)(\texttt{A}) - 1)) \\
&\quad + (\Delta(id_t)(\texttt{tt}) = \texttt{TEMP\_A\_INF} \, . \\
&\qquad (\sigma(id_t)(\texttt{stc}) \geq \sigma(id_t)(\texttt{A}) - 1))\big]\Bigg) \\
&+ \big(\sigma(id_t)(\texttt{srtc}) \, . \, \sigma(id_t)(\texttt{A}) = 1\big)
\end{aligned} \tag{D.33}$$

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $s.reset_t(t) = \sigma(id_t)(\texttt{srtc})$.
Let us perform case analysis on the value $s.reset_t(t)$:

i. **CASE** $s.reset_t(t) = \texttt{true}$: $\boxed{\texttt{checktc}(\Delta(id_t), \sigma(id_t)) = \texttt{true}}$

   From $s.reset_t(t) = \sigma(id_t)(\texttt{srtc})$, we can deduce that $\sigma(id_t)(\texttt{srtc}) = \texttt{true}$.
   From $\sigma(id_t)(\texttt{srtc}) = \texttt{true}$, we can simplify the term $\texttt{checktc}(\Delta(id_t), \sigma(id_t))$ as follows:
   $$\texttt{checktc}(\Delta(id_t), \sigma(id_t)) = \big(\sigma(id_t)(\texttt{A}) = 1\big) \tag{D.34}$$

   Rewriting the goal with (D.34), and simplifying the goal: $\boxed{\sigma(id_t)(\texttt{A}) = 1.}$

   By definition of $E_c, \tau \vdash s \overset{\downarrow}{\to} s'$ (Rule (3)), from $t \in Sens(s.M)$ and $s.reset_t(t) = \texttt{true}$, we can deduce $s'.I(t) = 1$. We know that $s'.I(t) \in I_s(t)$, and thus, we have $1 \in I_s(t)$.
   By definition of $1 \in I_s(t)$, there exist an $a \in \mathbb{N}^*$ and a $ni \in \mathbb{N}^* \sqcup \{\infty\}$ s.t. $I_s(t) = [a, ni]$ and $1 \in [a, ni]$.
   By definition of $1 \in [a, ni]$, we have $a \leq 1$, and since $a \in \mathbb{N}^*$, we can deduce $a = 1$.
   By construction, $<\texttt{time\_A\_value} \Rightarrow a> \in i_t$, and by property of stable σ, we have $\sigma(id_t)(\texttt{A}) = a = 1.$

ii. **CASE** $s.reset_t(t) = \texttt{false}$: $\boxed{\texttt{checktc}(\Delta(id_t), \sigma(id_t)) = \texttt{true}}$

From $s.reset_t(t) = \sigma(id_t)(\texttt{srtc})$, we can deduce $\sigma(id_t)(\texttt{srtc}) = \texttt{false}$. From $\sigma(id_t)(\texttt{srtc}) = \texttt{false}$, we can simplify the term $\texttt{checktc}(\Delta(id_t), \sigma(id_t))$ as follows:

$$\texttt{checktc}(\Delta(id_t), \sigma(id_t))$$
$$=$$
$$\begin{aligned}
&\big(\Delta(id_t)(\texttt{tt}) = \texttt{TEMP\_A\_B} \;.\; (\sigma(id_t)(\texttt{stc}) \geq \sigma(id_t)(\texttt{A}) - 1) \\
&\qquad\qquad\qquad\qquad\qquad .\; (\sigma(id_t)(\texttt{stc}) \leq \sigma(id_t)(\texttt{B}) - 1)\big) \\
&+ (\Delta(id_t)(\texttt{tt}) = \texttt{TEMP\_A\_A} \;.\; (\sigma(id_t)(\texttt{stc}) = \sigma(id_t)(\texttt{A}) - 1)) \\
&+ (\Delta(id_t)(\texttt{tt}) = \texttt{TEMP\_A\_INF} \;.\; (\sigma(id_t)(\texttt{stc}) \geq \sigma(id_t)(\texttt{A}) - 1))
\end{aligned} \tag{D.35}$$

Let us perform case analysis on $I_s(t)$; there are two cases:

– **CASE** $I_s(t) = [a, b]$ where $a, b \in \mathbb{N}^*$; then, either $a = b$ or $a \neq b$:

  – **CASE** $a = b$:

    Then, we have $I_s(t) = [a, a]$, and by construction $\texttt{<transition\_type} \Rightarrow \texttt{TEMP\_A\_A>} \in g_t$. By property of the elaboration relation, we have $\Delta(id_t)(\texttt{tt}) = \texttt{TEMP\_A\_A}$; thus we can simplify the $\texttt{checktc}$ term as follows:

$$\texttt{checktc}(\Delta(id_t), \sigma(id_t)) = (\sigma(id_t)(\texttt{stc}) = \sigma(id_t)(\texttt{A}) - 1) \tag{D.36}$$

    Rewriting the goal with (D.36), and simplifying the goal:
$$\boxed{\sigma(id_t)(\texttt{stc}) = \sigma(id_t)(\texttt{A}) - 1.}$$

    From $s'.I(t) \in [a, a]$, we can deduce that $s'.I(t) = a$. Let us perform case analysis on $s.I(t) < u(I_s(t))$ or $s.I(t) \geq u(I_s(t))$:

    * **CASE** $s.I(t) < u(I_s(t))$:

      By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $s.I(t) = \sigma(id_t)(\texttt{stc})$. By definition of $E_c, \tau \vdash s \overset{\downarrow}{\to} s'$ (Rule (4)), we have $s'.I(t) = s.I(t) + 1$. From $s'.I(t) = a$ and $s'.I(t) = s.I(t) + 1$, we can deduce $a - 1 = s.I(t)$.
      By construction, $\texttt{<time\_A\_value} \Rightarrow a\texttt{>} \in i_t$, and by property of stable σ, we have $\sigma(id_t)(\texttt{A}) = a$.
      Rewriting the goal with $\sigma(id_t)(\texttt{A}) = a$, $s.I(t) = \sigma(id_t)(\texttt{stc})$, and $a - 1 = s.I(t)$: tautology.

    * **CASE** $s.I(t) \geq u(I_s(t))$:
      In the case where $s.I(t) > u(I_s(t))$, then $s.I(t) > a$. By definition of $E_c, \tau \vdash s \overset{\downarrow}{\to} s'$ (Rule (5)), we have $s.I(t) = s'.I(t) = a$. Then, $a > a$ is a contradiction.

      In the case where $s.I(t) = u(I_s(t))$, then $s.I(t) = a$. By definition of $E_c, \tau \vdash s \overset{\downarrow}{\to} s'$ (Rule (4)), we have $s'.I(t) = s.I(t) + 1$. Then, we have

$s'.I(t) = a$ and $s'.I(t) = a+1$. Then, $a = a+1$ is a contradiction.

- **CASE** $a \neq b$: $\boxed{\texttt{checktc}(\Delta(id_t), \sigma(id_t)) = \texttt{true}}$

  Then, we have $I_s(t) = [a,b]$, and by construction $\texttt{<transition\_type} \Rightarrow$ $\texttt{TEMP\_A\_B>} \in g_t$. By property of the elaboration relation, we have $\Delta(id_t)(\texttt{tt}) = \texttt{TEMP\_A\_B}$; thus we can simplify the term $\texttt{checktc}$ as follows:

  $$\texttt{checktc}(\Delta(id_t), \sigma(id_t))$$
  $$= \tag{D.37}$$
  $$(\sigma(id_t)(\texttt{stc}) \geq \sigma(id_t)(\texttt{A}) - 1) \,.\, (\sigma(id_t)(\texttt{stc}) \leq \sigma(id_t)(\texttt{B}) - 1)$$

  Rewriting the goal with (D.37), and simplifying the goal:
  $\boxed{(\sigma(id_t)(\texttt{stc}) \geq \sigma(id_t)(\texttt{A}) - 1) \wedge (\sigma(id_t)(\texttt{stc}) \leq \sigma(id_t)(\texttt{B}) - 1).}$

  Let us perform case analysis on $s.I(t) < u(I_s(t))$ or $s.I(t) \geq u(I_s(t))$:
  * **CASE** $s.I(t) < u(I_s(t))$:

    By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $s.I(t) = \sigma(id_t)(\texttt{stc})$. By definition of $E_c, \tau \vdash s \overset{\downarrow}{\rightarrow} s'$(Rule (4)), we have $s'.I(t) = s.I(t) + 1$. By definition of $s'.I(t) \in [a,b]$:
    $\Rightarrow a \leq s'.I(t) \leq b$.
    $\Rightarrow a \leq s'.I(t) \wedge s'.I(t) \leq b$
    $\Rightarrow a \leq s.I(t) + 1 \wedge s.I(t) + 1 \leq b$
    $\Rightarrow a - 1 \leq s.I(t) \wedge s.I(t) \leq b - 1$
    By construction, $\texttt{<time\_A\_value} \Rightarrow a> \in i_t$ and $\texttt{<time\_B\_value} \Rightarrow b> \in i_t$, and by property of stable σ, we have $\sigma(id_t)(\texttt{A}) = a$ and $\sigma(id_t)(\texttt{B}) = b$.
    Rewriting the goal with $\sigma(id_t)(\texttt{A}) = a$, $\sigma(id_t)(\texttt{B}) = b$ and $s.I(t) = \sigma(id_t)(\texttt{stc})$: $a - 1 \leq s.I(t) \wedge s.I(t) \leq b - 1.$
  * **CASE** $s.I(t) \geq u(I_s(t))$:

    In the case where $s.I(t) > u(I_s(t))$, then $s.I(t) > b$. By definition of $E_c, \tau \vdash s \overset{\downarrow}{\rightarrow} s'$ (Rule (5)), we have $s.I(t) = s'.I(t) = b$. Then, $b > b$ is a contradiction.

    In the case where $s.I(t) = u(I_s(t))$, then $s.I(t) = b$. By definition of $E_c, \tau \vdash s \overset{\downarrow}{\rightarrow} s'$ (Rule (4)), we have $s'.I(t) = s.I(t) + 1$.
    By definition of $s'.I(t) \in [a,b]$, we have $s'.I(t) \leq b$:
    $\Rightarrow s.I(t) + 1 \leq b$
    $\Rightarrow b + 1 \leq b$ is contradiction.

- **CASE** $I_s(t) = [a, \infty]$ where $a \in \mathbb{N}^*$: $\boxed{\texttt{checktc}(\Delta(id_t), \sigma(id_t)) = \texttt{true}}$

By construction $<\texttt{transition\_type} \Rightarrow \texttt{TEMP\_A\_INF}> \in g_t$. By property of the elaboration relation, we have $\Delta(id_t)(\texttt{tt}) = \texttt{TEMP\_A\_INF}$; thus we can simplify the term $\texttt{checktc}$ as follows:

$$\texttt{checktc}(\Delta(id_t), \sigma(id_t)) = (\sigma(id_t)(\texttt{stc}) \geq \sigma(id_t)(\texttt{A}) - 1)) \qquad \text{(D.38)}$$

Rewriting the goal with (D.38), and simplifying the goal:
$\boxed{\sigma(id_t)(\texttt{stc}) \geq \sigma(id_t)(\texttt{A}) - 1.}$

From $s'.I(t) \in [a, \infty]$, we can deduce $a \leq s'.I(t)$. Then, let us perform case analysis on $s.I(t) \leq l(I_s(t))$ or $s.I(t) > l(I_s(t))$:

– **CASE** $s.I(t) \leq l(I_s(t))$:

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $s.I(t) = \sigma(id_t)(\texttt{stc})$.

By definition of $E_c, \tau \vdash s \overset{\downarrow}{\rightarrow} s'$ (Rule (4)), we have $s'.I(t) = s.I(t) + 1$:
$\Rightarrow s'.I(t) \geq a$
$\Rightarrow s.I(t) + 1 \geq a$
$\Rightarrow s.I(t) \geq a - 1$
By construction, $<\texttt{time\_A\_value} \Rightarrow a> \in i_t$, and by property of stable σ, we have $\sigma(id_t)(\texttt{A}) = a$.
Rewriting the goal with $\sigma(id_t)(\texttt{A}) = a$ and $s.I(t) = \sigma(id_t)(\texttt{stc})$:
$\boxed{s.I(t) \geq a - 1.}$

– **CASE** $s.I(t) > l(I_s(t))$:

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $\sigma(id_t)(\texttt{stc}) = l(I_s(t)) = a$.
By construction, $<\texttt{time\_A\_value} \Rightarrow a> \in i_t$, and by property of stable σ, we have $\sigma(id_t)(\texttt{A}) = a$.
Rewriting the goal with $\sigma(id_t)(\texttt{stc}) = a$ and $\sigma(id_t)(\texttt{A}) = a$: $\boxed{a \geq a - 1.}$

$\square$

Lemma 43 (Falling Edge Equal Firable 2). *For all sitpn, b, d, γ, Δ, σ_e, E_c, E_p, τ, s, s', σ, $\sigma_\downarrow$, σ' that verify the hypotheses of Definition 45, then $\forall t \in T, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, $\sigma'(id_t)(\texttt{s\_firable}) = \texttt{true} \Rightarrow t \in Firable(s')$.*

Proof.

Given a $t \in T$ and $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, and assuming that $\sigma'(id_t)(\texttt{s\_firable}) = \texttt{true}$, let us show $\boxed{t \in Firable(s').}$

By construction and by definition of id_t, there exist g_t, i_t, o_t s.t. $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$.

By property of the $\mathcal{H}$-VHDL falling edge relation, the stabilize relation, $\text{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$, and through the examination of the $\texttt{firable}$ process defined in the transition design architecture, we can deduce:

$$\sigma'(id_t)(\texttt{sfa}) = \sigma(id_t)(\texttt{se}) \cdot \sigma(id_t)(\texttt{scc}) \cdot \texttt{checktc}(\Delta(id_t), \sigma(id_t)) = \texttt{true} \qquad \text{(D.39)}$$

From (D.39), we can deduce:

$$\sigma(id_t)(\texttt{se}) = \texttt{true} \qquad \text{(D.40)}$$
$$\sigma(id_t)(\texttt{scc}) = \texttt{true} \qquad \text{(D.41)}$$
$$\texttt{checktc}(\Delta(id_t), \sigma(id_t)) = \texttt{true} \qquad \text{(D.42)}$$

Term $\texttt{checktc}(\Delta(id_t), \sigma(id_t))$ as the same definition as in Lemma Falling edge equal firable 1.

By definition of $t \in Firable(s')$, there are three points to prove:

1. $\boxed{t \in Sens(s'.M)}$

2. $\boxed{\forall c \in \mathcal{C}, \mathbb{C}(t,c) = 1 \Rightarrow s'.cond(c) = \texttt{true} \text{ and } \mathbb{C}(t,c) = -1 \Rightarrow s'.cond(c) = \texttt{false}}$

3. $\boxed{t \notin T_i \vee s'.I(t) \in I_s(t)}$

Let us prove these three points:

1. $\boxed{t \in Sens(s'.M)}$:

   By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$, we have $s.M = s'.M$. Rewriting the goal with $s.M = s'.M$: $\boxed{t \in Sens(s.M).}$

   By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $\sigma(id_t)(\texttt{se}) = \texttt{true} \Leftrightarrow t \in Sens(s.M)$.

   From $\sigma(id_t)(\texttt{se}) = \texttt{true}$, we can deduce: $t \in Sens(s.M).$

2. $\boxed{\forall c \in \mathcal{C}, \mathbb{C}(t,c) = 1 \Rightarrow s'.cond(c) = \texttt{true} \text{ and } \mathbb{C}(t,c) = -1 \Rightarrow s'.cond(c) = \texttt{false}}$

   Given a $c \in \mathcal{C}$, there are two points to prove:

   (a) $\boxed{\mathbb{C}(t,c) = 1 \Rightarrow s'.cond(c) = \texttt{true}.}$

   (b) $\boxed{\mathbb{C}(t,c) = -1 \Rightarrow s'.cond(c) = \texttt{false}.}$

   Let us prove these two points:

   (a) Assuming that $\mathbb{C}(t,c) = 1$, let us show $\boxed{s'.cond(c) = \texttt{true}.}$

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have:

$$\sigma(id_t)(\mathtt{scc}) = \prod_{c' \in conds(t)} \begin{cases} E_c(\tau, c') & \text{if } \mathbb{C}(t, c') = 1 \\ \mathtt{not}(E_c(\tau, c')) & \text{if } \mathbb{C}(t, c') = -1 \end{cases} = \mathtt{true} \qquad \text{(D.43)}$$

where $conds(t) = \{c_i \in \mathcal{C} \mid \mathbb{C}(t, c_i) = 1 \vee \mathbb{C}(t, c_i) = -1\}$.

From $\mathbb{C}(t, c) = 1$, we can deduce $c \in conds(t)$. By definition of the product expression, we have:

$$E_c(\tau, c) \cdot \prod_{c' \in conds(t) \setminus \{c\}} \begin{cases} E_c(\tau, c') & \text{if } \mathbb{C}(t, c') = 1 \\ \mathtt{not}(E_c(\tau, c')) & \text{if } \mathbb{C}(t, c') = -1 \end{cases} = \mathtt{true} \qquad \text{(D.44)}$$

From (D.44), we can deduce that $E_c(\tau, c) = \mathtt{true}$.

By definition of $E_c, \tau \vdash s \overset{\downarrow}{\to} s'$ (Rule (1)), we have $s'.cond(c) = E_c(\tau, c)$.

Rewriting the goal with $s'.cond(c) = E_c(\tau, c)$ and $E_c(\tau, c) = \mathtt{true}$: tautology.

(b) Assuming that $\mathbb{C}(t, c) = -1$, let us show $\boxed{s'.cond(c) = \mathtt{false}.}$

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have:

$$\sigma(id_t)(\mathtt{scc}) = \prod_{c' \in conds(t)} \begin{cases} E_c(\tau, c') & \text{if } \mathbb{C}(t, c') = 1 \\ \mathtt{not}(E_c(\tau, c')) & \text{if } \mathbb{C}(t, c') = -1 \end{cases} = \mathtt{true} \qquad \text{(D.45)}$$

where $conds(t) = \{c' \in \mathcal{C} \mid \mathbb{C}(t, c') = 1 \vee \mathbb{C}(t, c') = -1\}$.

From $\mathbb{C}(t, c) = -1$, we can deduce $c \in conds(t)$. By definition of the product expression, we have:

$$\mathtt{not}\, E_c(\tau, c) \cdot \prod_{c' \in conds(t) \setminus \{c\}} \begin{cases} E_c(\tau, c') & \text{if } \mathbb{C}(t, c') = 1 \\ \mathtt{not}(E_c(\tau, c')) & \text{if } \mathbb{C}(t, c') = -1 \end{cases} = \mathtt{true} \qquad \text{(D.46)}$$

From (D.46), we can deduce that $E_c(\tau, c) = \mathtt{false}$.

By definition of $E_c, \tau \vdash s \overset{\downarrow}{\to} s'$ (Rule (1)), we have $s'.cond(c) = E_c(\tau, c)$.

Rewriting the goal with $s'.cond(c) = E_c(\tau, c)$ and $E_c(\tau, c) = \mathtt{false}$: tautology.

3. $\boxed{t \notin T_i \vee s'.I(t) \in I_s(t)}$

Reasoning on $\mathtt{checktc}(\Delta(id_t), \sigma(id_t)) = \mathtt{true}$, there are 3 cases:

(a) $(\mathtt{not}\, \sigma(id_t)(\mathtt{srtc}) \cdot [\dots]) = \mathtt{true}^a$

(b) $(\sigma(id_t)(\mathtt{srtc}) \cdot \Delta(id_t)(\mathtt{tt}) \neq \mathtt{NOT\_TEMP} \cdot \sigma(id_t)(\mathtt{A}) = 1) = \mathtt{true}$

(c) $\big(\Delta(id_t)(\mathtt{tt}) = \mathtt{NOT\_TEMP}\big) = \mathtt{true}$

(a) **CASE** $\big(\mathtt{not}\ \sigma(id_t)(\mathtt{srtc})\,.\,[\dots]\big) = \mathtt{true}$:

Then, we can deduce $\mathtt{not}\ \sigma(id_t)(\mathtt{srtc}) = \mathtt{true}$ and $[\dots] = \mathtt{true}$.

From $\mathtt{not}\ \sigma(id_t)(\mathtt{srtc}) = \mathtt{true}$, we can deduce $\sigma(id_t)(\mathtt{srtc}) = \mathtt{false}$, and from $[\dots] = \mathtt{true}$, we have three other cases:

i. **CASE** $\big(\Delta(id_t)(\mathtt{tt}) = \mathtt{TEMP\_A\_B}\,.\,(\sigma(id_t)(\mathtt{stc}) \geq \sigma(id_t)(\mathtt{A}) - 1)\,.\,(\sigma(id_t)(\mathtt{stc}) \leq \sigma(id_t)(\mathtt{B}) - 1)\big) = \mathtt{true}$

ii. **CASE** $\big(\Delta(id_t)(\mathtt{tt}) = \mathtt{TEMP\_A\_A}\,.\,(\sigma(id_t)(\mathtt{stc}) = \sigma(id_t)(\mathtt{A}) - 1)\big) = \mathtt{true}$

iii. **CASE** $\big(\Delta(id_t)(\mathtt{tt}) = \mathtt{TEMP\_A\_INF}\,.\,(\sigma(id_t)(\mathtt{stc}) \geq \sigma(id_t)(\mathtt{A}) - 1)\big) = \mathtt{true}$

Let us prove the goal is these three contexts:

i. **CASE** $\big(\Delta(id_t)(\mathtt{tt}) = \mathtt{TEMP\_A\_B}\,.\,(\sigma(id_t)(\mathtt{stc}) \geq \sigma(id_t)(\mathtt{A}) - 1)\,.\,(\sigma(id_t)(\mathtt{stc}) \leq \sigma(id_t)(\mathtt{B}) - 1)\big) = \mathtt{true}$:

Then, converting Boolean equalities into intuitionistic predicates, we have:

– $\Delta(id_t)(\mathtt{tt}) = \mathtt{TEMP\_A\_B}$

– $\sigma(id_t)(\mathtt{stc}) \geq \sigma(id_t)(\mathtt{A}) - 1$

– $\sigma(id_t)(\mathtt{stc}) \leq \sigma(id_t)(\mathtt{B}) - 1$

By property of the elaboration relation, and $\Delta(id_t)(\mathtt{tt}) = \mathtt{TEMP\_A\_B}$, there exist $a, b \in \mathbb{N}^*$ s.t. $I_s(t) = [a, b]$. Let us take such an a and b. Then, let us show $\boxed{s'.I(t) \in I_s(t).}$

Rewriting the goal with $I_s(t) = [a, b]$: $\boxed{s'.I(t) \in [a, b].}$

By construction, $\mathtt{<time\_A\_value} \Rightarrow a\mathtt{>}$ and $\mathtt{<time\_B\_value} \Rightarrow b\mathtt{>}$, and by property of stable σ, we have $\sigma(id_t)(\mathtt{A}) = a$ and $\sigma(id_t)(\mathtt{B}) = b$.

Rewriting the goal with $\sigma(id_t)(\mathtt{A}) = a$ and $\sigma(id_t)(\mathtt{B}) = b$, and by definition of $\in$: $\boxed{\sigma(id_t)(\mathtt{A}) \leq s'.I(t) \leq \sigma(id_t)(\mathtt{B}).}$

Now, let us perform case analysis on $s.I(t) \leq u(I_s(t))$ or $s.I(t) > u(I_s(t))$:

– **CASE** $s.I(t) \leq u(I_s(t))$:

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $s.I(t) = \sigma(id_t)(\mathtt{stc})$.

From $\sigma(id_t)(\mathtt{se}) = \mathtt{true}$, we can deduce $t \in \mathit{Sens}(s.M)$, and from $\sigma(id_t)(\mathtt{srtc}) = \mathtt{false}$, we can deduce $s.\mathit{reset}_t(t) = \mathtt{false}$. Then, by definition of $E_c, \tau \vdash s \overset{\downarrow}{\to} s'$ (Rule (4)), we have $s'.I(t) = s.I(t) + 1$.

$\Rightarrow \boxed{\sigma(id_t)(\mathtt{A}) \leq s.I(t) + 1 \leq \sigma(id_t)(\mathtt{B})}$ (by $s'.I(t) = s.I(t) + 1$)

$\Rightarrow \boxed{\sigma(id_t)(\mathtt{A}) \leq \sigma(id_t)(\mathtt{stc}) + 1 \leq \sigma(id_t)(\mathtt{B})}$ (by $s.I(t) = \sigma(id_t)(\mathtt{stc})$)

$\Rightarrow \boxed{\sigma(id_t)(\mathtt{A}) - 1 \leq \sigma(id_t)(\mathtt{stc}) \leq \sigma(id_t)(\mathtt{B}) - 1}$

We assumed $\sigma(id_t)(\mathtt{stc}) \geq \sigma(id_t)(\mathtt{A}) - 1$ and $\sigma(id_t)(\mathtt{stc}) \leq \sigma(id_t)(\mathtt{B}) - 1$, and thus we can deduce: $\boxed{\sigma(id_t)(\mathtt{A}) - 1 \leq \sigma(id_t)(\mathtt{stc}) \leq \sigma(id_t)(\mathtt{B}) - 1}$

- **CASE** $s.I(t) > u(I_s(t))$:

  By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $\sigma(id_t)(\texttt{stc}) = u(I_s(t)) = b$. Then, from $\sigma(id_t)(\texttt{stc}) \leq \sigma(id_t)(\texttt{B}) - 1$, $\sigma(id_t)(\texttt{stc}) = u(I_s(t)) = b$ and $\sigma(id_t)(\texttt{B}) = b$, we can deduce the following contradiction: $\sigma(id_t)(\texttt{B}) \leq \sigma(id_t)(\texttt{B}) - 1.$

ii. $(\Delta(id_t)(\texttt{tt}) = \texttt{TEMP\_A\_A} \, . \, (\sigma(id_t)(\texttt{stc}) = \sigma(id_t)(\texttt{A}) - 1)) = \texttt{true}$:
    Then, converting Boolean equalities into logic predicates, we have:
    - $\Delta(id_t)(\texttt{tt}) = \texttt{TEMP\_A\_A}$
    - $\sigma(id_t)(\texttt{stc}) = \sigma(id_t)(\texttt{A}) - 1$

    By property of the elaboration relation, and $\Delta(id_t)(\texttt{tt}) = \texttt{TEMP\_A\_A}$, there exist $a \in \mathbb{N}^*$ s.t. $I_s(t) = [a, a]$. Let us take such an a. Then, let us show $s'.I(t) \in I_s(t).$

    Rewriting the goal with $I_s(t) = [a, a]$: $s'.I(t) \in [a, a].$
    By construction, $<\texttt{time\_A\_value} \Rightarrow a>$, and by property of stable σ, we have $\sigma(id_t)(\texttt{A}) = a$.
    Rewriting the goal with $\sigma(id_t)(\texttt{A}) = a$, unfolding the definition of $\in$, and simplifying the goal: $s'.I(t) = \sigma(id_t)(\texttt{A}).$
    Now, let us perform case analysis on $s.I(t) \leq u(I_s(t))$ or $s.I(t) > u(I_s(t))$:
    - **CASE** $s.I(t) \leq u(I_s(t))$:

      By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $s.I(t) = \sigma(id_t)(\texttt{stc})$.
      From $\sigma(id_t)(\texttt{se}) = \texttt{true}$, we can deduce $t \in \textit{Sens}(s.M)$, and from $\sigma(id_t)(\texttt{srtc}) = \texttt{false}$, we can deduce $s.reset_t(t) = \texttt{false}$. Then, by definition of $E_c, \tau \vdash s \overset{\downarrow}{\to} s'$ (Rule (4)), we have $s'.I(t) = s.I(t) + 1$.
      $\Rightarrow$ $s.I(t) + 1 = \sigma(id_t)(\texttt{A})$ (by $s'.I(t) = s.I(t) + 1$)
      $\Rightarrow$ $\sigma(id_t)(\texttt{stc}) + 1 = \sigma(id_t)(\texttt{A})$ (by $s.I(t) = \sigma(id_t)(\texttt{stc})$)
      $\Rightarrow$ $\sigma(id_t)(\texttt{stc}) = \sigma(id_t)(\texttt{A}) - 1$ (assumption)
    - **CASE** $s.I(t) > u(I_s(t))$:

      By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $\sigma(id_t)(\texttt{stc}) = u(I_s(t)) = a$. Then, from $\sigma(id_t)(\texttt{stc}) = \sigma(id_t)(\texttt{A}) - 1$, $\sigma(id_t)(\texttt{stc}) = u(I_s(t)) = a$, $\sigma(id_t)(\texttt{A}) = a$, and $a \in \mathbb{N}^*$, we can derive the following contradiction: $\sigma(id_t)(\texttt{A}) = \sigma(id_t)(\texttt{A}) - 1.$

iii. $(\Delta(id_t)(\texttt{tt}) = \texttt{TEMP\_A\_INF} \, . \, (\sigma(id_t)(\texttt{stc}) \geq \sigma(id_t)(\texttt{A}) - 1)) = \texttt{true}$:
     Then, converting Boolean equalities into logic predicates, we have:
     - $\Delta(id_t)(\texttt{tt}) = \texttt{TEMP\_A\_INF}$
     - $\sigma(id_t)(\texttt{stc}) \geq \sigma(id_t)(\texttt{A}) - 1$

By property of the elaboration relation, and $\Delta(id_t)(\text{tt}) = \text{TEMP\_A\_INF}$, there exist $a \in \mathbb{N}^*$ s.t. $I_s(t) = [a, \infty]$. Let us take such an a. Then, let us show $\boxed{s'.I(t) \in I_s(t).}$

Rewriting the goal with $I_s(t) = [a, \infty]$: $\boxed{s'.I(t) \in [a, \infty].}$

By construction, $<\text{time\_A\_value} \Rightarrow a>$, and by property of stable σ, we have $\sigma(id_t)(\text{A}) = a$.

Rewriting the goal with $\sigma(id_t)(\text{A}) = a$, unfolding the definition of $\in$, and simplifying the goal: $\boxed{\sigma(id_t)(\text{A}) \leq s'.I(t).}$

Now, let us perform case analysis on $s.I(t) \leq l(I_s(t))$ or $s.I(t) > l(I_s(t))$:

– **CASE** $s.I(t) \leq l(I_s(t))$:

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $s.I(t) = \sigma(id_t)(\text{stc})$.
From $\sigma(id_t)(\text{se}) = \text{true}$, we can deduce $t \in \textit{Sens}(s.M)$, and from $\sigma(id_t)(\text{srtc}) = \text{false}$, we can deduce $s.\textit{reset}_t(t) = \text{false}$. Then, by definition of $E_c, \tau \vdash s \overset{\downarrow}{\to} s'$ (Rule (4)), we have $s'.I(t) = s.I(t) + 1$.

$\Rightarrow$ $\boxed{\sigma(id_t)(\text{A}) \leq s.I(t) + 1}$ (by $s'.I(t) = s.I(t) + 1$)

$\Rightarrow$ $\boxed{\sigma(id_t)(\text{A}) \leq \sigma(id_t)(\text{stc}) + 1}$ (by $s.I(t) = \sigma(id_t)(\text{stc})$)

$\Rightarrow$ $\boxed{\sigma(id_t)(\text{A}) - 1 \leq \sigma(id_t)(\text{stc})}$ (assumption)

– **CASE** $s.I(t) > l(I_s(t))$:

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $\sigma(id_t)(\text{stc}) = l(I_s(t)) = a$.
From $\sigma(id_t)(\text{se}) = \text{true}$, we can deduce $t \in \textit{Sens}(s.M)$, and from $\sigma(id_t)(\text{srtc}) = \text{false}$, we can deduce $s.\textit{reset}_t(t) = \text{false}$. Then, by definition of $E_c, \tau \vdash s \overset{\downarrow}{\to} s'$ (Rule (4)), we have $s'.I(t) = s.I(t) + 1$.

$\Rightarrow$ $\boxed{\sigma(id_t)(\text{A}) \leq s.I(t) + 1}$ (by $s'.I(t) = s.I(t) + 1$)

$\Rightarrow$ $\boxed{a \leq s.I(t) + 1}$ (by $\sigma(id_t)(\text{A}) = a$)

$\Rightarrow$ $\boxed{a < s.I(t)}$

$\Rightarrow$ $\boxed{l(I_s(t)) < s.I(t)}$ (assumption)

(b) $\big(\sigma(id_t)(\text{srtc}) \,.\, \Delta(id_t)(\text{tt}) \neq \text{NOT\_TEMP} \,.\, \sigma(id_t)(\text{A}) = 1\big) = \text{true}$

Then, converting Boolean equalities into logic predicates, we have:

– $\sigma(id_t)(\text{srtc}) = \text{true}$
– $\Delta(id_t)(\text{tt}) \neq \text{NOT\_TEMP}$
– $\sigma(id_t)(\text{A}) = 1$

By property of the elaboration relation, and $\Delta(id_t)(\text{tt}) \neq \text{NOT\_TEMP}$, there exist an $a \in \mathbb{N}^*$ and a $ni \in \mathbb{N}^* \sqcup \{\infty\}$ s.t. $I_s(t) = [a, ni]$. Let us take such an a and ni.

By construction, $<\text{time\_A\_value} \Rightarrow a> \in i_t$, and by property of stable σ, we have $\sigma(id_t)(\text{A}) = a$. Thus, we can deduce $a = 1$ and $I_s(t) = [1, ni]$.

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, from $\sigma(id_t)(\mathtt{se}) = \mathtt{true}$, we can deduce $t \in Sens(s.M)$, and from $\sigma(id_t)(\mathtt{srtc}) = \mathtt{true}$, we can deduce $s.reset_t(t) = \mathtt{true}$.

By definition of $E_c, \tau \vdash s \overset{\downarrow}{\rightarrow} s'$ (Rule (3)), $t \in Sens(s.M)$ and $s.reset_t(t) = \mathtt{true}$, we have $s'.I(t) = 1$.

Now, let us show $\boxed{s'.I(t) \in I_s(t)}$.

Rewriting the goal with $s'.I(t) = 1$ and $I_s(t) = [1, ni]$: $\boxed{1 \in [1, ni]}$.

(c) $\big(\Delta(id_t)(\mathtt{tt}) = \mathtt{NOT\_TEMP}\big) = \mathtt{true}$

Let us show $\boxed{t \notin T_i.}$

By property of the elaboration relation and $\Delta(id_t)(\mathtt{tt}) = \mathtt{NOT\_TEMP}$, we have $\boxed{t \notin T_i.}$

$\square$

---

^aSee equation (D.31) for the full definition.

Lemma 44 (Falling edge equal not firable). *For all sitpn, b, d, γ, Δ, σ_e, E_c, E_p, τ, s, s', σ, $\sigma_\downarrow$, σ' that verify the hypotheses of Definition 45, then $\forall t \in T, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, $t \notin Firable(s') \Leftrightarrow \sigma'(id_t)(\mathtt{s\_firable}) = \mathtt{false}$.*

Proof.

Proving the above lemma is trivial by appealing to Lemma 41 and by reasoning on contrapositives. $\square$

D.4.7 Falling edge and fired transitions

Lemma 45 (Falling edge equal fired set). *For all sitpn, b, d, γ, Δ, σ_e, E_c, E_p, τ, s, s', σ, $\sigma_\downarrow$, σ' that verify the hypotheses of Definition 45, then $\forall t \in T$, $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, $\forall Fset \subseteq T$, s.t. $IsFiredSet(s', Fset)$, $t \in Fset \Leftrightarrow \sigma'(id_t)(\mathit{fired}) = \mathtt{true}$.*

Proof.

Given a $t \in T$, and $id_t \in Comps(\Delta)$, and a $Fset \subseteq T$ s.t. $IsFiredSet(s', Fset)$, let us show $\boxed{t \in Fset \Leftrightarrow \sigma'(id_t)(\mathtt{fired}) = \mathtt{true}.}$

By definition of $IsFiredSet(s', Fset)$, we have $IsFiredSetAux(s', T, \emptyset, Fset)$. Then, we can appeal to Lemma 46 to solve the goal, but first we must prove the following *extra hypothesis* (i.e, one of the premise of Lemma 46):

$\forall t' \in T, id_{t'} \in Comps(\Delta)$ s.t. $\gamma(t') = id_{t'}$,
$(t' \in \emptyset \Rightarrow \sigma'(id_{t'})(\texttt{fired}) = \texttt{true}) \wedge (\sigma'(id_{t'})(\texttt{fired}) = \texttt{true} \Rightarrow t' \in \emptyset \vee t' \in T)$.

Given a $t' \in T$ and an $id_{t'} \in Comps(\Delta)$ s.t. $\gamma(t') = id_{t'}$, there are two points to prove:

1. $t' \in \emptyset \Rightarrow \sigma'(id_{t'})(\texttt{fired}) = \texttt{true}$

2. $\sigma'(id_{t'})(\texttt{fired}) = \texttt{true} \Rightarrow t' \in \emptyset \vee t' \in T$

Let us show these two points:

1. Assuming $t' \in \emptyset$, let us show $\sigma'(id_{t'})(\texttt{fired}) = \texttt{true}.$

   $t' \in \emptyset$ is a contradiction.

2. Assuming $\sigma'(id_{t'})(\texttt{fired}) = \texttt{true}$, let us show $t' \in \emptyset \vee t' \in T.$

   By definition, $t' \in T.$

   □

Lemma 46 (Falling edge equal fired set aux). *For all sitpn, b, d, γ, Δ, σ_e, E_c, E_p, τ, s, s', σ, $\sigma_\downarrow$, σ' that verify the hypotheses of Definition 45, then $\forall t \in T, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, $\forall F \subseteq T$, $T_s \subseteq T$, $Fset \subseteq T$, assume that:*

– *$IsFiredSetAux(s', T_s, F, Fset)$*

– *EH (Extra. Hypothesis):*
  $\forall t' \in T$, $id_{t'} \in Comps(\Delta)$ s.t. $\gamma(t') = id_{t'}$,
  $(t' \in F \Rightarrow \sigma'(id_{t'})(fired) = \texttt{true}) \wedge (\sigma'(id_{t'})(fired) = \texttt{true} \Rightarrow t' \in F \vee t' \in T_s).$

then $t \in Fset \Leftrightarrow \sigma'(id_t)(fired) = \texttt{true}$.

Proof.

Given a $t \in T$, an $id_t \in Comps(\Delta)$, a $T_s, F, Fset \subseteq T$, and assuming $IsFiredSetAux(s', T_s, F, Fset)$, let us show

$(\forall t' \in T$, $id_{t'} \in Comps(\Delta)$ s.t. $\gamma(t') = id_{t'}$,
$(t' \in F \Rightarrow \sigma'(id_{t'})(\texttt{fired}) = \texttt{true}) \wedge (\sigma'(id_{t'})(\texttt{fired}) = \texttt{true} \Rightarrow t' \in F \vee t' \in T_s)) \Rightarrow t \in Fset \Leftrightarrow \sigma'(id_t)(\texttt{fired}) = \texttt{true}$.

Let us use rule induction on $IsFiredSetAux(s', T_s, F, Fset)$. Let us define the property P taken into account in the induction scheme as follows

$$P(s', T_s, F, Fset)$$
$$\equiv$$
$$(t' \in F \Rightarrow \sigma'(id_{t'})(\texttt{fired}) = \texttt{true}) \land (\sigma'(id_{t'})(\texttt{fired}) = \texttt{true} \Rightarrow t' \in F \lor t' \in T_s)) \Rightarrow$$
$$t \in Fset \Leftrightarrow \sigma'(id_t)(\texttt{fired}) = \texttt{true}$$

– **CASE** FSETEMP: we must show $P(s', \varnothing, F, F)$, i.e.

> $(\forall t' \in T, id_{t'} \in Comps(\Delta)$ s.t. $\gamma(t') = id_{t'}$,
> $(t' \in F \Rightarrow \sigma'(id_{t'})(\texttt{fired}) = \texttt{true}) \land (\sigma'(id_{t'})(\texttt{fired}) = \texttt{true} \Rightarrow t' \in F \lor t' \in$
> $\varnothing)) \Rightarrow t \in F \Leftrightarrow \sigma'(id_t)(\texttt{fired}) = \texttt{true}$.

Assuming

> $\forall t' \in T, id_{t'} \in Comps(\Delta)$ s.t. $\gamma(t') = id_{t'}$,
> $(t' \in F \Rightarrow \sigma'(id_{t'})(\texttt{fired}) = \texttt{true}) \land (\sigma'(id_{t'})(\texttt{fired}) = \texttt{true} \Rightarrow t' \in F \lor t' \in \varnothing)$

we can easily show $t \in F \Leftrightarrow \sigma'(id_t)(\texttt{fired}) = \texttt{true}$.

– **CASE** FSETFIRED:

Assuming

– $t \in Firable(s')$

– $t \in Sens(s'.M - \sum_{t_i \in Pr(t,F)} pre(t_i))$

– $IsFiredSetAux(s', T_s, F \cup \{t\}, Fset)$

– $\nexists t' \in T_s$ s.t. $t' \succ t$

– $Pr(t, F) = \{t' \mid t' \succ t \land t' \in F\}$

and the induction hypothesis (i.e. $P(s', T_s, F \cup \{t\}, Fset)$)

> $(\forall t' \in T, id_{t'} \in Comps(\Delta)$ s.t. $\gamma(t') = id_{t'}$,
> $(t' \in F \cup \{t\} \Rightarrow \sigma'(id_{t'})(\texttt{fired}) = \texttt{true})$
> $\land (\sigma'(id_{t'})(\texttt{fired}) = \texttt{true} \Rightarrow t' \in F \cup \{t\} \lor t' \in T_s)) \Rightarrow$
> $t \in Fset \Leftrightarrow \sigma'(id_t)(\texttt{fired}) = \texttt{true}$

we must show

> $(\forall t' \in T, id_{t'} \in Comps(\Delta)$ s.t. $\gamma(t') = id_{t'}$,
> $(t' \in F \Rightarrow \sigma'(id_{t'})(\texttt{fired}) = \texttt{true})$
> $\land (\sigma'(id_{t'})(\texttt{fired}) = \texttt{true} \Rightarrow t' \in F \lor t' \in T_s \cup \{t\})) \Rightarrow$
> $t \in Fset \Leftrightarrow \sigma'(id_t)(\texttt{fired}) = \texttt{true}$

Assuming the following hypothesis that we will call EH (for Extra Hypothesis)

$\forall t' \in T,\ id_{t'} \in Comps(\Delta)$ s.t. $\gamma(t') = id_{t'}$,
$(t' \in F \Rightarrow \sigma'(id_{t'})(\texttt{fired}) = \texttt{true}) \wedge (\sigma'(id_{t'})(\texttt{fired}) = \texttt{true} \Rightarrow t' \in F \vee t' \in T_s \cup \{t\})$

we must show

$t \in Fset \Leftrightarrow \sigma'(id_t)(\texttt{fired}) = \texttt{true}$

Appealing to the induction hypothesis, to prove the current goal, it is sufficient to prove that

$\forall t' \in T,\ id_{t'} \in Comps(\Delta)$ s.t. $\gamma(t') = id_{t'}$,
$(t' \in F \cup \{t\} \Rightarrow \sigma'(id_{t'})(\texttt{fired}) = \texttt{true})$
$\wedge (\sigma'(id_{t'})(\texttt{fired}) = \texttt{true} \Rightarrow t' \in F \cup \{t\} \vee t' \in T_s)$

Given a $t' \in T$, an $id_{t'} \in Comps(\Delta)$ s.t. $\gamma(t') = id_{t'}$, we must show that

$(t' \in F \cup \{t\} \Rightarrow \sigma'(id_{t'})(\texttt{fired}) = \texttt{true})$
$\wedge (\sigma'(id_{t'})(\texttt{fired}) = \texttt{true} \Rightarrow t' \in F \cup \{t\} \vee t' \in T_s)$

There are two points to prove

1. Assuming $t' \in F \cup \{t\}$, then $\sigma'(id_{t'})(\texttt{fired}) = \texttt{true}$
2. Assuming $\sigma'(id_{t'})(\texttt{fired}) = \texttt{true}$, then $t' \in F \cup \{t\} \vee t' \in T_s$

1. Assuming $t' \in F \cup \{t\}$, let us show $\boxed{\sigma'(id_{t'})(\texttt{fired}) = \texttt{true}}$. Let us perform case analysis on $t' \in F \cup \{t\}$; there are 2 cases:
   - **CASE** $t' \in F$: Appealing to EH, the goal is trivially proved.
   - **CASE** $t' = t$: Then, $id_t = id_{t'}$, and we must show $\boxed{\sigma'(id_t)(\texttt{fired}) = \texttt{true}}$.
     By definition of id_t, there exist a g_t, i_t, o_t s.t. $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$.
     By property of the stabilize relation and $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$, and through the examination of the $\texttt{fired\_evaluation}$ process defined in the $\texttt{transition}$ design architecture:

     $$\sigma(id_t)(\texttt{fired}) = \sigma(id_t)(\texttt{sfa}) \,.\, \sigma(id_t)(\texttt{spc})$$

     Rewriting the goal with the above equation: $\boxed{\sigma(id_t)(\texttt{sfa}) \,.\, \sigma(id_t)(\texttt{spc}) = \texttt{true}.}$
     Then, there are two points to prove:

(a) $\boxed{\sigma(id_t)(\texttt{sfa}) = \texttt{true.}}$

Appealing to Lemma 41, and since $t \in Firable(s')$, we can deduce $\boxed{\sigma(id_t)(\texttt{sfa}) = \texttt{true.}}$

(b) $\boxed{\sigma(id_t)(\texttt{spc}) = \texttt{true.}}$

Appealing to Lemma 47, and since $t \in Sens(s'M - \sum\limits_{t_i \in Pr(t,F)} pre(t_i))$, we can

deduce $\boxed{\sigma(id_t)(\texttt{spc}) = \texttt{true.}}$

2. Assuming $\sigma'(id_{t'})(\texttt{fired}) = \texttt{true}$, let us show $\boxed{t' \in F \cup \{t\} \; \lor \; t' \in T_s}$. Appealing to EH, we can deduce that $t' \in F \; \lor \; t' \in T_s \cup \{t\}$. Then, the goal is trivially shown.

– **CASE** FSETNOTFIRABLE: Assuming

   – $t \notin Firable(s')$
   – $IsFiredSetAux(s', T_s, F, Fset)$
   – $\nexists t' \in T_s$ s.t. $t' \succ t$

   and the induction hypothesis (i.e. $P(s', T_s, F, Fset)$)

$$
\begin{aligned}
&(\forall t' \in T, \; id_{t'} \in Comps(\Delta) \text{ s.t. } \gamma(t') = id_{t'}, \\
&(t' \in F \Rightarrow \sigma'(id_{t'})(\texttt{fired}) = \texttt{true}) \\
&\land \; (\sigma'(id_{t'})(\texttt{fired}) = \texttt{true} \Rightarrow t' \in F \; \lor \; t' \in T_s)) \Rightarrow \\
&t \in Fset \Leftrightarrow \sigma'(id_t)(\texttt{fired}) = \texttt{true}
\end{aligned}
$$

we must show

$$
\begin{aligned}
&(\forall t' \in T, \; id_{t'} \in Comps(\Delta) \text{ s.t. } \gamma(t') = id_{t'}, \\
&(t' \in F \Rightarrow \sigma'(id_{t'})(\texttt{fired}) = \texttt{true}) \\
&\land \; (\sigma'(id_{t'})(\texttt{fired}) = \texttt{true} \Rightarrow t' \in F \; \lor \; t' \in T_s \cup \{t\})) \Rightarrow \\
&t \in Fset \Leftrightarrow \sigma'(id_t)(\texttt{fired}) = \texttt{true}
\end{aligned}
$$

Assuming the following hypothesis that we will call EH (for Extra Hypothesis)

$$
\begin{aligned}
&\forall t' \in T, \; id_{t'} \in Comps(\Delta) \text{ s.t. } \gamma(t') = id_{t'}, \\
&(t' \in F \Rightarrow \sigma'(id_{t'})(\texttt{fired}) = \texttt{true}) \land (\sigma'(id_{t'})(\texttt{fired}) = \texttt{true} \Rightarrow t' \in F \; \lor \; t' \in T_s \cup \{t\})
\end{aligned}
$$

we must show

$$t \in \mathit{Fset} \Leftrightarrow \sigma'(id_t)(\mathtt{fired}) = \mathtt{true}$$

Appealing to the induction hypothesis, to prove the current goal, it is sufficient to prove that

$$\forall t' \in T, \; id_{t'} \in \mathit{Comps}(\Delta) \text{ s.t. } \gamma(t') = id_{t'},$$
$$(t' \in F \Rightarrow \sigma'(id_{t'})(\mathtt{fired}) = \mathtt{true}) \land (\sigma'(id_{t'})(\mathtt{fired}) = \mathtt{true} \Rightarrow t' \in F \lor t' \in T_s)$$

Given a $t' \in T$, an $id_{t'} \in \mathit{Comps}(\Delta)$ s.t. $\gamma(t') = id_{t'}$, we must show that

$$(t' \in F \Rightarrow \sigma'(id_{t'})(\mathtt{fired}) = \mathtt{true}) \land (\sigma'(id_{t'})(\mathtt{fired}) = \mathtt{true} \Rightarrow t' \in F \lor t' \in T_s)$$

There are two points to prove

1. Assuming $t' \in F$, then $\sigma'(id_{t'})(\mathtt{fired}) = \mathtt{true}$
2. Assuming $\sigma'(id_{t'})(\mathtt{fired}) = \mathtt{true}$, then $t' \in F \lor t' \in T_s$

1. Assuming $t' \in F$, let us show $\boxed{\sigma'(id_{t'})(\mathtt{fired}) = \mathtt{true}}$.
    Appealing to EH, the goal is trivially shown.

2. Assuming $\sigma'(id_{t'})(\mathtt{fired}) = \mathtt{true}$, let us show $\boxed{t' \in F \lor t' \in T_s}$.
    Appealing to EH, we can deduce $t' \in F \lor t' \in T_s \cup \{t\}$. Let us perform case analysis on $t' \in F \lor t' \in T_s \cup \{t\}$; there are 2 cases:

    - **CASE** $t' \in F$: trivially shown, as it is an assumption.
    - **CASE** $t' \in T_s \cup \{t\}$: In the case where $t' \in T_s$, the goal is trivially shown. In the case where $t' = t$, we can prove a contradiction based on $t \notin \mathit{Firable}(s')$ and $\sigma'(id_{t'})(\mathtt{fired}) = \mathtt{true}$.
      Since $t = t'$, then $id_t = id_{t'}$, and we know that $\sigma'(id_t)(\mathtt{fired}) = \mathtt{true}$.
      By definition of id_t, there exist a g_t, i_t, o_t s.t. $\mathtt{comp}(id_t, \mathtt{transition}, g_t, i_t, o_t) \in d.cs$.
      By property of the stabilize relation and $\mathtt{comp}(id_t, \mathtt{transition}, g_t, i_t, o_t) \in d.cs$, and through the examination of the $\mathtt{fired\_evaluation}$ process defined in the $\mathtt{transition}$ design architecture, we can deduce

      $$\sigma(id_t)(\mathtt{fired}) = \sigma(id_t)(\mathtt{sfa}) \, . \, \sigma(id_t)(\mathtt{spc}) = \mathtt{true}$$

      Thus, we have
      $$\sigma(id_t)(\mathtt{sfa}) = \mathtt{true}$$

      and, appealing to Lemma 41, we can deduce $t \in \mathit{Firable}(s')$, which directly contradicts $t \notin \mathit{Firable}(s')$.

– **CASE** FSETNOTSENS: Assuming

   – $t \notin Sens(s'.M - \sum\limits_{t_i \in Pr(t,F)} pre(t_i))$

   – $IsFiredSetAux(s', T_s, F, Fset)$

   – $\nexists t' \in T_s \ s.t. \ t' \succ t$

   – $Pr(t, F) = \{t' \mid t' \succ t \wedge t' \in F\}$

and the induction hypothesis (i.e. $P(s', T_s, F, Fset)$)

$$\begin{aligned}
&\big(\forall t' \in T, \ id_{t'} \in Comps(\Delta) \ \text{s.t.} \ \gamma(t') = id_{t'}, \\
&(t' \in F \Rightarrow \sigma'(id_{t'})(\texttt{fired}) = \texttt{true}) \\
&\wedge (\sigma'(id_{t'})(\texttt{fired}) = \texttt{true} \Rightarrow t' \in F \ \vee \ t' \in T_s)\big) \Rightarrow \\
&t \in Fset \Leftrightarrow \sigma'(id_t)(\texttt{fired}) = \texttt{true}
\end{aligned}$$

we must show

$$\begin{aligned}
&\big(\forall t' \in T, \ id_{t'} \in Comps(\Delta) \ \text{s.t.} \ \gamma(t') = id_{t'}, \\
&(t' \in F \Rightarrow \sigma'(id_{t'})(\texttt{fired}) = \texttt{true}) \\
&\wedge (\sigma'(id_{t'})(\texttt{fired}) = \texttt{true} \Rightarrow t' \in F \ \vee \ t' \in T_s \cup \{t\})\big) \Rightarrow \\
&t \in Fset \Leftrightarrow \sigma'(id_t)(\texttt{fired}) = \texttt{true}
\end{aligned}$$

Assuming the following hypothesis, which we will call EH (for Extra Hypothesis)

$$\begin{aligned}
&\forall t' \in T, \ id_{t'} \in Comps(\Delta) \ \text{s.t.} \ \gamma(t') = id_{t'}, \\
&(t' \in F \Rightarrow \sigma'(id_{t'})(\texttt{fired}) = \texttt{true}) \wedge (\sigma'(id_{t'})(\texttt{fired}) = \texttt{true} \Rightarrow t' \in F \ \vee \ t' \in T_s \cup \{t\})
\end{aligned}$$

we must show

$$t \in Fset \Leftrightarrow \sigma'(id_t)(\texttt{fired}) = \texttt{true}$$

Appealing to the induction hypothesis, to prove the current goal, it is sufficient to prove that

$$\begin{aligned}
&\forall t' \in T, \ id_{t'} \in Comps(\Delta) \ \text{s.t.} \ \gamma(t') = id_{t'}, \\
&(t' \in F \Rightarrow \sigma'(id_{t'})(\texttt{fired}) = \texttt{true}) \wedge (\sigma'(id_{t'})(\texttt{fired}) = \texttt{true} \Rightarrow t' \in F \ \vee \ t' \in T_s)
\end{aligned}$$

Given a $t' \in T$, an $id_{t'} \in Comps(\Delta)$ s.t. $\gamma(t') = id_{t'}$, we must show that

$$(t' \in F \Rightarrow \sigma'(id_{t'})(\texttt{fired}) = \texttt{true}) \wedge (\sigma'(id_{t'})(\texttt{fired}) = \texttt{true} \Rightarrow t' \in F \vee t' \in T_s)$$

There are two points to prove

1. Assuming $t' \in F$, then $\sigma'(id_{t'})(\texttt{fired}) = \texttt{true}$
2. Assuming $\sigma'(id_{t'})(\texttt{fired}) = \texttt{true}$, then $t' \in F \vee t' \in T_s$

1. Assuming $t' \in F$, let us show $\boxed{\sigma'(id_{t'})(\texttt{fired}) = \texttt{true}}$.

   Appealing to EH, the goal is trivially shown.

2. Assuming $\sigma'(id_{t'})(\texttt{fired}) = \texttt{true}$, let us show $\boxed{t' \in F \vee t' \in T_s}$.

   Appealing to EH, we can deduce $t' \in F \vee t' \in T_s \cup \{t\}$. Let us perform case analysis on $t' \in F \vee t' \in T_s \cup \{t\}$; there are 2 cases:

   - **CASE** $t' \in F$: trivially shown, as it is an assumption.
   - **CASE** $t' \in T_s \cup \{t\}$: In the case where $t' \in T_s$, the goal is trivially shown. In the case where $t' = t$, we can prove a contradiction based on $t \notin Sens(s'.M - \sum\limits_{t_i \in Pr(t,F)} pre(t_i))$ and $\sigma'(id_{t'})(\texttt{fired}) = \texttt{true}$.

     Since $t = t'$, then $id_t = id_{t'}$, and we know that $\sigma'(id_t)(\texttt{fired}) = \texttt{true}$.
     By definition of id_t, there exist a g_t, i_t, o_t s.t. $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$. By property of the stabilize relation and $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$, and through the examination of the $\texttt{fired\_evaluation}$ process defined in the $\texttt{transition}$ design architecture, we can deduce

     $$\sigma(id_t)(\texttt{fired}) = \sigma(id_t)(\texttt{sfa}) . \sigma(id_t)(\texttt{spc}) = \texttt{true}$$

   Thus, we have
   $$\sigma(id_t)(\texttt{spc}) = \texttt{true}$$

   and, appealing to Lemma 47, we can deduce $t \in Sens(s'.M - \sum\limits_{t_i \in Pr(t,F)} pre(t_i))$,

   which directly contradicts $t \notin Sens(s'.M - \sum\limits_{t_i \in Pr(t,F)} pre(t_i))$.

   $\square$

Lemma 47 (Stabilize compute priority combination after falling edge). *For all sitpn, b, d, $\gamma, \Delta, \sigma_e, E_c, E_p, \tau, s, s', \sigma, \sigma_{\downarrow}, \sigma'$ that verify the hypotheses of Definition 45, then $\forall t \in T, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t, \forall T_s, F, Fset \subseteq T$ assume that:*

− $t \in Firable(s')$

− $\nexists t' \in T_s$ s.t. $t' \succ t$

– EH: $\forall t' \in T, id_{t'} \in Comps(\Delta)$ s.t. $\gamma(t') = id_{t'}$,
$(t' \in F \Rightarrow \sigma'(id_{t'})(\textit{fired}) = \texttt{true}) \wedge (\sigma'(id_{t'})(\textit{fired}) = \texttt{true} \Rightarrow t' \in F \vee t' \in T_s).$

then $t \in Sens(s'.M - \sum\limits_{t_i \in Pr(t,F)} pre(t_i)) \Leftrightarrow \sigma'(id_t)(\textit{spc}) = \texttt{true}$

Proof.

Given a $t \in T$ and an $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, a $T_s, F, Fset \subseteq T$ and assuming

– $t \in Firable(s')$

– $\nexists t' \in T_s$ s.t. $t' \succ t$

– EH: $\forall t' \in T, id_{t'} \in Comps(\Delta)$ s.t. $\gamma(t') = id_{t'}$,
$(t' \in F \Rightarrow \sigma'(id_{t'})(\texttt{fired}) = \texttt{true}) \wedge (\sigma'(id_{t'})(\texttt{fired}) = \texttt{true} \Rightarrow t' \in F \vee t' \in T_s).$

let us show

$t \in Sens(s'.M - \sum\limits_{t_i \in Pr(t,F)} pre(t_i)) \Leftrightarrow \sigma'(id_t)(\texttt{spc}) = \texttt{true}.$

By construction and by definition of id_t, there exist g_t, i_t, o_t s.t. $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$.
By property of the stabilize relation, $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$, and through the examination of the $\texttt{priority\_authorization\_evaluation}$ process defined in the $\texttt{transition}$ design architecture, we can deduce:

$$\sigma'(id_t)(\texttt{spc}) = \prod_{i=0}^{\Delta(id_t)(\texttt{ian})-1} \sigma'(id_t)(\texttt{pauths})[i]$$

Rewriting the goal with the above equation:

$t \in Sens(s'.M - \sum\limits_{t_i \in Pr(t,F)} pre(t_i)) \Leftrightarrow \prod\limits_{i=0}^{\Delta(id_t)(\texttt{ian})-1} \sigma'(id_t)(\texttt{pauths})[i] = \texttt{true}.$

Then, the proof is in two parts:

1. $t \in Sens(s'.M - \sum\limits_{t_i \in Pr(t,F)} pre(t_i)) \Rightarrow \prod\limits_{i=0}^{\Delta(id_t)(\texttt{ian})-1} \sigma'(id_t)(\texttt{pauths})[i] = \texttt{true}$

2. $\prod\limits_{i=0}^{\Delta(id_t)(\texttt{ian})-1} \sigma'(id_t)(\texttt{pauths})[i] = \texttt{true} \Rightarrow t \in Sens(s'.M - \sum\limits_{t_i \in Pr(t,F)} pre(t_i))$

Let us prove both sides of the equivalence:

1. Assuming that $t \in Sens(s'.M - \sum_{t_i \in Pr(t,F)} pre(t_i))$, let us show

$$\prod_{i=0}^{\Delta(id_t)(\mathtt{ian})-1} \sigma'(id_t)(\mathtt{pauths})[i] = \mathtt{true}.$$

Let us perform case analysis on $input(t)$; there are 2 cases:

- **CASE** $input(t) = \varnothing$:
  By construction, $<\mathtt{input\_arcs\_number} \Rightarrow 1> \in g_t$ and $<\mathtt{priority\_authoriza}$-$\mathtt{tions(0)} \Rightarrow \mathtt{true}> \in i_t$.
  By property of the elaboration relation, we have $\Delta(id_t)(\mathtt{ian}) = 1$, and by property of the stabilize relation, we have $\sigma'(id_t)(\mathtt{pauths})[0] = \mathtt{true}$.
  Rewriting the goal with $\Delta(id_t)(\mathtt{ian}) = 1$ and $\sigma'(id_t)(\mathtt{pauths})[0] = \mathtt{true}$, and simplifying the goal: tautology.

- **CASE** $input(t) \neq \varnothing$:
  Then, let us show an equivalent goal:
  $$\forall i \in [0, \Delta(id_t)(\mathtt{ian}) - 1], \ \sigma'(id_t)(\mathtt{pauths})[i] = \mathtt{true}.$$

  Given an $i \in [0, \Delta(id_t)(\mathtt{ian}) - 1]$, let us show $\sigma'(id_t)(\mathtt{pauths})[i] = \mathtt{true}.$

  By construction, $<\mathtt{input\_arcs\_number} \Rightarrow |input(t)|> \in g_t$.
  By property of the elaboration relation, we have $\Delta(id_t)(\mathtt{ian}) = |input(t)|$. Then, we can deduce $i \in [0, |input(t)| - 1]$.
  By construction, for all $i \in [0, |input(t)| - 1]$, there exist a $p \in input(t)$ and an $id_p \in Comps(\Delta)$ s.t. $\gamma(p) = id_p$, there exist a g_p, i_p, o_p s.t. $\mathtt{comp}(id_p, \mathtt{place}, g_p, i_p, o_p) \in d.cs$, and there exist a $j \in [0, |output(p)|]$ and an $id_{ji} \in Sigs(\Delta)$ s.t. $<\mathtt{input\_arcs\_valid(i)} \Rightarrow \mathtt{id_{ji}}> \in i_t$ and $<\mathtt{output\_arcs\_valid(j)} \Rightarrow \mathtt{id_{ji}}> \in o_t$. Let us take such a $p \in input(t), id_p \in Comps(\Delta), g_p, i_p, o_p, j \in [0, |output(p)|]$ and $id_{ji} \in Sigs(\Delta)$.

  Now, let us perform case analysis on the nature of the arc connecting p and t; there are 2 cases:

  - **CASE** $pre(p,t) = (\omega, \mathtt{test})$ or $pre(p,t) = (\omega, \mathtt{inhib})$:
    By construction, $<\mathtt{priority\_authorizations(i)} \Rightarrow \mathtt{true}> \in i_t$, and by property of the stabilize relation: $\sigma'(id_t)(\mathtt{pauths})[i] = \mathtt{true}.$

  - **CASE** $pre(p,t) = (\omega, \mathtt{basic})$:
    Let us define $output_c(p) = \{t \in T \mid \exists \omega, \ pre(p,t) = (\omega, \mathtt{basic})\}$, the set of output transitions of p that are in conflict. Then, there are two cases, one for each way to solve the conflicts between the output transitions of p:

* **CASE** For all pair of transitions in $output_c(p)$, all conflicts are solved by mutual exclusion:

  By construction, $<\texttt{priority\_authorizations(i)} \Rightarrow \texttt{true}> \in i_t$, and by property of the stabilize relation: $\sigma'(id_t)(\texttt{pauths})[i] = \texttt{true.}$

* **CASE** The priority relation is a strict total order over the set $output_c(p)$:

By construction, there exists an $id'_{ji} \in Sigs(\Delta)$ s.t.
$<\texttt{priority\_authorizations(i)} \Rightarrow id'_{ji}> \in i_t$ and
$<\texttt{priority\_authorizations(j)} \Rightarrow id'_{ji}> \in o_p$.
By property of the stabilize relation, $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$ and
$\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$, we can deduce:

$$\sigma'(id_t)(\texttt{pauths})[i] = \sigma'(id'_{ji}) = \sigma'(id_p)(\texttt{pauths})[j]$$

Rewriting the goal with the above equation: $\boxed{\sigma'(id_p)(\texttt{pauths})[j] = \texttt{true.}}$

By property of the stabilize relation, $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$, and through the examination of the $\texttt{priority\_evaluation}$ process defined in the $\texttt{place}$ design behavior, we can deduce:

$$\sigma'(id_p)(\texttt{pauths})[j] = (\sigma'(id_p)(\texttt{sm}) \geq \texttt{vsots} + \sigma'(id_p)(\texttt{oaw})[j]) \qquad \text{(D.47)}$$

Let us define the $\texttt{vsots}$ term as follows:

$$\texttt{vsots} = \sum_{i=0}^{j-1} \begin{cases} \sigma'(id_p)(\texttt{oaw})[i] \text{ if } \sigma'(id_p)(\texttt{otf})[i]. \\ \qquad\qquad \sigma'(id_p)(\texttt{oat})[i] = \texttt{basic} \\ 0 \text{ } otherwise \end{cases} \qquad \text{(D.48)}$$

Rewriting the goal with (D.47): $\boxed{\sigma'(id_p)(\texttt{sm}) \geq \texttt{vsots} + \sigma'(id_p)(\texttt{oaw})[j]}$

By definition of $t \in Sens(s'.M - \sum\limits_{t_i \in Pr(t,F)} pre(t_i))$, we can deduce:

$s'.M(p) \geq \sum\limits_{t_i \in Pr(t,F)} pre(p, t_i) + \omega.$

Then, there are three points to prove:

(a) $\boxed{s'.M(p) = \sigma'(id_p)(\texttt{sm})}$

(b) $\boxed{\omega = \sigma'(id_p)(\texttt{oaw})[j]}$

(c) $\boxed{\sum\limits_{t_i \in Pr(t,F)} pre(p, t_i) = \texttt{vsots}}$

Let us prove these three points:

(a) $\boxed{s'.M(p) = \sigma'(id_p)(\texttt{sm})}$

Appealing to Lemma 34, $s'.M(p) = \sigma'(id_p)(\texttt{sm}).$

(b) $\boxed{\omega = \sigma'(id_p)(\texttt{oaw})[j]}$

By construction, and as $pre(p,t) = (\omega, \texttt{basic})$, we know that $<\texttt{output\_-}$
$\texttt{arcs\_weights(j)} \Rightarrow \omega> \in i_p$.
By property of the stabilize relation and $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$:
$\boxed{\omega = \sigma'(id_p)(\texttt{oaw})[j].}$

(c) $\boxed{\displaystyle\sum_{t_i \in Pr(t,F)} pre(p,t_i) = \texttt{vsots}}$

Let us replace the left and right term of the equality by their full definition:

$$\sum_{t_i \in Pr(t,F)} \begin{cases} \omega \text{ if } pre(p,t_i) = (\omega, \texttt{basic}) \\ 0 \text{ } otherwise \end{cases}$$
$$=$$
$$\sum_{i=0}^{j-1} \begin{cases} \sigma'(id_p)(\texttt{oaw})[i] \text{ if } \sigma'(id_p)(\texttt{otf})[i]. \\ \qquad\qquad \sigma'(id_p)(\texttt{oat})[i] = \texttt{basic} \\ 0 \text{ } otherwise \end{cases}$$

Now, we must reason on the priority status of transition t regarding the group of conflicting output transitions of p. There 2 cases:

* **CASE** t is the top-priority transition in the group of conflicting output transitions of p:
  In that case, the set $Pr(t,F)$ is empty and, by construction, $j = 0$. Thus, the goal is a tautology $\boxed{0 = 0.}$

* **CASE** t is not the top-priority transition in the group of conflicting output transitions of p:

In that case, we know that there is a least one element in $Pr(t,F)$ and the index $j > 0$.
Let us replace the sum terms in the goal by equivalent terms:

$$\sum_{t_i \in Pr_p} \begin{cases} \omega \text{ if } pre(p,t_i) = (\omega, \texttt{basic}) \text{ and } t_i \in F \\ 0 \text{ } otherwise \end{cases}$$
$$=$$
$$\sum_{i \in IPr_p} \begin{cases} \sigma'(id_p)(\texttt{oaw})[i] \text{ if } \sigma'(id_p)(\texttt{otf})[i] \\ 0 \text{ } otherwise \end{cases}$$

Let us define the set Pr_p as
$$Pr_p = \{t_i \mid t_i \succ t \wedge \exists \omega \text{ s.t. } pre(p,t_i) = (\omega, \texttt{basic})\}$$
and set IPr_p as

$$IPr_p = \{i \mid i \in [0, j-1] \wedge \sigma'(id_p)(\texttt{oat})[i] = \texttt{basic}\}$$

Let us define $f(t_i)$ as

$$f(t_i) = \begin{cases} \omega \text{ if } pre(p, t_i) = (\omega, \texttt{basic}) \text{ and } t_i \in F \\ 0 \text{ otherwise} \end{cases}$$

and $g(i)$ as

$$g(i) = \begin{cases} \sigma'(id_p)(\texttt{oaw})[i] \text{ if } \sigma'(id_p)(\texttt{otf})[i] \\ 0 \text{ otherwise} \end{cases}$$

then, we must prove $\boxed{\sum\limits_{t_i \in Pr_p} f(t_i) = \sum\limits_{i \in IPr_p} g(i).}$

To prove the above equality, it is sufficient to prove that there exists a bijection β from Pr_p to IPr_p such that for all $t_i \in Pr_p$, $f(t_i) = g(\beta(t_i))$. Let us use the function β that takes a $t_i \in Pr_p$ and yields the index denoting the position of t_i in the priority-ordered version of set Pr_p. We assumed that a total order existed over the conflicting output transitions of place p, then there exists a total ordering of the transitions of set Pr_p, i.e. the conflicting output transitions of place p with a higher priority than t. By property of the HILECOP transformation function, we know that the index returned by the function β belongs to the interval $[0, j-1]$ and verifies $\sigma'(id_p)(\texttt{oat})[i] = \texttt{basic}$. Given a $t_i \in Pr_p$, we must show $\boxed{f(t_i) = g(\beta(t_i)).}$

Let us unfold terms $f(t_i)$ and $g(\beta(t_i))$ to their full definition:

$$\begin{cases} \omega \text{ if } pre(p, t_i) = (\omega, \texttt{basic}) \text{ and } t_i \in F \\ 0 \text{ otherwise} \end{cases}$$
$$=$$
$$\begin{cases} \sigma'(id_p)(\texttt{oaw})[\beta(t_i)] \text{ if } \sigma'(id_p)(\texttt{otf})[\beta(t_i)] \\ 0 \text{ otherwise} \end{cases}$$

By construction, there exists an $id_{t_i} \in Comps(\Delta)$ such that $\gamma(t_i) = id_{t_i}$, and there exist g_{t_i}, i_{t_i} and o_{t_i} such that $\texttt{comp}(id_{t_i}, \texttt{transition}, g_{t_i}, i_{t_i}, o_{t_i}) \in d.cs$.

By property of the function β and by construction, we can deduce that the element of index $\beta(t_i)$ of the $\texttt{otf}$ input port of PCI id_p is connected the $\texttt{fired}$ output port of TCI id_{t_i}. Thus, there exists an $id_{\beta i} \in Sigs(\Delta)$ s.t. $<\texttt{otf}(\beta(t_i)) \Rightarrow id_{\beta i}> \in i_p$ and
$<\texttt{fired} \Rightarrow id_{\beta i}> \in o_{t_i}$.

By property of the stabilize relation, $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$ and $\texttt{comp}(id_{t_i}, \texttt{transition}, g_{t_i}, i_{t_i}, o_{t_i}) \in d.cs$, we have

$$\sigma'(id_{t_i})(\texttt{fired}) = \sigma'(id_{\beta i}) = \sigma'(id_p)(\texttt{otf})[\beta(t_i)]$$

then, we can rewrite the goal with the above equation

$$
\begin{cases} \omega \text{ if } pre(p, t_i) = (\omega, \texttt{basic}) \text{ and } t_i \in F \\ 0 \text{ } otherwise \end{cases}
$$
$$
=
$$
$$
\begin{cases} \sigma'(id_p)(\texttt{oaw})[\beta(t_i)] \text{ if } \sigma'(id_{t_i})(\texttt{fired}) \\ 0 \text{ } otherwise \end{cases}
$$

By property of the function β and by construction, we can deduce that the element of index $\beta(t_i)$ of the $\texttt{oaw}$ input port of PCI id_p is connected to a constant value denoting the weight of the arc between place p and transition t_i. Thus, we have

$$
<\texttt{oaw}(\beta(t_i)) \Rightarrow \omega> \in i_p \text{ where } pre(p, t_i) = (\omega, \texttt{basic})
$$

By property of the stabilize relation and $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$, we have

$$
\sigma'(id_p)(\texttt{oaw})[\beta(t_i)] = \omega
$$

then, we can rewrite the goal with the above equation

$$
\begin{cases} \omega \text{ if } pre(p, t_i) = (\omega, \texttt{basic}) \text{ and } t_i \in F \\ 0 \text{ } otherwise \end{cases}
$$
$$
=
$$
$$
\begin{cases} \omega \text{ if } \sigma'(id_{t_i})(\texttt{fired}) \\ 0 \text{ } otherwise \end{cases}
$$

Finally, proving the goal comes down to proving

$$
t_i \in F \Leftrightarrow \sigma'(id_{t_i})(\texttt{fired}) = \texttt{true}
$$

Let us prove both sense of the equivalence:

(a) Assuming $t_i \in F$, let us show $\boxed{\sigma'(id_{t_i})(\texttt{fired}) = \texttt{true}}$.
Appealing to EH, proving the goal is trivial.

(b) Assuming $\sigma'(id_{t_i})(\texttt{fired}) = \texttt{true}$, let us show $\boxed{t_i \in F}$.
Appealing to EH, we have $t_i \in F \vee t_i \in T_s$. There are two cases: either $t_i \in F$ or $t_i \in T$. In the case where $t_i \in T$, we can show a contradiction with the fact that t is a top-priority transition in set T_s. By definition, transition t_i has a higher firing priority than t, and thus, if t_i belongs to set T_s, then t is no longer a top-priority transition of set T_s; whence the contradiction.

2. Assuming that $\prod\limits_{i=0}^{\Delta(id_t)(\texttt{ian})-1} \sigma'(id_t)(\texttt{pauths})[i] = \texttt{true}$, let us show

$$\boxed{t \in Sens(s'.M - \sum_{t_i \in Pr(t,F)} pre(t_i)).}$$

By definition of $t \in Sens(s'.M - \sum\limits_{t_i \in Pr(t,F)} pre(t_i))$:

$$\boxed{\begin{array}{l} \forall p \in P, \omega \in \mathbb{N}^*, \\ ((pre(p,t) \quad = \quad (\omega,\texttt{basic}) \vee pre(p,t) \quad = \quad (\omega,\texttt{test})) \quad \Rightarrow \quad s'.M(p) - \\ \sum\limits_{t_i \in Pr(t,F)} pre(p,t_i) \geq \omega) \\ \wedge \, (pre(p,t) = (\omega,\texttt{inhib}) \Rightarrow s'.M(p) - \sum\limits_{t_i \in Pr(t,F)} pre(p,t_i) < \omega) \end{array}}$$

Given a $p \in P$ and an $\omega \in \mathbb{N}^*$, let us show

$$\boxed{\begin{array}{l} ((pre(p,t) \quad = \quad (\omega,\texttt{basic}) \vee pre(p,t) \quad = \quad (\omega,\texttt{test})) \quad \Rightarrow \quad s'.M(p) - \\ \sum\limits_{t_i \in Pr(t,F)} pre(p,t_i) \geq \omega) \\ \wedge \, (pre(p,t) = (\omega,\texttt{inhib}) \Rightarrow s'.M(p) - \sum\limits_{t_i \in Pr(t,F)} pre(p,t_i) < \omega) \end{array}}$$

By construction, there exists an $id_p \in Comps(\Delta)$ s.t. $\gamma(p) = id_p$. By construction and by definition of id_p, there exist g_p, i_p, o_p s.t. $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$.

To prove the goal, there are different cases:

(a) Assuming that $pre(p,t) = (\omega,\texttt{test})$, let us show
$$\boxed{s'.M(p) - \sum_{t_i \in Pr(t,F)} pre(p,t_i) \geq \omega.}$$

Then, assuming that the priority relation is well-defined, there exists no transition t_i connected by a $\texttt{basic}$ arc to p that verifies $t_i \succ t$. This is because t is connected to p by a $\texttt{test}$ arc; thus, t is not in conflict with the other output transitions of p; thus, there is no relation of priority between t and the other output transitions of p.

Then, we can deduce that $\sum\limits_{t_i \in Pr(t,F)} pre(p,t_i) = 0$.

Then, the new goal is $s'.M(p) \geq \omega$.

Knowing that $t \in Firable(s')$, thus, $t \in Sens(s'.M)$, thus, we have $\boxed{s'.M(p) \geq \omega.}$

(b) Assuming that $pre(p,t) = (\omega,\texttt{inhib})$, let us show
$$\boxed{s'.M(p) - \sum_{t_i \in Pr(t,F)} pre(p,t_i) < \omega.}$$

Use the same strategy as above.

(c) Assuming that $pre(p,t) = (\omega, \texttt{basic})$, let us show

$$\boxed{s'.M(p) - \sum_{t_i \in Pr(t,F)} pre(p,t_i) \geq \omega.}$$

Then, there are two cases:

i. **CASE** For all pair of transitions in $output_c(p)$, all conflicts are solved by mutual exclusion.

Then, assuming that the priority relation is well-defined, it must not be defined over the set $output_c(t)$, and we know that $t \in output_c(p)$ since $pre(p,t) = (\omega, \texttt{basic})$.

Then, there exists no transition t_i connected to p by a $\texttt{basic}$ arc that verifies $t_i \succ t$.

Then, we can deduce $\sum_{t_i \in Pr(t,F)} pre(p,t_i) = 0$.

Then, the new goal is $s'.M(p) \geq \omega$.

We know $t \in Firable(s')$, thus, $t \in Sens(s'.M)$, thus, $\boxed{s'.M(p) \geq \omega.}$

ii. **CASE** The priority relation is a strict total order over the set $output_c(p)$.

By construction, there exists $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$. By construction and by definition of id_t, there exist g_t, i_t, o_t s.t. $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$.

By construction, there exist $j \in [0, |input(t)| - 1]$, $k \in [0, |output(t)| - 1]$, and $id_{kj} \in Sigs(\Delta)$ s.t. $\texttt{<priority\_authorizations(j)} \Rightarrow id_{kj}\texttt{>} \in i_t$ and $\texttt{<priority\_authorizations(k)} \Rightarrow id_{kj}\texttt{>} \in o_p$. Let us take such an j, k and id_{kj}.

From $\prod_{i=0}^{\Delta(id_t)(\texttt{ian})-1} \sigma'(id_t)(\texttt{pauths})[i] = \texttt{true}$, we can deduce that for all $i \in [0, \Delta(id_t) (\texttt{ian}) - 1]$, $\sigma'(id_t)(\texttt{pauths})[i] = \texttt{true}$.

By construction, $\texttt{<input\_arcs\_number} \Rightarrow |input(t)|\texttt{>} \in g_t$, and by property of the elaboration relation, we have $\Delta(id_t)(\texttt{ian}) = |input(t)|$. Then, from $j \in [0, |input(t)| - 1]$, we can deduce $j \in [0, \Delta(id_t)(\texttt{ian}) - 1]$. And, from $\forall i \in [0, \Delta(id_t)(\texttt{ian}) - 1]$, $\sigma'(id_t) (\texttt{pauths})[i] = \texttt{true}$, we can deduce $\sigma'(id_t)(\texttt{pauths})[j] = \texttt{true}$.

By property of the stabilize relation, $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$ and $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \in d.cs$:

$$\sigma'(id_p)(\texttt{pauths})[k] = \sigma'(id_{kj}) = \sigma'(id_t)(\texttt{pauths})[j] = \texttt{true} \qquad (\text{D.49})$$

By property of the stabilize relation and $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \in d.cs$:

$$\sigma'(id_p)(\texttt{pauths})[k] = (\sigma'(id_p)(\texttt{sm}) \geq \texttt{vsots} + \sigma'(id_p)(\texttt{oaw})[k]) \qquad (\text{D.50})$$

Let us define the `vsots` term as follows:

$$\texttt{vsots} = \sum_{i=0}^{k-1} \begin{cases} \sigma'(id_p)(\texttt{oaw})[i] \text{ if } \sigma'(id_p)(\texttt{otf})[i]. \\ \qquad\qquad \sigma'(id_p)(\texttt{oat})[i] = \texttt{basic} \\ 0 \text{ otherwise} \end{cases} \qquad \text{(D.51)}$$

From (D.49) and (D.50), we can deduce that $\sigma'(id_p)(\texttt{sm}) \geq \texttt{vsots} + \sigma'(id_p)(\texttt{oaw})[k]$.

Then, there are three points to prove:

A. $\boxed{s'.M(p) = \sigma'(id_p)(\texttt{sm})}$

B. $\boxed{\omega = \sigma'(id_p)(\texttt{oaw})[k]}$

C. $\boxed{\sum_{t_i \in Pr(t,F)} pre(p,t_i) = \texttt{vsots}}$

See 1 for the remainder of the proof.

$\square$

Lemma 48 (Falling edge equal not fired). *For all sitpn, b, d, γ, Δ, σ_e, E_c, E_p, τ, s, s', σ, $\sigma_\downarrow$, σ' that verify the hypotheses of Definition 45, then $\forall t, id_t$ s.t. $\gamma(t) = id_t$, $t \notin Fired(s') \Leftrightarrow \sigma'(id_t)(fired) = \texttt{false}$.*

Proof.

Proving the above lemma is trivial by appealing to Lemma Falling edge equal fired and by reasoning on contrapositives. $\square$

Bibliography

[1] J. -R. Abrial et al. "The B-Method". In: *VDM '91 Formal Software Development Methods*. Ed. by Søren Prehn and Hans Toetenel. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1991, pp. 398–405. ISBN: 978-3-540-46456-3. DOI: 10.1007/BFb0020001.

[2] Jean-Raymond Abrial. "From Z to B and Then Event-B: Assigning Proofs to Meaningful Programs". In: *International Conference on Integrated Formal Methods*. Springer. 2013, pp. 1–15.

[3] Roberto Amadini et al. "Abstract Interpretation, Symbolic Execution and Constraints". In: *Recent Developments in the Design and Implementation of Programming Languages*. Ed. by Frank S. de Boer and Jacopo Mauro. Vol. 86. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 7:1–7:19. ISBN: 978-3-95977-171-9. DOI: 10.4230/OASIcs.Gabbrielli.7.

[4] David Andreu, David Guiraud, and Guillaume Souquet. "A Distributed Architecture for Activating the Peripheral Nervous System". In: *Journal of Neural Engineering* 6.2 (Apr. 1, 2009), p. 026001. ISSN: 1741-2560, 1741-2552. DOI: 10.1088/1741-2560/6/2/026001.

[5] Peter J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann, Oct. 7, 2010. 933 pp. ISBN: 978-0-08-056885-0.

[6] Johan Bengtsson et al. "Uppaal in 1995". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Tiziana Margaria and Bernhard Steffen. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1996, pp. 431–434. ISBN: 978-3-540-49874-2. DOI: 10.1007/3-540-61042-1_66.

[7] Karima Berramla, El Abbassia Deba, and Mohammed Senouci. "Formal Validation of Model Transformation with Coq Proof Assistant". In: *2015 First International Conference on New Technologies of Information and Communication (NTIC)*. 2015 First International Conference on New Technologies of Information and Communication (NTIC). Nov. 2015, pp. 1–6. DOI: 10.1109/NTIC.2015.7368755.

[8] Yves Bertot and P. Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Berlin ; New York: Springer, 2004. 469 pp. ISBN: 978-3-540-20854-9.

[9] Dines Bjørner and Klaus Havelund. "40 Years of Formal Methods". In: *FM 2014: Formal Methods*. Ed. by Cliff Jones, Pekka Pihlajasaari, and Jun Sun. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 42–61. ISBN: 978-3-319-06410-9. DOI: 10.1007/978-3-319-06410-9_4.

[10] Dines Bjorner et al., eds. *VDM '87. VDM - A Formal Method at Work: VDM-Europe Symposium 1987, Brussels, Belgium, March 23-26, 1987, Proceedings*. Lecture Notes in Computer Science. Berlin Heidelberg: Springer-Verlag, 1987. ISBN: 978-3-540-17654-1. DOI: 10.1007/3-540-17654-3.

[11] David C. Black et al. *SystemC: From the Ground Up, Second Edition*. Springer Science & Business Media, Dec. 18, 2009. 291 pp. ISBN: 978-0-387-69958-5.

[12] B. Blanchet et al. "Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software, Invited Chapter". In: *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil d. Jones*. Ed. by T. Mogensen, D.A. Schmidt, and I.H. Sudborough. LNCS 2566. Springer–Verlag, Oct. 2002, pp. 85–108. ISBN: 3-540-00326-6.

[13] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. "Formal Verification of a C Compiler Front-End". In: *FM 2006: Formal Methods*. International Symposium on Formal Methods. Springer, Berlin, Heidelberg, Aug. 21, 2006, pp. 460–475. DOI: 10.1007/11813040_31.

[14] Richard Bonichon, David Delahaye, and Damien Doligez. "Zenon: An Extensible Automated Theorem Prover Producing Checkable Proofs". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Nachum Dershowitz and Andrei Voronkov. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2007, pp. 151–165. ISBN: 978-3-540-75560-9. DOI: 10.1007/978-3-540-75560-9_13.

[15] Egon Börger, Uwe Glässer, and Wolfgang Muller. "A Formal Definition of an Abstract VHDL'93 Simulator by EA-Machines". In: *Formal Semantics for VHDL*. Ed. by Carlos Delgado Kloos and Peter T. Breuer. Red. by Jonathan Allen. Vol. 307. Boston, MA: Springer US, 1995, pp. 107–139. ISBN: 978-1-4613-5941-8 978-1-4615-2237-9. DOI: 10.1007/978-1-4615-2237-9_5.

[16] D.D. Borrione, L.V. Pierre, and A.M. Salem. "Formal Verification of VHDL Descriptions in the Prevail Environment". In: *IEEE Design Test of Computers* 9.2 (June 1992), pp. 42–56. ISSN: 1558-1918. DOI: 10.1109/54.143145.

[17] Dominique Borrione and Ashraf Salem. "Denotational Semantics of a Synchronous VHDL Subset". In: *Formal Methods in System Design* 7.1-2 (Aug. 1995), pp. 53–71. ISSN: 0925-9856, 1572-8102. DOI: 10.1007/BF01383873.

[18] Thomas Bourgeat et al. "The Essence of Bluespec: A Core Language for Rule-Based Hardware Design". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. New York, NY, USA: Association for Computing Machinery, June 11, 2020, pp. 243–257. ISBN: 978-1-4503-7613-6. DOI: 10.1145/3385412.3385965.

[19] Timothy Bourke et al. "A formally verified compiler for Lustre". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Association for Computing Machinery, June 2017, pp. 586–601. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062358. URL: https://doi.org/10.1145/3062341.3062358.

[20] Jonathan Bowen and Victoria Stavridou. "Safety-Critical Systems, Formal Methods and Standards". In: *Software engineering journal* 8.4 (1993), pp. 189–209.

[21] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. "SELECT: a Formal System for Testing and Debugging Programs by Symbolic Execution". In: *Proceedings of the International Conference on Reliable Software*. New York, NY, USA: Association for Computing Machinery, Apr. 1, 1975, pp. 234–245. ISBN: 978-1-4503-7385-2. DOI: 10.1145/800027.808445.

[22] Thomas Braibant and Adam Chlipala. "Formal Verification of Hardware Synthesis". In: *Computer Aided Verification*. Ed. by Natasha Sharygina and Helmut Veith. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 213–228. ISBN: 978-3-642-39799-8. DOI: 10.1007/978-3-642-39799-8_14.

[23] Peter T. Breuer, Luis Sánchez Fernández, and Carlos Delgado Kloos. "A Functional Semantics for Unit-Delay VHDL". In: *Formal Semantics for VHDL*. Ed. by Carlos Delgado Kloos and Peter T. Breuer. Red. by Jonathan Allen. Vol. 307. Boston, MA: Springer US, 1995, pp. 43–70. ISBN: 978-1-4613-5941-8 978-1-4615-2237-9. DOI: 10.1007/978-1-4615-2237-9_3.

[24] Peter T. Breuer, Luis Sánchez Fernández, and Carlos Delgado Kloos. "A Simple Denotational Semantics, Proof Theory and a Validation Condition Generator for Unit-Delay VHDL". In: *Formal Methods in System Design* 7.1 (Aug. 1, 1995), pp. 27–51. ISSN: 1572-8102. DOI: 10.1007/BF01383872.

[25] Daniel Calegari et al. "A Type-Theoretic Framework for Certified Model Transformations". In: *Formal Methods: Foundations and Applications*. Ed. by Jim Davies, Leila Silva, and Adenilso Simao. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 112–127. ISBN: 978-3-642-19829-8. DOI: 10.1007/978-3-642-19829-8_8.

[26] Judith Carlton and David Crocker. "Escher Verification Studio Perfect Developer and Escher C Verifier". In: *Industrial Use of Formal Methods*. John Wiley & Sons, Ltd, pp. 155–193. ISBN: 978-1-118-56182-9. DOI: 10.1002/9781118561829.ch5.

[27] BA Carré and TJ Jennings. *SPARK: The SPADE Ada Kernel: Version 1.0*. HM Stationery Office, 1988.

[28] Jose R. Celaya, Alan A. Desrochers, and Robert J. Graves. "Modeling and Analysis of Multi-Agent Systems Using Petri Nets". In: *2007 IEEE International Conference on Systems, Man and Cybernetics*. 2007 IEEE International Conference on Systems, Man and Cybernetics. Oct. 2007, pp. 1439–1444. DOI: 10.1109/ICSMC.2007.4413960.

[29] Junjie Chen et al. "A Survey of Compiler Testing". In: *ACM Computing Surveys* 53.1 (Feb. 5, 2020), 4:1–4:36. ISSN: 0360-0300. DOI: 10.1145/3363562.

[30] Adam Chlipala. "A Verified Compiler for an Impure Functional Language". In: *ACM SIGPLAN Notices* 45.1 (Jan. 17, 2010), pp. 93–106. ISSN: 0362-1340. DOI: 10.1145/1707801.1706312.

[31] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, Dec. 6, 2013. 437 pp. ISBN: 978-0-262-02665-9.

[32]   E. M. Clarke, E. A. Emerson, and A. P. Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". In: *ACM Transactions on Programming Languages and Systems* 8.2 (Apr. 1, 1986), pp. 244–263. ISSN: 0164-0925. DOI: 10.1145/5397.5399.

[33]   Alain Colmerauer. "An Introduction to Prolog III". In: *Computational Logic* (1990), pp. 37–79. DOI: 10.1007/978-3-642-76274-1_2.

[34]   Benoît Combemale et al. "Essay on Semantics Definition in MDE. An Instrumented Approach for Model Verification". In: *Journal of Software* 4 (Nov. 1, 2009). DOI: 10.4304/jsw.4.9.943-958.

[35]   Byron Cook, Andreas Podelski, and Andrey Rybalchenko. "Terminator: Beyond Safety". In: *Computer Aided Verification*. Ed. by Thomas Ball and Robert B. Jones. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 415–418. ISBN: 978-3-540-37411-4. DOI: 10.1007/11817963_37.

[36]   Thierry Coquand and Christine Paulin. "Inductively Defined Types". In: *COLOG-88*. Ed. by Per Martin-Löf and Grigori Mints. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1990, pp. 50–66. ISBN: 978-3-540-46963-6. DOI: 10.1007/3-540-52335-9_47.

[37]   Patrick Cousot and Radhia Cousot. *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. Jan. 1, 1977, p. 252. 238 pp. DOI: 10.1145/512950.512973.

[38]   Judy Crow et al. "A Tutorial Introduction to PVS". In: WIFT. 1995.

[39]   Maulik A. Dave. "Compiler Verification: A Bibliography". In: *ACM SIGSOFT Software Engineering Notes* 28.6 (Nov. 1, 2003), p. 2. ISSN: 01635948. DOI: 10.1145/966221.966235.

[40]   René David and Hassane Alla. "Petri Nets for Modeling of Dynamic Systems: A Survey". In: *Automatica* 30.2 (Feb. 1, 1994), pp. 175–202. ISSN: 0005-1098. DOI: 10.1016/0005-1098(94)90024-8.

[41]   Frank Dederichs, Claus Dendorfer, and Rainer Weber. "Focus: A Formal Design Method for Distributed Systems". In: *Parallel Computer Architectures* (1993), pp. 190–202. DOI: 10.1007/978-3-662-21577-7_14.

[42]   David Déharbe and Dominique Borrione. "Semantics of a Verification-Oriented Subset of VHDL". In: *Correct Hardware Design and Verification Methods*. Advanced Research Working Conference on Correct Hardware Design and Verification Methods. Springer, Berlin, Heidelberg, Oct. 2, 1995, pp. 293–310. DOI: 10.1007/3-540-60385-9_18.

[43]   Michel Diaz. *Les Réseaux de Petri: Modèles Fondamentaux*. Hermès science publications, 2001.

[44]   Edsger Wybe Dijkstra et al. *A Discipline of Programming*. Vol. 613924118. prentice-hall Englewood Cliffs, 1976.

[45] Gert Döhmen and Ronald Herrmann. "A Deterministic Finite-State Model for VHDL". In: *Formal Semantics for VHDL*. Ed. by Carlos Delgado Kloos and Peter T. Breuer. The Kluwer International Series in Engineering and Computer Science. Boston, MA: Springer US, 1995, pp. 170–204. ISBN: 978-1-4615-2237-9. DOI: 10.1007/978-1-4615-2237-9_7.

[46] Johannes Dyck, Holger Giese, and Leen Lambers. "Automatic Verification of Behavior Preservation at the Transformation Level for Relational Model Transformation". In: *Software & Systems Modeling* 18.5 (5 Oct. 1, 2019), pp. 2937–2972. ISSN: 1619-1374. DOI: 10.1007/s10270-018-00706-9.

[47] E. Allen Emerson and A. Prasad Sistla. "Deciding Full Branching Time Logic". In: *Information and Control* 61.3 (June 1, 1984), pp. 175–201. ISSN: 0019-9958. DOI: 10.1016/S0019-9958(84)80047-9.

[48] Robert W. Floyd. "Assigning Meanings to Programs". In: *Program Verification: Fundamental Issues in Computer Science*. Ed. by Timothy R. Colburn, James H. Fetzer, and Terry L. Rankin. Studies in Cognitive Systems. Dordrecht: Springer Netherlands, 1993, pp. 65–81. ISBN: 978-94-011-1793-7. DOI: 10.1007/978-94-011-1793-7_4.

[49] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML: The Systems Modeling Language*. Morgan Kaufmann, Oct. 23, 2014. 631 pp. ISBN: 978-0-12-800800-3.

[50] Lukasz Fronc and Franck Pommereau. "Towards a Certified Petri Net Model-Checker". In: *Programming Languages and Systems*. Ed. by Hongseok Yang. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 322–336. ISBN: 978-3-642-25318-8. DOI: 10.1007/978-3-642-25318-8_24.

[51] Max Fuchs and Michael Mendler. "A Functional Semantics for Delta-Delay VHDL Based on Focus". In: *Formal Semantics for VHDL*. Ed. by Carlos Delgado Kloos and Peter T. Breuer. Red. by Jonathan Allen. Vol. 307. Boston, MA: Springer US, 1995, pp. 9–42. ISBN: 978-1-4613-5941-8 978-1-4615-2237-9. DOI: 10.1007/978-1-4615-2237-9_2.

[52] Hubert Garavel et al. "CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes". In: *International Journal on Software Tools for Technology Transfer* 15.2 (Apr. 1, 2013), pp. 89–107. ISSN: 1433-2787. DOI: 10.1007/s10009-012-0244-z.

[53] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed Automated Random Testing". In: *ACM SIGPLAN Notices* 40.6 (June 12, 2005), pp. 213–223. ISSN: 0362-1340. DOI: 10.1145/1064978.1065036.

[54] K. G. W. Goossens. "Reasoning about VHDL Using Operational and Observational Semantics". In: *Correct Hardware Design and Verification Methods*. Advanced Research Working Conference on Correct Hardware Design and Verification Methods. Springer, Berlin, Heidelberg, Oct. 2, 1995, pp. 311–327. DOI: 10.1007/3-540-60385-9_19.

[55] David Guiraud et al. "An Implantable Neuroprosthesis for Standing and Walking in Paraplegia: 5-Year Patient Follow-Up". In: *Journal of Neural Engineering* 3.4 (Sept. 2006), pp. 268–275. ISSN: 1741-2552. DOI: 10.1088/1741-2560/3/4/003.

[56] A. Habibi and S. Tahar. "Design and Verification of SystemC Transaction-Level Models". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 14.1 (Jan. 2006), pp. 57–68. ISSN: 1557-9999. DOI: 10.1109/TVLSI.2005.863187.

[57] Reiner Hähnle and Marieke Huisman. "Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools". In: *Computing and Software Science: State of the Art and Perspectives*. Ed. by Bernhard Steffen and Gerhard Woeginger. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 345–373. ISBN: 978-3-319-91908-9. DOI: 10.1007/978-3-319-91908-9_18.

[58] Michael R. Hansen, Jan Madsen, and Aske Wiid Brekling. "Semantics and Verification of a Language for Modelling Hardware Architectures". In: *Formal Methods and Hybrid Real-Time Systems* (2007), pp. 300–319. DOI: 10.1007/978-3-540-75221-9_13.

[59] Thomas A. Henzinger et al. "Lazy Abstraction". In: *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '02. New York, NY, USA: Association for Computing Machinery, Jan. 1, 2002, pp. 58–70. ISBN: 978-1-58113-450-6. DOI: 10.1145/503272.503279.

[60] Yann Herklotz et al. "Formal verification of high-level synthesis". In: *Proceedings of the ACM on Programming Languages* 5.OOPSLA (Oct. 2021), 117:1–117:30. DOI: 10.1145/3485494.

[61] C. A. R. Hoare. "Communicating Sequential Processes". In: *Communications of the ACM* 21.8 (Aug. 1, 1978), pp. 666–677. ISSN: 0001-0782. DOI: 10.1145/359576.359585.

[62] William A Howard. "The Formulae-as-Types Notion of Construction". In: *To HB Curry: essays on combinatory logic, lambda calculus and formalism* 44 (1980), pp. 479–490.

[63] Graham Hutton. "Introduction to HOL: A Theorem Proving Environment for Higher Order Logic by Mike Gordon and Tom Melham (Eds.), Cambridge University Press, 1993, ISBN 0-521-44189-7". In: *Journal of Functional Programming* 4.4 (Oct. 1994), pp. 557–559. ISSN: 1469-7653, 0956-7968. DOI: 10.1017/S0956796800001180.

[64] IEEE Computer Society et al. *IEEE Standard VHDL Language Reference Manual*. New York, N.Y.: Institute of Electrical and Electronics Engineers, 2000. ISBN: 978-0-7381-1948-9 978-0-7381-1949-6.

[65] "IEEE Standard Hardware Description Language Based on the Verilog(R) Hardware Description Language". In: *IEEE Std 1364-1995* (Oct. 1996), pp. 1–688. DOI: 10.1109/IEEESTD.1996.81542.

[66] *IEEE/IEC 62142-2005 - IEC/IEEE International Standard - Verilog(R) Register Transfer Level Synthesis*.

[67] Mark P. Jones. *The Implementation of the Gofer Functional Programming System*. Yale University, Department of Computer Science, May 1994, p. 52.

[68] Florent Kirchner et al. "Frama-c: A Software Analysis Perspective". In: *Formal Aspects of Computing* 27.3 (2015), pp. 573–609.

[69] Carlos Delgado Kloos and P. Breuer. *Formal Semantics for VHDL*. Springer Science & Business Media, Dec. 6, 2012. 263 pp. ISBN: 978-1-4615-2237-9.

[70] Leslie Lamport. "The Temporal Logic of Actions". In: *ACM Transactions on Programming Languages and Systems* 16.3 (May 1, 1994), pp. 872–923. ISSN: 0164-0925. DOI: 10.1145/177492.177726.

[71] Hélène Leroux. "Méthodologie de conception d'architectures numériques complexes : du formalisme à l'implémentation en passant par l'analyse, préservation de la conformité. Application aux neuroprothèses". PhD thesis. Université Montpellier II - Sciences et Techniques du Languedoc, Oct. 28, 2014.

[72] Xavier Leroy. "A Formally Verified Compiler Back-End". In: *Journal of Automated Reasoning* 43.4 (Nov. 4, 2009), p. 363. ISSN: 1573-0670. DOI: 10.1007/s10817-009-9155-4.

[73] Xavier Leroy. "Formal Certification of a Compiler Back-End or: Programming a Compiler with a Proof Assistant". In: *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. Ed. by J. Gregory Morrisett and Simon L. Peyton Jones. ACM, 2006, pp. 42–54. DOI: 10.1145/1111037.1111042.

[74] Roger Lipsett, Erich Marschner, and Moe Shahdad. "VHDL - The Language". In: *IEEE Design Test of Computers* 3.2 (Apr. 1986), pp. 28–41. ISSN: 1558-1918. DOI: 10.1109/MDT.1986.294900.

[75] David Long and Zane Scott. *A Primer for Model-Based Systems Engineering*. Lulu.com, 2011. 126 pp. ISBN: 978-1-105-58810-5.

[76] Andreas Lööw. "Lutsig: A Verified Verilog Compiler for Verified Circuit Development". In: *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2021. New York, NY, USA: Association for Computing Machinery, Jan. 17, 2021, pp. 46–60. ISBN: 978-1-4503-8299-1. DOI: 10.1145/3437992.3439916.

[77] Said Meghzili et al. "On the Verification of UML State Machine Diagrams to Colored Petri Nets Transformation Using Isabelle/HOL". In: *2017 IEEE International Conference on Information Reuse and Integration (IRI)*. 2017 IEEE International Conference on Information Reuse and Integration (IRI). Aug. 2017, pp. 419–426. DOI: 10.1109/IRI.2017.63.

[78] Ibrahim Merzoug. "Validation formelle des systèmes numériques critiques : génération de l'espace d'états de réseaux de Petri exécutés en synchrone". PhD thesis. Université Montpellier, Jan. 15, 2018.

[79] R. Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science. Berlin Heidelberg: Springer-Verlag, 1980. ISBN: 978-3-540-10235-9. DOI: 10.1007/3-540-10235-3.

[80] Gordon E. Moore. "Cramming More Components onto Integrated Circuits, Reprinted from Electronics, Volume 38, Number 8, April 19, 1965, Pp.114 Ff." In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (Sept. 2006), pp. 33–35. ISSN: 1098-4232. DOI: 10.1109/N-SSC.2006.4785860.

[81] Joan Moschovakis. "Intuitionistic Logic". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Winter 2018. Metaphysics Research Lab, Stanford University, 2018.

[82] Yiannis Moschovakis. *Notes on Set Theory*. Undergraduate Texts in Mathematics. New York: Springer-Verlag, 1994. ISBN: 978-1-4757-4153-7. DOI: 10.1007/978-1-4757-4153-7.

[83] Ben Moszkowski. "Executing Temporal Logic Programs". In: *Seminar on Concurrency*. Ed. by Stephen D. Brookes, Andrew William Roscoe, and Glynn Winskel. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1985, pp. 111–130. ISBN: 978-3-540-39593-5. DOI: 10.1007/3-540-15670-4_6.

[84] T. Murata. "Petri Nets: Properties, Analysis and Applications". In: *Proceedings of the IEEE* 77.4 (Apr. 1989), pp. 541–580. ISSN: 1558-2256. DOI: 10.1109/5.24143.

[85] Peter G. Neumann. "Illustrative Risks to the Public in the Use of Computer Systems and Related Technology". In: *ACM SIGSOFT Software Engineering Notes* 19.1 (Jan. 1, 1994), pp. 16–29. ISSN: 0163-5948. DOI: 10.1145/181610.181612.

[86] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science. Berlin Heidelberg: Springer-Verlag, 2002. ISBN: 978-3-540-43376-7. DOI: 10.1007/3-540-45949-9.

[87] Serafín Olcoz. "A Formal Model of VHDL Using Coloured Petri Nets". In: *Formal Semantics for VHDL*. Ed. by Carlos Delgado Kloos and Peter T. Breuer. The Kluwer International Series in Engineering and Computer Science. Boston, MA: Springer US, 1995, pp. 140–169. ISBN: 978-1-4615-2237-9. DOI: 10.1007/978-1-4615-2237-9_6.

[88] S. Owre et al. "A Tutorial on Using PVS for Hardware Verification". In: *Theorem Provers in Circuit Design*. International Conference on Theorem Provers in Circuit Design. Springer, Berlin, Heidelberg, Sept. 26, 1994, pp. 258–279. DOI: 10.1007/3-540-59047-1_53.

[89] S.L. Pandey, K. Umamageswaran, and P.A. Wilsey. "VHDL Semantics and Validating Transformations". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 18.7 (July 1999), pp. 936–955. ISSN: 1937-4151. DOI: 10.1109/43.771177.

[90] Marco Patrignani, Amal Ahmed, and Dave Clarke. "Formal Approaches to Secure Compilation: A Survey of Fully Abstract Compilation and Related Work". In: *ACM Computing Surveys* 51.6 (Feb. 4, 2019), 125:1–125:36. ISSN: 0360-0300. DOI: 10.1145/3280984.

[91] Christine Paulin-Mohring. "Introduction to the Coq Proof-Assistant for Practical Software Verification". In: *Tools for Practical Software Verification*. Ed. by Bertrand Meyer and Martin Nordio. Vol. 7682. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 45–95. ISBN: 978-3-642-35745-9 978-3-642-35746-6. DOI: 10.1007/978-3-642-35746-6_3.

[92] Volnei A. Pedroni. *Circuit Design with VHDL, Third Edition*. MIT Press, Apr. 14, 2020. 609 pp. ISBN: 978-0-262-35392-2.

[93] Carl Adam Petri. "Kommunikation mit Automaten". PhD thesis. Technical University Darmstadt, Germany, 1962.

[94] Gordon Plotkin. "A Structural Approach to Operational Semantics". In: *J. Log. Algebr. Program.* 60–61 (July 1, 2004), pp. 17–139. DOI: 10.1016/j.jlap.2004.05.001.

[95] Amir Pnueli. "The Temporal Logic of Programs". In: *18th Annual Symposium on Foundations of Computer Science (Sfcs 1977)*. 18th Annual Symposium on Foundations of Computer Science (Sfcs 1977). Oct. 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[96] J. P. Queille and J. Sifakis. "Specification and Verification of Concurrent Systems in CESAR". In: *International Symposium on Programming*. Ed. by Mariangiola Dezani-Ciancaglini and Ugo Montanari. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1982, pp. 337–351. ISBN: 978-3-540-39184-5. DOI: 10.1007/3-540-11494-7_22.

[97] Ralf Reetz and Thomas Kropf. "A Flow Graph Semantics of VHDL: A Basis for Hardware Verification with VHDL". In: *Formal Semantics for VHDL*. Ed. by Carlos Delgado Kloos and Peter T. Breuer. The Kluwer International Series in Engineering and Computer Science. Boston, MA: Springer US, 1995, pp. 205–238. ISBN: 978-1-4615-2237-9. DOI: 10.1007/978-1-4615-2237-9_8.

[98] Alexandre Riazanov and Andrei Voronkov. "Vampire 1.1 (System Description)". In: *Proceedings of the First International Joint Conference on Automated Reasoning*. IJCAR '01. Berlin, Heidelberg: Springer-Verlag, June 18, 2001, pp. 376–380. ISBN: 978-3-540-42254-9.

[99] K. Rustan, M. Leino, and Greg Nelson. "An Extended Static Checker for Modula-3". In: *Compiler Construction*. Ed. by Kai Koskimies. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1998, pp. 302–305. ISBN: 978-3-540-69724-4. DOI: 10.1007/BFb0026441.

[100] Stephan Schulz. "E - A Brainiac Theorem Prover". In: *AI Communications* 15 (Sept. 8, 2002).

[101] Koushik Sen and Gul Agha. "CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools". In: *Computer Aided Verification*. Ed. by Thomas Ball and Robert B. Jones. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 419–423. ISBN: 978-3-540-37411-4. DOI: 10.1007/11817963_38.

[102] Stewart Shapiro and Teresa Kouri Kissel. "Classical logic". In: *The Stanford encyclopedia of philosophy*. Ed. by Edward N. Zalta. Spring 2021. Citation Key: ClassicalLogic. Metaphysics Research Lab, Stanford University, 2021. URL: https://plato.stanford.edu/archives/spr2021/entries/logic-classical/.

[103] Martin Strecker. "Formal Verification of a Java Compiler in Isabelle". In: *Automated Deduction—CADE-18*. International Conference on Automated Deduction. Springer, Berlin, Heidelberg, July 27, 2002, pp. 63–77. DOI: 10.1007/3-540-45620-1_5.

[104] Yong Kiam Tan et al. "A New Verified Compiler Backend for CakeML". In: (Sept. 4, 2016). DOI: 10.17863/CAM.6525.

[105] Yong Kiam Tan et al. "A New Verified Compiler Backend for CakeML". In: (), p. 14.

[106] The Coq Development Team. *Coq, Version 8.13.2*. manual. July 2021.

[107] Krishnaprasad Thirunarayan and Robert L. Ewing. "Structural Operational Semantics for a Portable Subset of Behavioral VHDL-93". In: *Formal Methods in System Design* 18.1 (Jan. 1, 2001), pp. 69–88. ISSN: 1572-8102. DOI: 10.1023/A:1008786720393.

[108]   Dmitry Tsarkov and Ian Horrocks. "FaCT++ Description Logic Reasoner: System Description". In: *Automated Reasoning*. Ed. by Ulrich Furbach and Natarajan Shankar. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 292–297. ISBN: 978-3-540-37188-5. DOI: 10.1007/11814771_26.

[109]   John P. Van Tassel. "An Operational Semantics for a Subset of VHDL". In: *Formal Semantics for VHDL*. Ed. by Carlos Delgado Kloos and Peter T. Breuer. Red. by Jonathan Allen. Vol. 307. Boston, MA: Springer US, 1995, pp. 71–106. ISBN: 978-1-4613-5941-8 978-1-4615-2237-9. DOI: 10.1007/978-1-4615-2237-9_4.

[110]   Philip Wadler. "The Essence of Functional Programming". In: *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '92. New York, NY, USA: Association for Computing Machinery, Feb. 1, 1992, pp. 1–14. ISBN: 978-0-89791-453-6. DOI: 10.1145/143165.143169.

[111]   Freek Wiedijk. "The De Bruijn Factor". In: (Aug. 12, 2000).

[112]   Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT press, 1993.

[113]   Alex Yakovlev and Albert Koelmans. "Petri Nets and Digital Hardware Design". In: *Lecture Notes in Computer Science - LNCS*. Apr. 11, 2006, pp. 154–236. DOI: 10.1007/3-540-65307-4_49.

[114]   Zhibin Yang et al. "From AADL to Timed Abstract State Machines: A Verified Model Transformation". In: *Journal of Systems and Software* 93 (July 1, 2014), pp. 42–68. ISSN: 0164-1212. DOI: 10.1016/j.jss.2014.02.058.

[115]   Zhibin Yang et al. "Towards a Verified Compiler Prototype for the Synchronous Language SIGNAL". In: *Frontiers of Computer Science* 10.1 (Feb. 1, 2016), pp. 37–53. ISSN: 2095-2236. DOI: 10.1007/s11704-015-4364-y.

[116]   Yana Yankova et al. "Automated HDL Generation: Comparative Evaluation". In: *2007 IEEE International Symposium on Circuits and Systems*. 2007 IEEE International Symposium on Circuits and Systems. May 2007, pp. 2750–2753. DOI: 10.1109/ISCAS.2007.378622.