THÈSE POUR OBTENIR LE GRADE DE DOCTEUR DE L'UNIVERSITÉ DE MONTPELLIER

En Informatique

École doctorale : Information, Structures, Systèmes

Unité de recherche LIRMM

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

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

UNIVERSITÉ DE MONTPELLIER

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. A 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 [15]). 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'execution 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 [15]). 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 informal *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 [10], des variantes du langage C [27], ou encore les réseaux de Petri (RdPs) [24], 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 oeuvre 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.

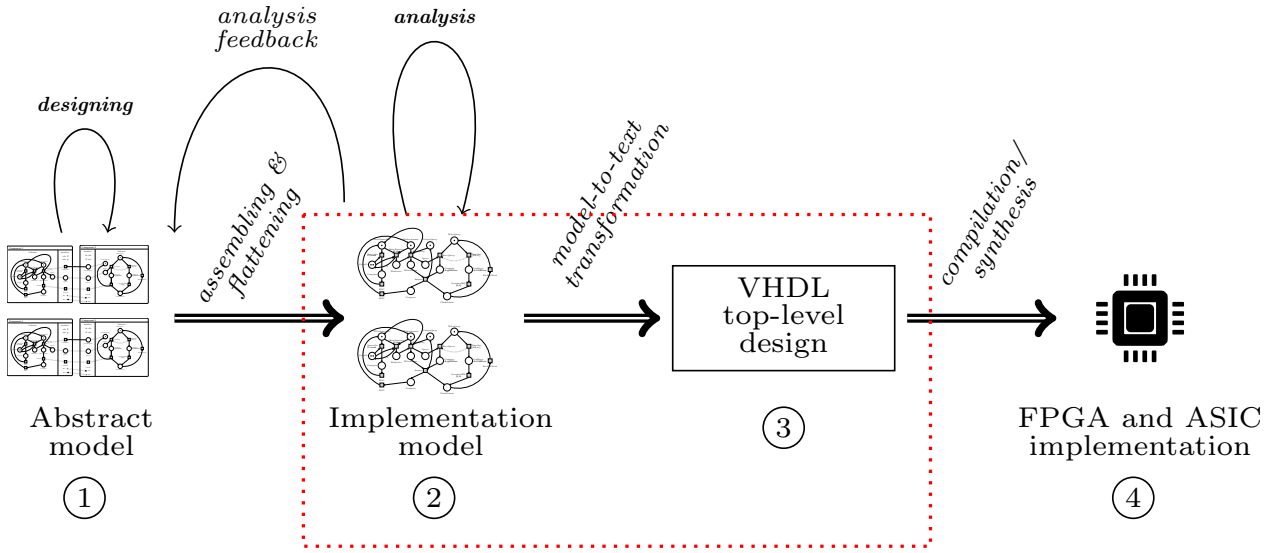

FIGURE 1: Principe de la méthodologie HILECOP; les double flêches horizontales représentent des phases de transformation; les simple flêches indiquent les autres types d'opérations ayant cours à une étape précise, ou entre étapes.

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 [21]. 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 sûr 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.

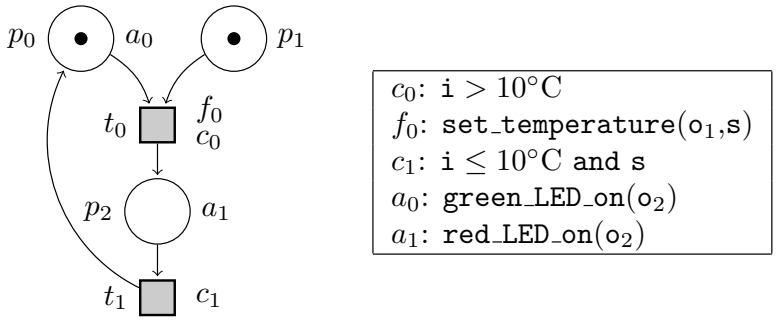

FIGURE 2: Un exemple de réseau de Petri interprété; sur le côté gauche, le RdP; sur le côté droit, les expressions Booléennes associées aux conditions et les opérations associées aux actions et fonctions.

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.

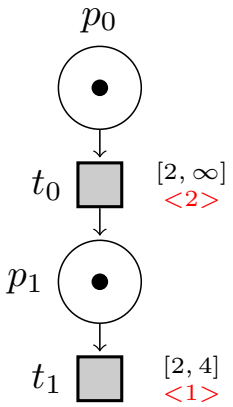

FIGURE 3: Un exemple de RdP temporel. La valeur des compteurs de temps apparaît en rouge.

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.

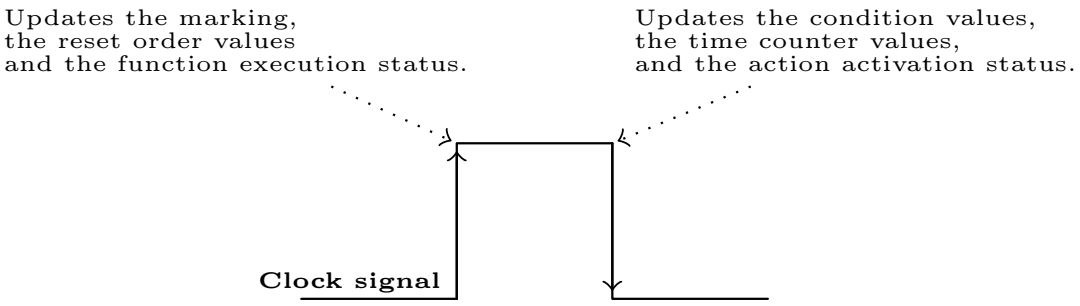

FIGURE 4: Evolution d'un SITPN synchronisée avec un signal d'horloge.

La structure et la sémantique des SITPNs ont été formalisées dans [14, 18]. 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, a lliste 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 utilisées une autre version de l'implémentation 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 [13]. 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  σ'_e ← RunAllOnce(Δ,σ_e,cs)
2  σ ← Stabilize(Δ,σ'_e,cs)
   // Main loop.
3  θ ← [σ]
4  while T_c > 0 do
5      σ_i ← Inject(Δ,σ,E_p,T_c)
6      σ_↑ ← RisingEdge(Δ,σ_i,cs)
7      σ' ← Stabilize(Δ,σ_↑,cs)
8      σ_↓ ← FallingEdge(Δ,σ',cs)
9      σ ← Stabilize(Δ,σ_↓,cs)
10     θ ← θ +++ [σ',σ]
11     T_c ← 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. La 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. La paramètre E_p est l'environnement de simulation. Il permet l'injection de nouvelle valeur sur les ports d'entrée du design à chaque nouveau cycle d'horloge. La paramètre T_c correspond au nombre de cycles de simulation à effectuer, i.e. 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 ($\texttt{RunAllOnce}(\Delta, \sigma_e, cs)$). S'en suit une phase de stabilisation des signaux ($\texttt{Stabilize}(\Delta, \sigma_e, cs)$) 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, i.e. 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 [22, 3]. 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 générique (ou spécifique). C'est un formalisme graphique, i.e. celui des RdPs. La structure des modèles d'entrée est alors bien différente de celle de l'arbre syntaxique des langages de programmation. Par conséquent, l'expression de la transformation ne peut pas suivre la définition récursive suivant l'arbre syntaxique, définition qui est usuelle pour les compilateurs de langage de programmation. Deuxièmement, le langage cible de la transformation HILECOP est une langage de description d'architecture de circuits, i.e. VHDL. Même spécifique, ce langage reste un langage de programmation. Dans le cas des compilateurs pour langage de programmation, chaque instruction du langage source est traduite en une ou plusieurs instructions du langage cible ayant la même sémantique. Dans le cas de la transformation HILECOP, c'est mise en relation directe entre instructions "source" et instructions "cible" n'est pas effective. Comme dit précédemment, il n'existe pas de notion d'instruction dans un RdP. Même en considérant les éléments qui composent un RdP comme des entités atomiques qui pourraient se rapprocher d'instructions, par exemple, les places, les transitions, les actions, les intervalles de temps, etc; même en les considérant ainsi, la transformation HILECOP ne traduit pas un élément atomique d'un RdP en une ou plusieurs instructions du langage cible. Selon l'élément atomique traduit, le code généré est complexe car il rend à la fois compte de la structure du modèle d'entrée (i.e. comment l'élément atomique est connecté aux autres éléments du modèle d'entrée) et de sa sémantique (qui est aussi liée à la structure du modèle).

Nous allons illustrer la transformation modèle-vers-texte HILECOP en prenant le SITPN présenté en Figure 5 comme modèle d'entrée.

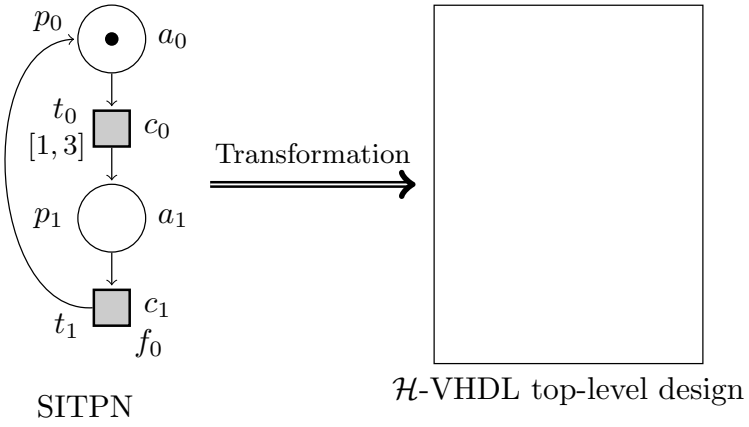

FIGURE 5: Transformation d'un modèle d'entrée SITPN en un design de top-niveau $\mathcal{H}$-VHDL. Le modèle d'entrée est composé de deux places p_0 et p_1, deux transitions t_0 et t_1. La transition t_0 est associée à l'intervalle de temps $[1,3]$ et à la condition c_0. La transition t_1 est associée 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*, i.e. 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.

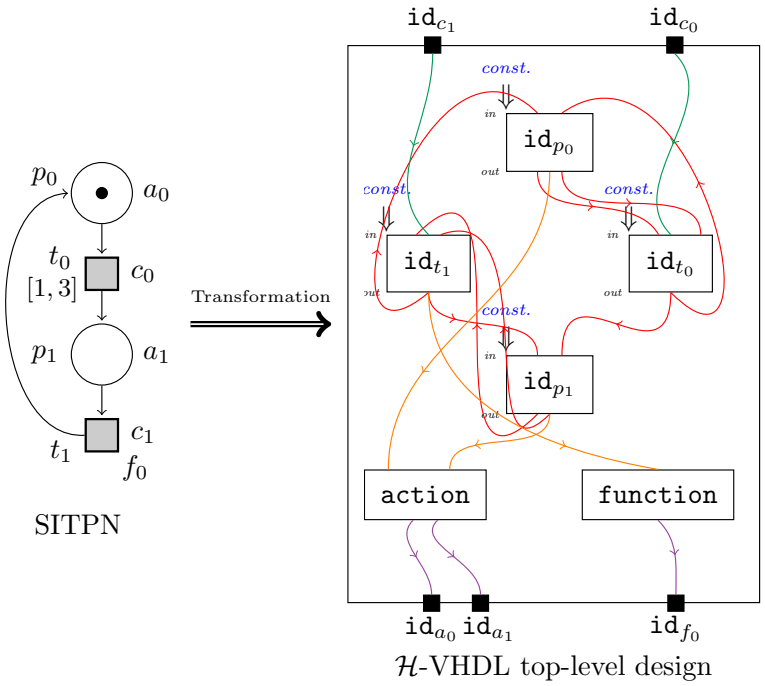

FIGURE 6: Design $\mathcal{H}$-VHDL de top-niveau résultant de la transformation HILECOP.

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 $\texttt{id}_{p_0}$, la transition t_0 au composant `transition` d'identifiant $\texttt{id}_{t_0}$, 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 (i.e. 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 composants que la même sémantique d'exécution du SITPN d'entrée sera obtenue dans sa version VHDL. Du moins, sans considérer les éléments d'interprétation, i.e. les conditions, actions et fonctions.

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* (i.e. 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 en 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 impérative. Ensuite, l'algorithme a été implémenté 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 présentent 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.

0.6 Conclusion

Le but de la thèse est 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. La finalité de ce travail sera la spécification et la démonstration d'un théorème de préservation de comportement pour cette phase de transformation.

Jusqu'ici, la sémantique des SITPNs, modèles de haut niveau de HILECOP, et la sémantique de $\mathcal{H}$-VHDL ont été implantées à l'aide du langage Coq. Concernant les SITPNs, 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).

La transformation d'un SITPN en un modèle VHDL est en cours de programmation avec le langage Coq. Enfin, la dernière étape de ce travail sera d'établir la preuve de préservation de comportement.

Chapter 1

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 **??**. The structure of this chapter is as follows: in Section 1.1, we present our review of the literature related to the proof of semantic preservation theorems for transformation functions; in Section 1.2, we detail our state similarity relation, i.e. the semantic relation between an SITPN and its $\mathcal{H}$-VHDL translation; in Section 1.3, we draw out our behavior preservation theorem; in Section 1.4, 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 1.5, we present some points of the mechanization of the proof with the Coq proof assistant.

1.1 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 [15], 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 1.1, excerpt from [15], shows the different kinds of simulation diagrams possibly relating two programs together.

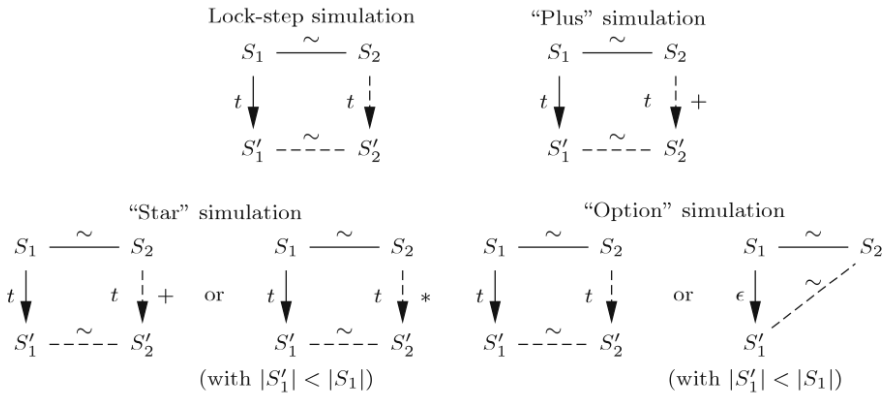

FIGURE 1.1: Simulation diagrams relating the execution of a source program to the execution of a target program; S_1 and S_2 are the initial states of the source and the target program, and S'_1 and S'_2 are the final states of the source and target program, i.e. the states resulting of the execution 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 1.

Definition 1 (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 representation 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?

1.1.1 Compilers for generic programming languages

Taking the CompCert compiler as an example, the compilation pass from Clight programs to Cminor programs is described in [2, 15]. Clight is a subset of the C language, and Cminor is a low-level imperative language. The two languages are endowed with a big-step operational semantics. Here, the execution state of the source and target languages are memory models. 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 [15, 19], or compilers for functional languages [8, 20], 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 [15]). As a consequence, the complexity of the memory model comparison relation increases.

1.1.2 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 [4, 6], 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 [4], 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 1, 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.

In [5], 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

[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.

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 [16], 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 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 [26], 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 [12], 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.

1.1.3 Model transformations

Regarding model transformations, a lot of articles consider semantic preservation as the preservation of structural properties in the transformed model [1, 7, 17].

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 1.1).

In [9] and [25], the authors are interested in giving a translational semantics to a given model having itself a reference execution semantics. In [9], 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 1.2.

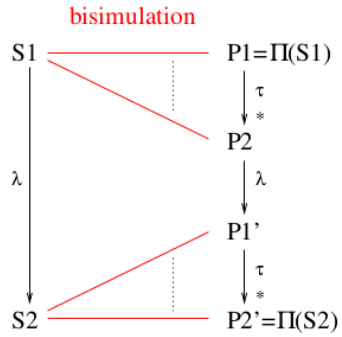

FIGURE 1.2: Bisimulation diagram relating an xSpem model execution and a Petri net execution

In Figure 1.2, one 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 [25], 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 [11], 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.

1.1.4 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 1. 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 are 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.

1.2 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 **??**). 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 **??**) 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 2 (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 1.4), the state similarity relation of Definition 1.2 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 3 (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. $\forall p \in P, id_p \in Comps(\Delta)\ s.t.\ \gamma(p) = id_p,\ s.M(p) = \sigma(id_p)(\texttt{s\_marking})$.
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. $\forall f \in \mathcal{F}, id_f \in Outs(\Delta)\ s.t.\ \gamma(f) = id_f,\ s.ex(f) = \sigma(id_f)$.

Definition 3 is similar to Definition 2 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 4 (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 3) 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. $\forall t \in T, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t, t \notin Sens(s.M) \Leftrightarrow \sigma(id_t)(\texttt{s\_enabled}) = \texttt{false}$.
3. $\forall t \in T, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t,$
   $\sigma(id_t)(\texttt{s\_condition\_combination}) = \prod_{c \in conds(t)} \begin{cases} E_c(\tau, c) & if\ \mathbb{C}(t,c) = 1 \\ \texttt{not}(E_c(\tau,c)) & if\ \mathbb{C}(t,c) = -1 \end{cases}$
   $where\ conds(t) = \{c \in \mathcal{C} \mid \mathbb{C}(t,c) = 1 \vee \mathbb{C}(t,c) = -1\}$.
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 4 extends Definition 3 with the correspondence of the sensitization of transitions and the value of signal `s_enabled`, and the computation of the boolean product of condition values and the value of signal `s_condition_combination`. The last item of Definition 4 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 4 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 5 (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. $\forall p \in P, id_p \in Comps(\Delta)\ s.t.\ \gamma(p) = id_p,\ s.M(p) = \sigma(id_p)(\texttt{s\_marking})$.
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. $\forall c \in \mathcal{C}, id_c \in Ins(\Delta)\ s.t.\ \gamma(c) = id_c,\ s.cond(c) = \sigma(id_c)$.
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 5 is similar to Definition 2 except for the equality between reset orders and the value of the `s_reinit_time_counter` signals. The extended version of the

post falling edge state similarity relation is defined as follows:

Definition 6 (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 5) and*

1. $\forall t \in T, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t, t \in Firable(s) \Leftrightarrow \sigma(id_t)(\texttt{s\_firable}) = \texttt{true}$.
2. $\forall t \in T, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t, t \notin Firable(s) \Leftrightarrow \sigma(id_t)(\texttt{s\_firable}) = \texttt{false}$.
3. $\forall t \in T, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t, t \in Fired(s) \Leftrightarrow \sigma(id_t)(\texttt{fired}) = \texttt{true}$.
4. $\forall t \in T, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t, t \notin Fired(s) \Leftrightarrow \sigma(id_t)(\texttt{fired}) = \texttt{false}$.
5. $\forall p \in P, id_p \in Comps(\Delta)\ s.t.\ \gamma(p) = id_p, \sum_{t \in Fired(s)} pre(p,t) = \sigma(id_p)(\texttt{s\_output\_token\_sum})$.
6. $\forall p \in P, id_p \in Comps(\Delta)\ s.t.\ \gamma(p) = id_p, \sum_{t \in Fired(s)} post(t,p) = \sigma(id_p)(\texttt{s\_input\_token\_sum})$.

Definition 6 extends Definition 5 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`.

1.3 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 **??**, a graph of the dependencies between our high-level theorems and the other theorems and lemmas to which they appeal in their proof.

1.3.1 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): $\texttt{true} = \texttt{true}$

– 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 **??** 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 $\gamma(c) = id_c$.

1.3.2 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 7 states that the execution environment for a given SITPN and the simulation environment for a given $\mathcal{H}$-VHDL design are similar.

Definition 7 (Similar environments)**.** *For a given $sitpn \in SITPN$, a $\mathcal{H}$-VHDL design $d \in design$, a design store $\mathcal{D} \in entity\text{-}id \nrightarrow design$, an elaborated version $\Delta \in ElDesign$ of design d, and a binder $\gamma \in WM(sitpn, d)$, the environment $E_p \in \mathbb{N} \to Ins(\Delta) \to value$, 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 $\gamma \vdash E_p \overset{env}{=} E_c$, 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 8 allows us to compare such traces.

Definition 8 (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 $\theta_s \in \texttt{list}(S(sitpn))$ and the simulation trace $\theta_\sigma \in \texttt{list}(\Sigma)$ are similar if $\gamma \vdash \theta_s \overset{clk}{\sim} \theta_\sigma$ (where $clk \in \{\uparrow, \downarrow\}$) is derivable according to the following rules:*

$$\frac{}{\gamma \vdash [\,] \overset{clk}{\sim} [\,]}\ \textsc{SimTraceNil}\ clk \in \{\uparrow,\downarrow\} \qquad \frac{\gamma \vdash s \overset{\uparrow}{\sim} \sigma \quad \gamma \vdash \theta_s \overset{\downarrow}{\sim} \theta_\sigma}{\gamma \vdash (s :: \theta_s) \overset{\uparrow}{\sim} (\sigma :: \theta_\sigma)}\ \textsc{SimTrace}\uparrow \qquad \frac{\gamma \vdash s \overset{\downarrow}{\sim} \sigma \quad \gamma \vdash \theta_s \overset{\uparrow}{\sim} \theta_\sigma}{\gamma \vdash (s :: \theta_s) \overset{\downarrow}{\sim} (\sigma :: \theta_\sigma)}\ \textsc{SimTrace}\downarrow$$

In Definition 8, 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 9.

Definition 9 (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 $\theta_s \in \texttt{list}(S(sitpn))$ and the simulation trace $\theta_\sigma \in \texttt{list}(\Sigma)$ are fully similar, written $\gamma \vdash \theta_s \sim \theta_\sigma$, according to the following rules:*

$$\frac{}{\gamma \vdash [\,] \sim [\,]}\ \textsc{FullSimTraceNil} \qquad \frac{\gamma \vdash s \sim \sigma \quad \gamma \vdash \theta_s \overset{\uparrow}{\sim} \theta_\sigma}{\gamma \vdash (s :: \theta_s) \sim (\sigma :: \theta_\sigma)}\ \textsc{FullSimTraceCons}$$

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.

1.3.3 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 **??**) and bounded (see Definition **??**) 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 $\theta_s \in \texttt{list}(S(sitpn))$ 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 $\theta_\sigma \in \texttt{list}(\Sigma)$ 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 ($\emptyset$), design d elaborates into Δ and yields the simulation trace θ_σ after τ simulation cycles:*
  $\mathcal{D}_\mathcal{H}, \Delta, \emptyset, E_p, \tau \vdash \mathrm{d} \xrightarrow{full} \theta_\sigma$
- *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 $\theta_s \in \texttt{list}(S(sitpn))$, and a $b \in P \to \mathbb{N}$, let us show that

$$\boxed{\exists \Delta, \theta_\sigma, \forall E_p,\ \gamma \vdash E_p \overset{env}{=} E_c \Rightarrow (\mathcal{D}_\mathcal{H}, \Delta, \emptyset, E_p, \tau \vdash \mathrm{d} \xrightarrow{full} \theta_\sigma) \wedge \theta_s \sim \theta_\sigma}$$

Appealing to Theorems 2 (p. 18), 3 (p. 18) and 4 (p. 18), let us take an elaborated design $\Delta \in ElDesign$, 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:
  $\mathcal{D}_\mathcal{H}, \emptyset \vdash d \xrightarrow{elab} (\Delta, \sigma_e)$
- σ_0 is the initial simulation state: $\mathcal{D}_\mathcal{H}, \Delta, \sigma_e \vdash d.cs \xrightarrow{init} \sigma_0$
- Design d yields the simulation trace θ_σ after τ simulation cycles, starting from initial state σ_0:
  $\mathcal{D}_\mathcal{H}, E_p, \Delta, \tau, \sigma_0 \vdash \mathrm{d.cs} \rightarrow \theta_\sigma$

Let us use this Δ and this θ_σ to prove the current goal. Given an E_p such that $\gamma \vdash E_p \overset{env}{=} E_c$, it remains to be proved that:

$$\boxed{(\mathcal{D}_\mathcal{H}, \Delta, \emptyset, E_p, \tau \vdash \mathrm{d} \xrightarrow{full} \theta_\sigma) \wedge \theta_s \sim \theta_\sigma}$$

First, we must prove that $\boxed{(\mathcal{D}_\mathcal{H}, \Delta, \emptyset, E_p, \tau \vdash \mathrm{d} \xrightarrow{full} \theta_\sigma)}$ holds. By definition of the $\mathcal{H}$-VHDL full simulation relation, we have:

$$\begin{aligned} \mathcal{D}_\mathcal{H}, \Delta, \emptyset, E_p, \tau \vdash \mathrm{d} \xrightarrow{full} \theta_\sigma \equiv \exists \sigma_e, \sigma_0 \in \Sigma(\Delta),\ & \mathcal{D}_\mathcal{H}, \emptyset \vdash d \xrightarrow{elab} (\Delta, \sigma_e) \\ & \wedge \mathcal{D}_\mathcal{H}, \Delta, \sigma_e \vdash d.cs \xrightarrow{init} \sigma_0 \\ & \wedge \mathcal{D}_\mathcal{H}, E_p, \Delta, \tau, \sigma_0 \vdash \mathrm{d.cs} \rightarrow \theta_\sigma \end{aligned} \tag{1.1}$$

Thus, it is equivalent to prove:

$$\boxed{\exists \sigma_e, \sigma_0 \text{ s.t. } \mathcal{D}_\mathcal{H}, \emptyset \vdash d \xrightarrow{elab} (\Delta, \sigma_e) \wedge \mathcal{D}_\mathcal{H}, \Delta, \sigma_e \vdash d.cs \xrightarrow{init} \sigma_0 \wedge \mathcal{D}_\mathcal{H}, E_p, \Delta, \tau, \sigma_0 \vdash \mathrm{d.cs} \rightarrow \theta_\sigma}$$

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. $\boxed{\theta_s \sim \theta_\sigma}$.
□

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* $b \in P \to \mathbb{N}$ *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*:* $\mathcal{D}_{\mathcal{H}}, \emptyset \vdash d \xrightarrow{elab} (\Delta, \sigma_e)$.

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* $sitpn \in SITPN$, $d \in design$, $b \in P \to \mathbb{N}$, $\gamma \in WM(sitpn, d)$, $\Delta \in ElDesign$, $\sigma_e \in \Sigma(\Delta)$ *s.t.*

– $\lfloor sitpn \rfloor_b = (d, \gamma)$ *and* $\lceil sitpn \rceil^b$ *and* $\mathcal{D}_{\mathcal{H}}, \emptyset \vdash d \xrightarrow{elab} (\Delta, \sigma_e)$

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 (Simulation). *For all well-defined* $sitpn \in SITPN$, $d \in design$, $b \in P \to \mathbb{N}$, $\gamma \in WM(sitpn, d)$, $\Delta \in ElDesign$, $\sigma_e, \sigma_0 \in \Sigma$ *s.t.*

– $\lfloor sitpn \rfloor_b = (d, \gamma)$ *and* $\lceil sitpn \rceil^b$ *and* $\mathcal{D}_{\mathcal{H}}, \emptyset \vdash d \xrightarrow{elab} (\Delta, \sigma_e)$ *and* $\mathcal{D}_{\mathcal{H}}, \Delta, \sigma_e \vdash d.cs \xrightarrow{init} \sigma_0$

then there exists a simulation trace $\theta_\sigma \in \texttt{list}(\Sigma)$ *such that for all simulation environment* $E_p \in \mathbb{N} \to Ins(\Delta) \to value$ *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*:*
$\mathcal{D}_{\mathcal{H}}, E_p, \Delta, \tau, \sigma_0 \vdash \text{d.cs} \to \theta_\sigma$

Remark 1 (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.*

1.3.4 The bisimulation theorem

Here, we present the bisimulation theorem. The bisimulation 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 9. 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 bisimulation 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. 23) 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 2); this is done by appealing to Lemma **??** (p. **??**). 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 **??** (p. **??**). The detailed proofs for Lemmas **??** and **??** are given in Sections **??** and **??**.

Theorem 5 (Full bisimulation)**.** *For all $sitpn \in SITPN$, $b \in P \to \mathbb{N}$, $d \in design$, $\gamma \in WM(sitpn, d)$, $\tau \in \mathbb{N}$, $E_c \in \mathbb{N} \to \mathcal{C} \to \mathbb{B}$, $\theta_s \in \texttt{list}(S(sitpn))$, $\Delta \in ElDesign$, $E_p \in \mathbb{N} \to Ins(\Delta) \to value$, $\theta_\sigma \in \texttt{list}(\Sigma)$ such that*

- $\lfloor sitpn \rfloor_b = (d, \gamma)$

– $\gamma \vdash E_p \overset{env}{=} E_c$

– $E_c, \tau \vdash sitpn \xrightarrow{full} \theta_s$

– $\mathcal{D}_\mathcal{H}, \Delta, \emptyset, E_p, \tau \vdash \mathrm{d} \xrightarrow{full} \theta_\sigma$

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 $\theta_s = [s_0]$

  – $\mathcal{D}_\mathcal{H}, \emptyset \vdash d \xrightarrow{elab} (\Delta, \sigma_e)$ 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 **??** (p. **??**), 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, $\mathcal{D}_\mathcal{H}, \Delta, \emptyset, E_p, \tau \vdash \mathrm{d} \xrightarrow{full} \theta_\sigma$), 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 $\theta_s = s_0 :: s_0 :: s :: \theta$

  – $\mathcal{D}_\mathcal{H}, \emptyset \vdash d \xrightarrow{elab} (\Delta, \sigma_e)$ and $\Delta, \sigma_e \vdash d.cs \xrightarrow{init} \sigma_0$ and $E_p, \Delta, \tau, \sigma_0 \vdash \mathrm{d.cs} \to \theta'$ 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. $E_p, \Delta, \tau, \sigma_0 \vdash \mathrm{d.cs} \to \theta'$), we have:

  $E_p, \Delta, \tau, \sigma_0 \vdash d.cs \xrightarrow{\uparrow,\downarrow} \sigma, \sigma'$ and $E_p, \Delta, \tau - 1, \sigma' \vdash \mathrm{d.cs} \to \theta''$ 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 **??**, we can show $\gamma \vdash s_0 \sim \sigma_0$.
2. $\boxed{\gamma, E_c, \tau \vdash s_0 \overset{\uparrow}{\sim} \sigma.}$ Appealing to Lemma **??** (p. **??**), 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 $\gamma, E_c, \tau \vdash s_0 \overset{\uparrow}{\sim} \sigma$.
3. $\boxed{\gamma \vdash s \overset{\downarrow}{\sim} \sigma'.}$ Appealing to Lemma **??** and 3 (p. 27), we have $\gamma \vdash s \overset{\downarrow}{\approx} \sigma'$.

   By definition of $\gamma \vdash s \overset{\downarrow}{\approx} \sigma'$, we can show $\gamma \vdash s \overset{\downarrow}{\sim} \sigma'$.
4. $\boxed{\gamma \vdash \theta \overset{\uparrow}{\sim} \theta''.}$

   Appealing to Lemma **??** and 3, we have $\gamma \vdash s \overset{\downarrow}{\approx} \sigma'$.

   Then, we can appeal to Lemma 1 to show $\gamma \vdash \theta \overset{\uparrow}{\sim} \theta''$.

□

To prove the semantic preservation property, we want to prove that a given SITPN and its translated $\mathcal{H}$-VHDL version follow the bisimulation diagram of Figure 1.3.

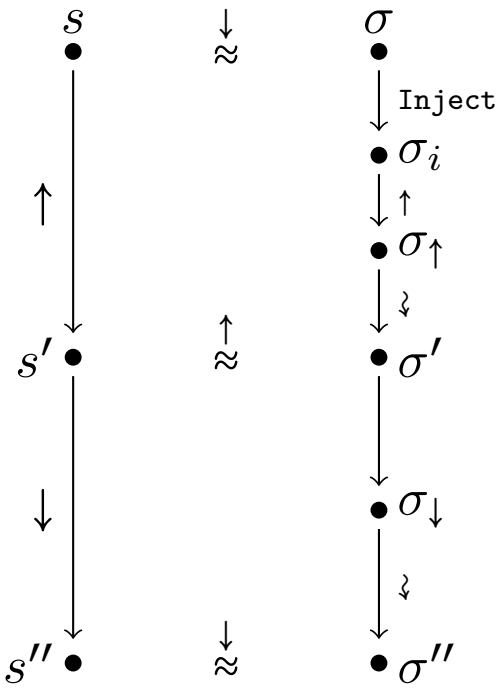

FIGURE 1.3: Bisimulation diagram over one clock cycle for a source SITPN and a target $\mathcal{H}$-VHDL design; the left part of the diagram presents the execution of an SITPN over one clock cycle, and the right part of the diagram presents the simulation of an $\mathcal{H}$-VHDL design over one clock 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. 25). 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 1.3). 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. 27). 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 1.3). 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.

The proof of Lemma 1 is based on the bisimulation diagram of Figure 1.3. 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 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 (Bisimulation). *For all* $sitpn \in SITPN$, $b \in P \to \mathbb{N}$, $d \in design$, $\gamma \in WM(sitpn, d)$, $\Delta \in ElDesign$, $E_p \in \mathbb{N} \to Ins(\Delta) \to value$, $E_c \in \mathbb{N} \to \mathcal{C} \to \mathbb{B}$, $\tau \in \mathbb{N}$, $s \in S(sitpn)$, $\sigma_e, \sigma \in \Sigma$, $\theta_s, \theta_\sigma \in \texttt{list}(\Sigma)$, *such that*

- $\lfloor sitpn \rfloor_b = (d, \gamma)$ *and* $\gamma \vdash E_p \overset{env}{=} E_c$ *and* $\mathcal{D}_\mathcal{H}, \emptyset \vdash d \xrightarrow{elab} \Delta, \sigma_e$
- *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 $\mathcal{D}_\mathcal{H}, \emptyset \vdash d \xrightarrow{elab} \Delta, \sigma_e$, let us show

> $\forall s, \sigma, \theta_s, \theta_\sigma$ s.t. $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$ and $E_c, \tau \vdash sitpn, s \to \theta_s$ and $E_p, \Delta, \tau, \sigma \vdash d.cs \to \theta_\sigma$ 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

  > $\forall s, \sigma, \theta_s, \theta_\sigma$ s.t. $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$ and $E_c, \tau \vdash sitpn, s \to \theta_s$ and $E_p, \Delta, \tau\sigma \vdash d.cs \to \theta_\sigma$ 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:

- $E_c, \tau + 1 \vdash s \xrightarrow{\uparrow} s'$ and $E_c, \tau + 1 \vdash s' \xrightarrow{\downarrow} s''$ and $E_c, \tau \vdash sitpn, s'' \to \theta$.
- $\texttt{Inject}(\sigma, E_p, \tau + 1, \sigma_i)$
- $\Delta, \sigma_i \vdash \text{d.cs} \xrightarrow{\uparrow} \sigma_\uparrow$ and $\Delta, \sigma_\uparrow \vdash \text{d.cs} \xrightarrow{\rightsquigarrow} \sigma'$
- $\Delta, \sigma' \vdash \text{d.cs} \xrightarrow{\downarrow} \sigma_\downarrow$ and $\Delta, \sigma_\downarrow \vdash \text{d.cs} \xrightarrow{\rightsquigarrow} \sigma''$
- $E_p, \Delta, \tau, \sigma'' \vdash d.cs \to \theta'$
- $\theta_s = s' :: s'' :: \theta$ 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. 27), we have $\gamma \vdash s' \overset{\uparrow}{\approx} \sigma'$.

   By definition of $\gamma \vdash s' \overset{\uparrow}{\approx} \sigma'$, we can show $\gamma \vdash s' \overset{\uparrow}{\sim} \sigma'$.

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 $\gamma \vdash s' \overset{\downarrow}{\sim} \sigma'$.

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 $\gamma \vdash s'' \overset{\downarrow}{\approx} \sigma''$.

□

Now, let us present Lemma 2 and Lemma 3, along with their proofs. In the two lemmas, we added a 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 1.4 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 **??**. 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 1.4.

Lemma 2 (Rising edge). *For all* $sitpn \in SITPN$, $b \in P \to \mathbb{N}$, $d \in design$, $\gamma \in WM(sitpn, d)$, $E_c \in \mathbb{N} \to \mathcal{C} \to \mathbb{B}$, $\Delta \in ElDesign$, $E_p \in \mathbb{N} \to Ins(\Delta) \to value$, $\tau \in \mathbb{N}$, $s, s' \in S(sitpn)$, $\sigma_e, \sigma, \sigma_i, \sigma_\uparrow, \sigma' \in \Sigma$, *such that*

- $\lfloor sitpn \rfloor_b = (d, \gamma)$ *and* $\gamma \vdash E_p \overset{env}{=} E_c$ *and* $\mathcal{D}_\mathcal{H}, \emptyset \vdash \mathrm{d} \xrightarrow{elab} \Delta, \sigma_e$
- $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$
- $E_c, \tau \vdash s \xrightarrow{\uparrow} s'$
- $\texttt{Inject}(\sigma, E_p, \tau, \sigma_i)$ *and* $\mathcal{D}_\mathcal{H}, \Delta, \sigma_i \vdash \mathrm{d.cs} \xrightarrow{\uparrow} \sigma_\uparrow$ *and* $\mathcal{D}_\mathcal{H}, \Delta, \sigma_\uparrow \vdash \mathrm{d.cs} \xrightarrow{\rightsquigarrow} \sigma'$
- *State* σ *is a stable design state:* $\mathcal{D}_\mathcal{H}, \Delta, \sigma \vdash \mathrm{d.cs} \xrightarrow{comb} \sigma$

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 **??** lemma (p. **??**) to solve Point 1.
- Apply the **??** lemma (p. **??**) to solve Point 2.
- Apply the **??** lemma (p. **??**) to solve Point 3.
- Apply the **??** lemma (p. **??**) to solve Point 4.
- Apply the **??** lemma (p. **??**) to solve Point 5.
- Apply the **??** lemma (p. **??**) to solve Point 6.
- Apply the **??** lemma (p. **??**) to solve Point 7.
- Apply the **??** lemma (p. **??**) to solve Point 8.

– Apply the **??** lemma (p. **??**) to solve Point 9.

All the lemmas used above, and their corresponding proofs, are to be found in Appendix **??**, Section **??**. □

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* $\mathcal{D}_\mathcal{H}, \emptyset \vdash \mathrm{d} \xrightarrow{elab} \Delta, \sigma_e$

– $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$

– $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$

– $\Delta, \sigma \vdash \mathrm{d.cs} \xrightarrow{\downarrow} \sigma_\downarrow$ *and* $\Delta, \sigma_\downarrow \vdash \mathrm{d.cs} \xrightarrow{\rightsquigarrow} \sigma'$

– *State* σ *is a stable design state:* $\mathcal{D}_\mathcal{H}, \Delta, \sigma \vdash \mathrm{d.cs} \xrightarrow{comb} \sigma$

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. $\forall p \in P, id_p \in Comps(\Delta)\ s.t.\ \gamma(p) = id_p,\ s'.M(p) = \sigma'(id_p)(\texttt{s\_marking})$.
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. $\forall c \in \mathcal{C}, id_c \in Ins(\Delta)\ s.t.\ \gamma(c) = id_c,\ s'.cond(c) = \sigma'(id_c)$.
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. $\forall t \in T, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t,$
   $t \in Firable(s') \Leftrightarrow \sigma'(id_t)(\texttt{s\_firable}) = \texttt{true}$.
7. $\forall t \in T, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t,$
   $t \notin Firable(s') \Leftrightarrow \sigma'(id_t)(\texttt{s\_firable}) = \texttt{false}$.
8. $\forall t \in T, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t, t \in Fired(s') \Leftrightarrow \sigma'(id_t)(\texttt{fired}) = \texttt{true}$.
9. $\forall t \in T, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t, t \notin Fired(s') \Leftrightarrow \sigma'(id_t)(\texttt{fired}) = \texttt{false}$.
10. $\forall p \in P, id_p \in Comps(\Delta)\ s.t.\ \gamma(p) = id_p,$
    $\sum_{t \in Fired(s')} pre(p,t) = \sigma'(id_p)(\texttt{s\_output\_token\_sum})$.
11. $\forall p \in P, id_p \in Comps(\Delta)\ s.t.\ \gamma(p) = id_p,$
    $\sum_{t \in Fired(s')} post(t,p) = \sigma'(id_p)(\texttt{s\_input\_token\_sum})$.

Each point is proved by a separate lemma:

– Apply the **??** lemma (p. **??**) to solve Point 1.

– Apply the **??** lemma (p. **??**) to solve Point 2.

– Apply the **??** lemma (p. **??**) to solve Point 3.

– Apply the **??** lemma (p. **??**) to solve Point 4.

– Apply the **??** lemma (p. **??**) to solve Point 5.

– Apply the **??** lemma (p. **??**) to solve Point 6.

– Apply the **??** lemma (p. **??**) to solve Point 7.

– Apply the Falling edge equal fired lemma (p. 30) to solve Point 8.

– Apply the **??** lemma (p. **??**) to solve Point 9.

– Apply the **??** lemma (p. **??**) to solve Point 10.

– Apply the **??** lemma (p. **??**) to solve Point 11.

All the lemmas used above, and their corresponding proofs, are to be found in Appendix **??**, Section **??**. □

1.4 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 technics. 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.

1.4.1 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 10 (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:* $\gamma \vdash E_p \overset{env}{=} E_c$

- Δ *is the elaborated version of design* d*, and* σ_e *is the default design state of* Δ*:* $\mathcal{D}_{\mathcal{H}}, \emptyset \vdash \mathrm{d} \xrightarrow{elab} \Delta, \sigma_e$
- *Starting states are similar according to the full post rising edge similarity relation:* $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$
- *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* $\Delta, \sigma_\downarrow \vdash \mathrm{d.cs} \xrightarrow{\rightsquigarrow} \sigma'$
- *State* σ *is a stable design state:* $\mathcal{D}_{\mathcal{H}}, \Delta, \sigma \vdash \mathrm{d.cs} \xrightarrow{comb} \sigma$

The hypotheses of Definition 10 are used in all the lemmas expressing properties of the falling edge phase. Therefore, Definition 10 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 10, then for all* $t \in T, id_t \in Comps(\Delta)$ *s.t.* $\gamma(t) = id_t$, $t \in Fired(s') \Leftrightarrow \sigma'(id_t)(\texttt{fired}) = \texttt{true}$.

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 `fired` port of id_t equals `true` at the end of the falling edge phase. Formally, we want to prove: $\boxed{t \in Fired(s') \Leftrightarrow \sigma'(id_t)(\texttt{fired}) = \texttt{true}.}$

As a reminder, the expression $Fired(s')$ qualifies the set of fired transitions at the SITPN state s', and $\sigma'(id_t)(\texttt{fired})$ denotes the value of the `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 **??**) 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 **??**) and sensitized by the *residual* marking (see Sections **??** and **??**) at the considered SITPN state. Figure 1.4 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.

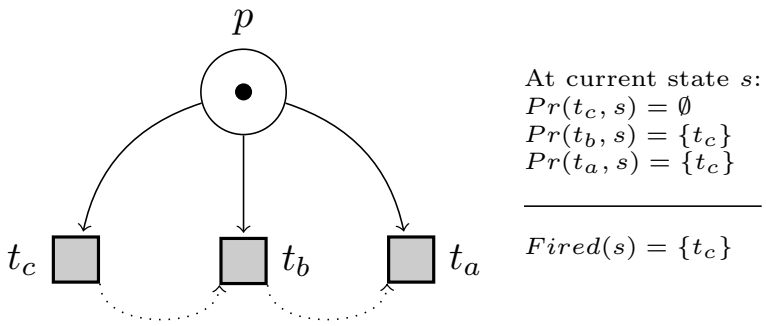

FIGURE 1.4: The set of fired transitions on an example SITPN at a given SITPN state s; on the left side, the dotted arrows indicates the priority relation between the three transitions (t_c is the top-priority transition); on the right side, each transition is associated to its Pr set, which is necessary 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 1.4). 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 **??**). 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 1.4, 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 1.5 shows a part of the transition design architecture describing how the `fired` port is connected to the other internal signals.

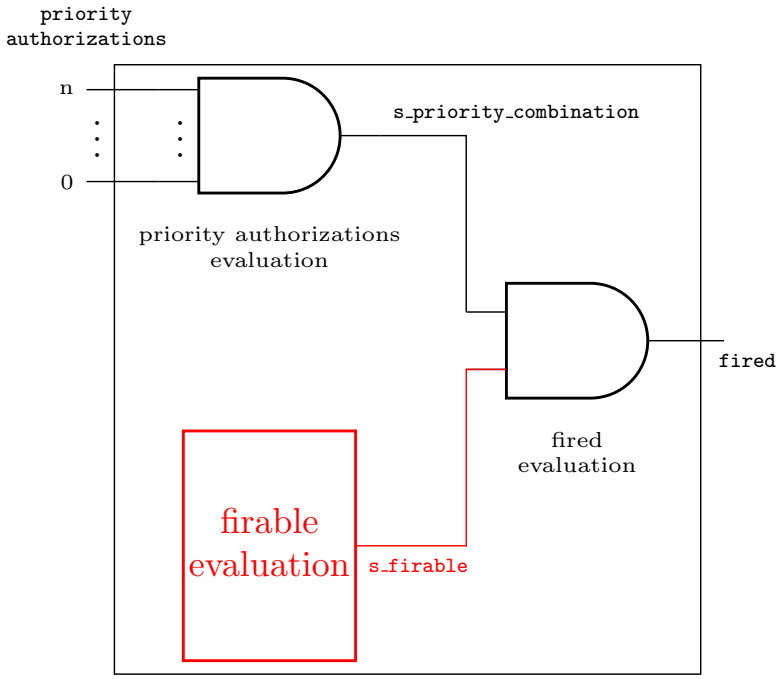

FIGURE 1.5: Wiring of the `fired` output port in the transition design architecture; on the left side is the input interface of the transition design; on the right side is the output interface of the transition design, with the `fired` port; in red are the parts of the architecture that depend on synchronous logic and in black are the parts that are purely combinational.

In Figure 1.5, 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 1.5 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:

$$\sigma(id_t)(\texttt{fired}) = \sigma(id_t)(\texttt{s\_firable}) \,.\, \sigma(id_t)(\texttt{s\_priority\_combination}) \tag{1.2}$$

Equation (1.2) 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 2 (Signals and combinational equations)**.** *In the proceeding of the proof, a lot of combinational equations are established (e.g, Equation (1.2)). 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 $\mathcal{H}$-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 1.1, assigns the fired output port. Reasoning on the fired_evaluation process statement body and on the $\mathcal{H}$-VHDL semantics rules permits us to deduce Equation (1.2).

```
fired_evaluation : process (s_firable, s_priority_combination)
begin
   fired ⇐ s_firable and s_priority_combination;
end process fired_evaluation;
```

LISTING 1.1: The fired_evaluation process in the transition design architecture; its statement body assigns the fired output port; symbol ⇐ is the signal assignment operator.

Listing 1.2 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 1.2: 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 (1.3) 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.

$$\sigma(id_t)(spc) = \prod_{i=0}^{\Delta(id_t)(input\_arcs\_number)-1} \sigma(id_t)(priority\_authorizations)[i] \qquad (1.3)$$

In Equation (1.3), 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 **??** *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* **??** *and* **??***).*

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 **??**, 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 (1.3), we can replace the value of the `s_priority_combination` signal by its equivalent product expression:

$$t \in Sens(s'.M - \sum_{t_i \in Pr(t,s')} pre(t_i)) \Leftrightarrow$$
$$\Big(\prod_{i=0}^{\Delta(id_t)(\texttt{input\_arcs\_number})-1} \sigma'(id_t)(\texttt{priority\_authorizations})[i]\Big) = \texttt{true}$$

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

$$\Big(\prod_{i=0}^{\Delta(id_t)(\texttt{ian})-1} \sigma'(id_t)(\texttt{pauths})[i]\Big) = \texttt{true}.$$

2. Assuming $\left(\prod_{i=0}^{\Delta(id_t)(\texttt{ian})-1} \sigma'(id_t)(\texttt{pauths})[i]\right) = \texttt{true}$, 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 $t \in Sens(s'.M - \sum_{t_i \in Pr(t,s')} pre(t_i))$, let us show

$$\boxed{\left(\prod_{i=0}^{\Delta(id_t)(\texttt{ian})-1} \sigma'(id_t)(\texttt{pauths})[i]\right) = \texttt{true}.}$$

To prove the current goal, we can equivalently show that:

$$\boxed{\forall i \in [0, \Delta(id_t)(\texttt{ian}) - 1],\ \sigma'(id_t)(\texttt{pauths})[i] = \texttt{true}.}$$

For a given $i \in [0, \Delta(id_t)(\texttt{ian}) - 1]$, let us show that $\boxed{\sigma'(id_t)(\texttt{pauths})[i] = \texttt{true}.}$ As shown in Figure 1.5, the `priority_authorizations` signal is an input port of the `transition` design. Therefore, to know what is the value of the element 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 **??**, p. **??**), 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 1.6 shows the connection of the `priority_authorizations` port between the component instances id_p and id_t.

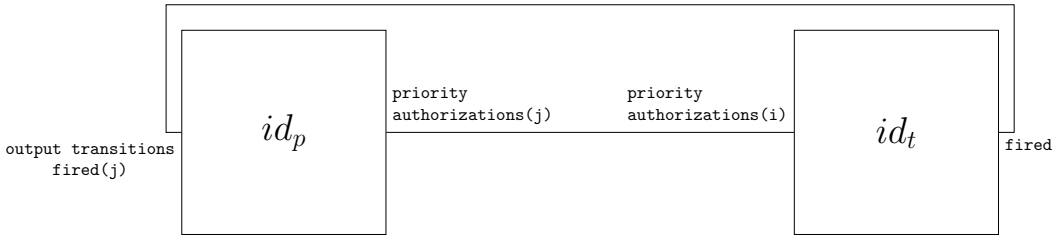

FIGURE 1.6: Connection of the j-th element of the `priority_authorizations` output port of the PCI id_p to the i-th element of the `priority_authorizations` input port of the TCI id_t; also the `fired` output port of id_t is connected to the j-th element of the `output_transitions_fired` input port of 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 $\boxed{\sigma'(id_t)(\texttt{pauths})[i] = \texttt{true}}$, 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 determine the combinational equation associated to the j-th element of the `priority_authorizations` output port. Figure 1.7 illustrates the wiring of the `priority_authorizations` output port in a `place` design.

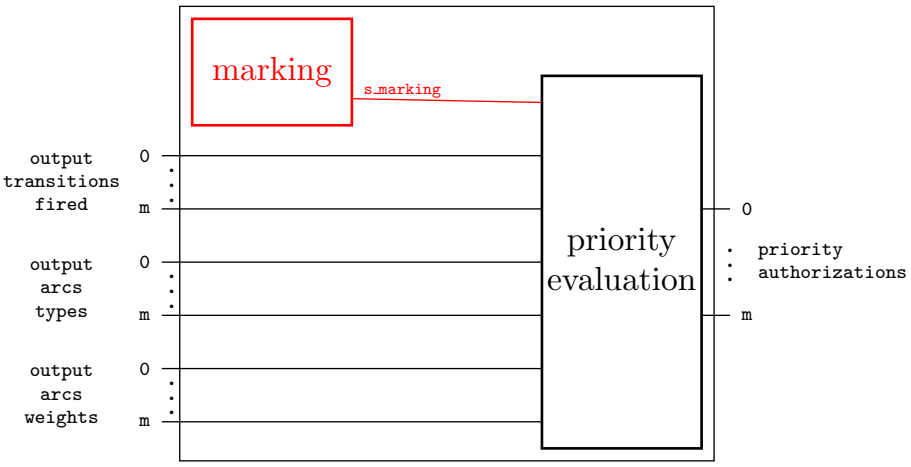

FIGURE 1.7: Wiring of the `priority_authorizations` output port in the architecture of the place design; the input port interface is on the left side and the output port interface is on the right side; the synchronous parts are in red and the combinational ones are in black.

Figure 1.7 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 1.7, 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 1.3 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 1.3: 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':

$$\sigma'(id_p)(\texttt{pauths})[j] = (\sigma'(id_p)(\texttt{s\_marking}) - \texttt{vsots} \geq \sigma'(id)(\texttt{output\_arcs\_weights})[j]) \quad (1.4)$$

Then, rewriting the goal with Equation (1.4), 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 1.3 (Lines 10 to 12), we can deduce the value of the `vsots` variable:

$$\texttt{vsots} = \sum_{l=0}^{j-1} \begin{cases} \sigma'(id_p)(\texttt{oaw})[l] \texttt{ if } \sigma'(id_p)(\texttt{otf})[l] \,.\, (\sigma'(id_p)(\texttt{oat})[l] = \texttt{basic}) \\ 0 \textit{ otherwise} \end{cases} \quad (1.5)$$

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 1.8 reuses the SITPN of Figure 1.4 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).

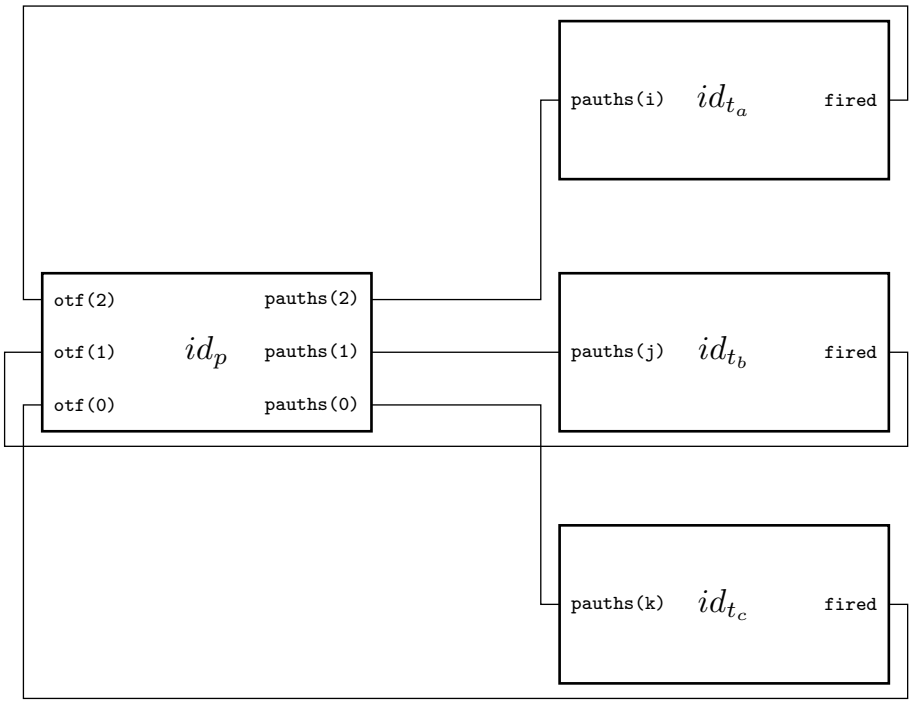

FIGURE 1.8: Connection between the `priority_authorizations` output port of PCI id_p and the `priority_authorizations` input port of TCIs id_{t_a}, id_{t_b} and id_{t_c}, and between the `output_transitions_fired` input port of id_p and the `fired` ports of id_{t_a}, id_{t_b} and id_{t_c}. `pauths` stands for `priority_authorizations` and `otf` stands for `output_transitions_fired`.

In Figure 1.8, 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:

$$\sigma'(id_p)(\texttt{s\_marking}) - \texttt{vsots} \geq \sigma'(id)(\texttt{output\_arcs\_weights})[j].$$

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 **??**), 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 **??**, 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 `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:

$$\sum_{t_i \in Pr(t,s')} \begin{cases} \omega \texttt{ if } pre(p,t_i) = (\omega, \texttt{basic}) \\ 0 \textit{ otherwise} \end{cases}$$
$$=$$
$$\sum_{l=0}^{j-1} \begin{cases} \sigma'(id_p)(\texttt{oaw})[l] \texttt{ if } \sigma'(id_p)(\texttt{otf})[l] \,.\, (\sigma'(id_p)(\texttt{oat})[l] = \texttt{basic}) \\ 0 \textit{ otherwise} \end{cases}$$

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 **??** in Section **??**). On the lower part is the full definition of the `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:

$$t_i \in Pr_p \equiv t_i \succ t \wedge \exists \omega \textit{ s.t. } pre(p,t_i) = (\omega, \texttt{basic}) \wedge t_i \in Fired(s')$$

Let us also define the IPr_p set, which the set of indexes from 0 to $j-1$ for which the `otf` port of id_p equals `true` at state σ' and the `oat` port of id_p equals `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: $\sum_{t_i \in Pr_p} \texttt{fst}(pre(p,t_i)) = \sum_{l \in IPr_p} \sigma'(id_p)(\texttt{oaw})[l]$

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 `priority_authorizations` output port of id_p reflects the priority order of the conflicting output transitions of place p (see Figure 1.8). 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 `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 2.

Algorithm 2: fired(s)

Data: An SITPN state s
Result: Returns the set of fired transitions at state s

1 $F \leftarrow \emptyset$
2 $T_s \leftarrow T$
3 **while** $T_s \neq \emptyset$ **do**
4 $t \leftarrow \texttt{GetTopPriorityTransition}(T_s, \succ)$
5 **if** $t \in Firable(s)$ ***and*** $t \in Sens(s.M - \sum_{t_i \in Pr(t,F)} pre(t_i))$ **then** $F \leftarrow F \cup \{t\}$
6
7 $T_s \leftarrow T_s \setminus \{t\}$
8 **return** F

Algorithm 2 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 $s.M - \sum_{t_i \in Pr(t,F)} pre(t_i)$. 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 2 through the definition of the $IsFiredSet$ and the $IsFiredSetAux$ relations given in Definition 12 and 13. Definition 11 states that a given transition is fired in relation to the $IsFiredSet$ relation.

Definition 11 (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 12 (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 13 (IsFiredSetAux). *The $IsFiredSetAux$ relation is defined by the following rules:*

FSetEmp
$$\frac{}{IsFiredSetAux(s, \emptyset, F, F)}$$

FSetFired
$$\frac{t \in Firable(s) \quad t \in Sens(s.M - \sum_{t_i \in Pr(t,F)} pre(t_i)) \quad IsFiredSetAux(s, T_s, F \cup \{t\}, Fset)}{IsFiredSetAux(s, T_s \cup \{t\}, F, Fset)} \quad \begin{array}{l} \nexists t' \in T_s \text{ s.t. } t' \succ t \\ Pr(t, F) = \{t' \mid t' \succ t \wedge t' \in F\} \end{array}$$

FSetNotFirable
$$\frac{t \notin Firable(s) \quad IsFiredSetAux(s, T_s, F, Fset)}{IsFiredSetAux(s, T_s \cup \{t\}, F, Fset)} \quad \nexists t' \in T_s \text{ s.t. } t' \succ t$$

FSetNotSens
$$\frac{t \notin Sens(s.M - \sum_{t_i \in Pr(t,F)} pre(t_i)) \quad IsFiredSetAux(s, T_s, F, Fset)}{IsFiredSetAux(s, T_s \cup \{t\}, F, Fset)} \quad \begin{array}{l} \nexists t' \in T_s \text{ s.t. } t' \succ t \\ Pr(t, F) = \{t' \mid t' \succ t \wedge t' \in F\} \end{array}$$

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 **??**.

The full formal proof of Lemma **??** is given in Section **??** of Appendix **??**. 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 **??**.

1.4.2 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 1.9 shows a case of divergence between time counters and `s_time_counter` signals values in the course of an execution.

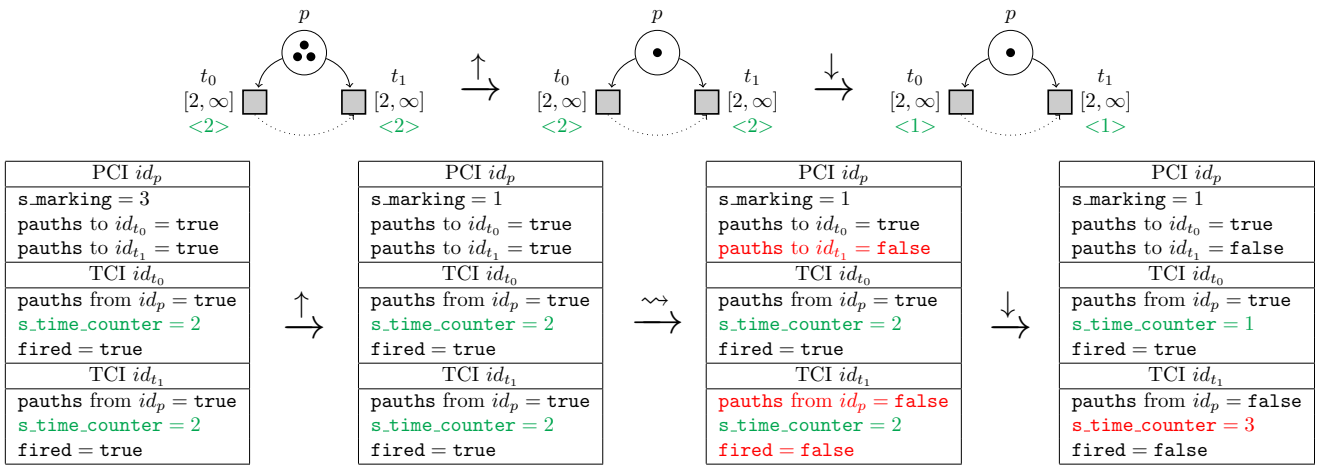

FIGURE 1.9: Bug detection: divergence between the value of time counters and the value of the `s_time_counter` signals due to the loss of the firing status information during the stabilization phase; the value of time counters and of the `s_time_counter` signals are in green; the value of diverging signals are in red.

In Figure 1.9, 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 hold (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 **??**). 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 **??**, p. **??**, and **??**, p. **??**, about the equivalence of time counters).

1.5 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 thirty per cent of the proof concerning the **??** lemma. However, the effort to achieve this thirty per cent of the 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 **??**) 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 1.4 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, Simulation theorems as axioms in the `Soundness.v` file under the `soundness`

folder of the Git repository.

```
1  Theorem sitpn2hvhdl_full_bisim :
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 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 1.4: Coq implementation of the Full bisimulation theorem and the mechanized version of its proof.

The proof laid out in Listing 1.4 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; we name it `Hsimloop`. 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 bisimulation theorem.

Robustness to change

The proof laid out in Listing 1.4 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 [8]. 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 automate 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 involve in the HILECOP transformation function carry a compile-time state, defined as the Coq type `CompileTimeState`. Each function either return a value, modify the compile-time state or do both. To give an example of the use of the `minv` tactic, Listing 1.5 shows the implementation of the `generate_place_comp_inst` function 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 1.5: 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 intermediate 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 [15] 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 bisimulation theorem given in this Chapter and in Appendix **??** and the machine-checked formal proof. Right now, 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

[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.

Lemma **??** 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.

1.6 Conclusion

In this chapter, the aim was to present the behavior preservation theorem stating that the HILECOP 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 describe 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 kind of states is not straightforward, and we thoroughly presented our state similarity relation in Section 1.2.

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 to kind 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 1.3 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 $\mathcal{H}$-VHDL design resulting from the HILECOP transformation 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 **??**. 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 **??** is long; about a hundred pages. However, as we explained in Section 1.4, 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 behavior 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 1.5, 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 [23], 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.

Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] Yong Kiam Tan et al. "A New Verified Compiler Backend for CakeML". In: (), p. 14.

[21] The Coq Development Team. *Coq, Version 8.13.2*. manual. July 2021.

[22] 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.

[23] Freek Wiedijk. "The De Bruijn Factor". In: (Aug. 12, 2000).

[24] 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.

[25] 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.

[26] 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.

[27] 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.