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

En Informatique

École doctorale : Information, Structures, Systèmes

Unité de recherche LIRMM

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

Présenté par Vincent IAMPIETRO

Le Date de la soutenance

Sous la direction de David Delahaye
et David Andreu

Devant le jury composé de

[Nom Prénom], [Titre], [Labo]	[Statut jury]
[Nom Prénom], [Titre], [Labo]	[Statut jury]
[Nom Prénom], [Titre], [Labo]	[Statut jury]

Acknowledgements

The acknowledgments and the people to thank go here, don't forget to include your project advisor…

Contents

List of Figures

List of Tables

List of Abbreviations

SITPN	**S**ynchronously executed **I**nterpreted **T**ime **P**etri **N**et with priorities
VHDL	**V**ery high speed integrated circuit **H**ardware **D**escription **L**anguage
PCI	**P**lace **C**omponent **I**nstance
TCI	**T**ransition **C**omponent **I**nstance
GPL	**G**eneric **P**rogramming **L**anguage
HDL	**H**ardware **D**escription **L**anguage
LRM	**L**anguage **R**eference **M**anual

For/Dedicated to/To my...

Chapter 1

The HILECOP methodology

In this chapter, we present the context of our work, and more specifically, the subject of our verification task, i.e. HILECOP, a methodology for the design and implementation of critical digital systems. In Section 1.1, we motivate the use of Model-Based Systems Engineering (MBSE) and formal methods in the design and production of safety-critical digital systems; in Section 1.2, we give an overall presentation of the HILECOP methodology which applies both the principles of MBSE and formal methods; in Section 1.3, we point out the specific transformation phase, intervening in the HILECOP methodology, that we propose to verify and discuss on how we propose to verify it and which research questions does it give rise to.

1.1 Designing critical digital systems

According to Moore's law [41], the complexity of digital integrated circuits is always increasing. To give an example, the cut-of-the-edge *AMD Epyc Rome* microprocessor (2019) is made out of 50 billions of transistors. Composing billions of transistors on a wired circuit is no more a task for humans but is very suited to computers. However, engineers need to think about the design of digital circuits in a way that is understandable for humans. Therefore, they need high-level views of the circuits they are designing in order to work together and to communicate about the designs. The domain of Model-Based Systems Engineering (MBSE) [37] proposes a framework to help engineers to design and produce digital circuits, in a well-documented, safe and reliable way. Comparable to what Model Driven Engineering (MDE) does in the world of software engineering, models are first order concepts in MBSE. A model represents a simplified view of real object. As illustrated in Figure 1.1, a MBSE process describes a way to design a digital circuit starting from a high-level view of the system. This high-level view can follow a graphical formalism such as SysML [25] or Petri nets [47], or a textual one such as SystemC [5] or VHDL [2]. Then, the MBSE process describe many refinements phases (the green arrows in Figure 1.1) during which the input model will be transformed; at each refinement phase, the model goes down in abstraction towards its final implementation as a hardware circuit. A refinement phase, which is also a transformation phase, can be performed automatically or manually.

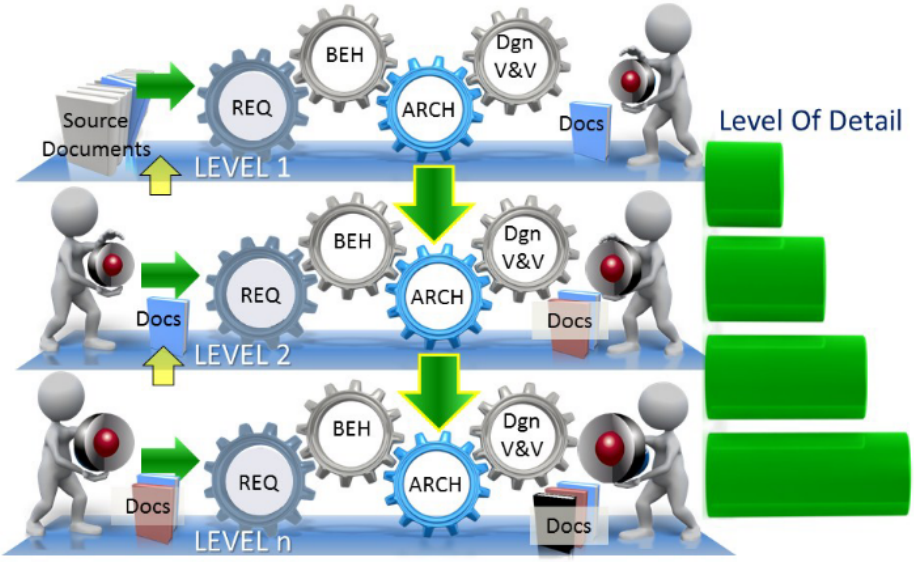


FIGURE 1.1: A Model-Based Systems Engineering process; REQ stands for requirements, BEH for behavior, ARCH for architecture, Dgn V&V for design verification and validation. This figure is an excerpt from [37].

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

1.2 Introducing the HILECOP methodology

The INRIA CAMIN team (former DEMAR team) has developed a new technology of neuroprotheses [29]. Neuroprotheses are medical devices which purpose is to electro-stimulate the nerves of patients suffering from moving disabilities. The nerves are responding to the stimulation, i.e an electric influx, in order to activate the muscles and so that the patient can recover some movements. Thus, controlling the intensity and the form of the electric signal sent to the patient's nerve is a critical point of the device overall functioning. These two parameters are controlled by a digital hardware circuit (i.e. a microcontroller) that is a part of the neuroprosthesis. Therefore, the design of such digital systems becomes utterly critical as a faulty circuit could result in the injury of patients. To assist the engineers in the design and the implementation of these critical digital systems, the CAMIN team came up with a process called the "HILECOP methodology" [1]. This methodology follows the principles of a MBSE process and relies on several transformations going from abstract models to concrete FPGA implementations through the production of VHDL code. Figure 1.2 details the global workflow of HILECOP.

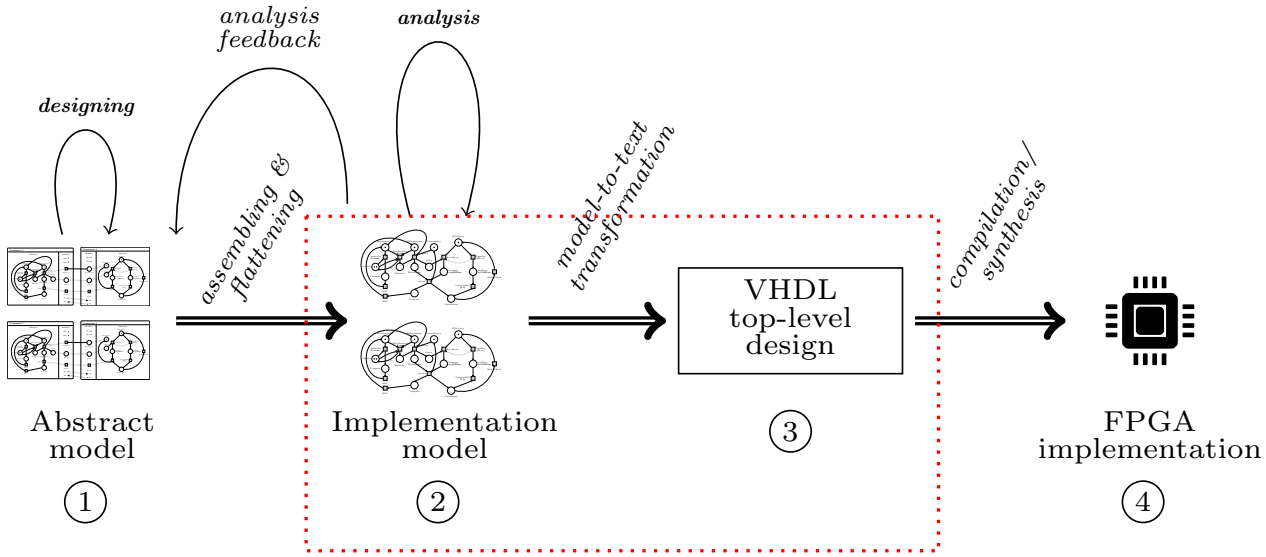


FIGURE 1.2: Workflow of the HILECOP methodology; horizontal double arrows indicate the transformation phases, i.e. the refinement phases in MBSE terms; simple arrows indicate different kinds of operations performed at a given step.

In Figure 1.2, Step 1 corresponds to the design phase of a critical digital system. At this step, the engineers produce a model of the wanted system; the leveraged model formalism is a graphical formalism specially designed for the methodology and based on component diagrams. Figure 1.3 provides an example of such a model.

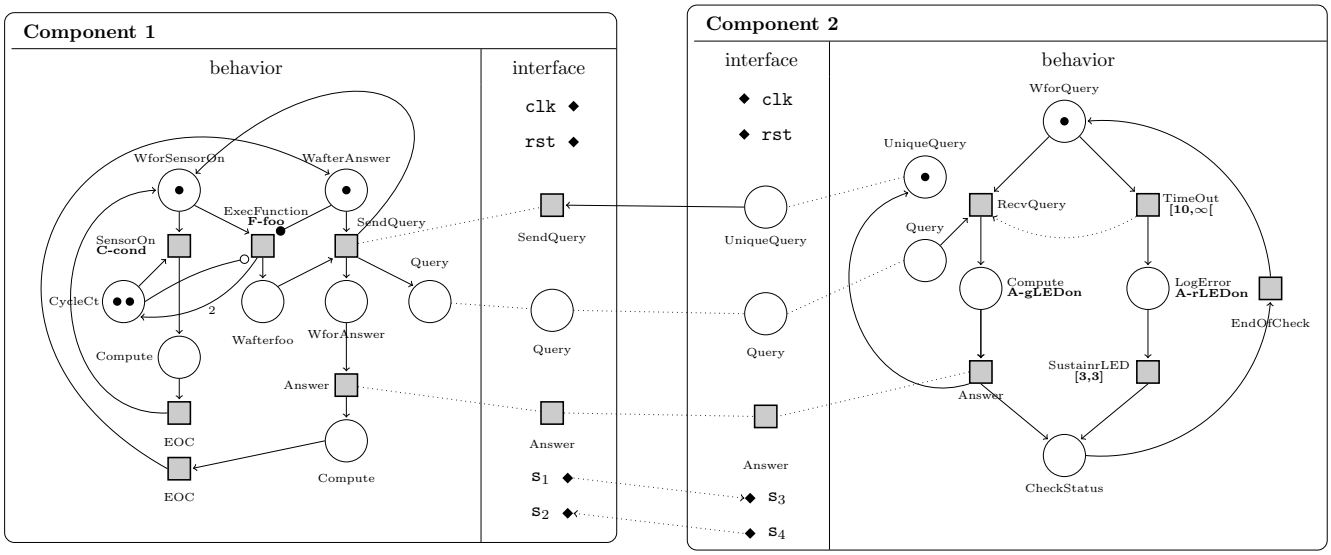


FIGURE 1.3: An Example of HILECOP high-level model. Black diamonds represent VHDL signals.

As shown in Figure 1.3, a component of the HILECOP high-level model formalism is represented by a box having an internal behavior and an interface that permits the connection to

other components. The internal behavior of a component is defined with a specific kind of Petri Net (PN) model. These PNs and their distiguishing features will be thoroughly presented in Chapter 2. The component interface exposes places, transitions, and signals, which are references to the elements of its internal behavior, to the outside so that multiple components can be assembled. Each component has a clock and a reset input port (`clk` and `rst`) in its interface. Indeed, the HILECOP methodology has been built for the design of synchronous digital systems. To a certain extent, VHDL signals can be integrated to the high-level components to represent a direct wiring between components. A component behavior can also be defined through the composition of other components. In that case, we talk about a composite hardware structure.

Next, in Figure 1.2, the transformation from Step 1 to Step 2 flattens the model. The internal behaviors are connected according to the interface compositions, and embedding component structures are removed. Figure 1.4 gives the result of the flattening phase for the model of Figure 1.3.

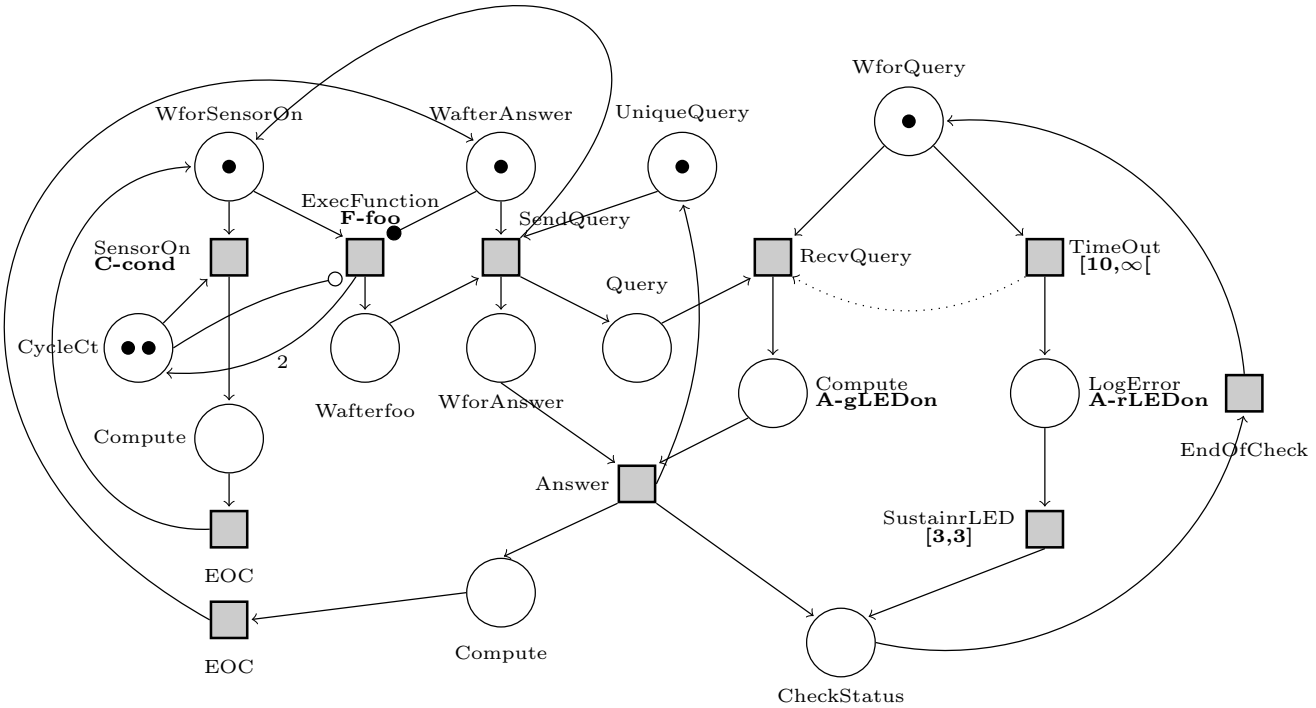


FIGURE 1.4: A global Petri net model obtained after the flattening of a HILECOP high level model.

The PN formalism is a formal model and therefore permits to apply mathematical reasoning on its instances. Particularly, a PN model can be analyzed, and a proof that a given model meet some properties can be automatically produced through the direct analysis of the structure or through the use of model checking techniques. This feature of PNs has been one of the reason of the adoption of this formalism as HILECOP's base formalism. A whole thesis has been dedicated to the development of new methods to analyze the HILECOP PN models [40]. In fact, the transformation of the abstract model is a bit different in preparation of the model analysis. The transformation adds new information to the flattened model to help the analysis.

Figure 1.4 only gives the flattened version of the model produced in preparation of the next transformation into VHDL design. The analysis phase is here to convince the engineers that they are designing a safe system. The analysis process is a round trip between Step 1 and Step 2. It aims at producing a model that is conflict-free (see Section **??** for more details about the definition of a conflict), bounded, and deadlock-free, using model-checking techniques. After several iterations, the model should reach soundness and is then said to be implementation-ready.

From Step 2 to Step 3, VHDL source code is then generated by means of an automatic model-to-text transformation. The generated code describes a VHDL design, i.e. a textual description of a hardware system, which has an interface defining input and output ports and an internal behavior called an architecture. Details about the syntax and the semantics of the VHDL language will be given in Chapter 3. Figure 1.5 succinctly illustrates the transformation happening between Step 2 and Step 3. The transformation from Step 2 to Step 3 will be thoroughly presented in Chapter.

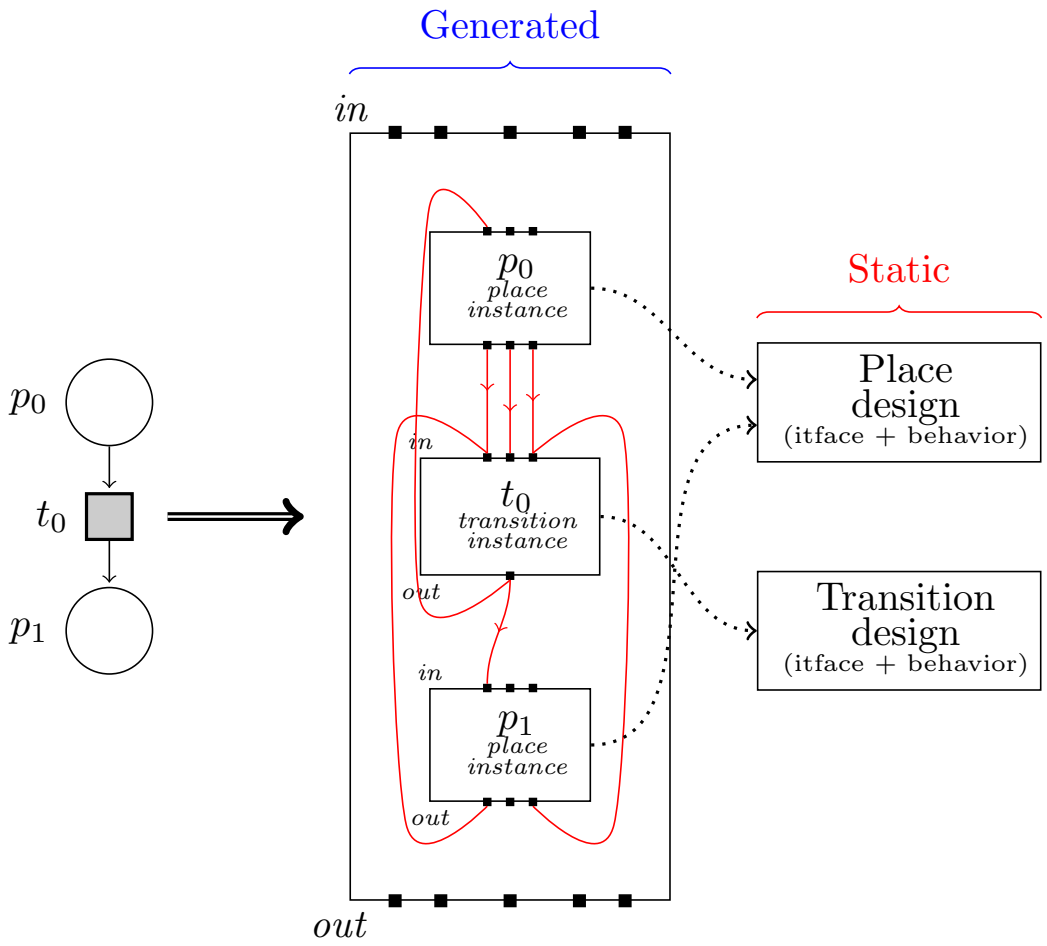


FIGURE 1.5: Generation of a top-level VHDL design from a Petri net. On the left, the input PN and on the right the generated VHDL top-level design. Dotted arrows shows the relation between the component instances and their source design.

For the purpose of the HILECOP methodology, two VHDL designs have been defined: the place design which is a hardware description of a PN place (circle nodes in a PN) and the transition design which is a hardware description of a PN transition (square nodes in a PN). Like all VHDL designs, the place and the transition design have an input and output port interface, and their own internal behavior. A VHDL design is a mould to describe a hardware component. Thus, a design can be instantiated in the behavior of other designs in order to obtain more complex behaviors. As illustrated in Figure 1.5, the transformation from Step 2 to Step 3 creates a place component (or design) instance (PCI) and a transition component (or design) instance (TCI) for each place and transition of the input Petri net. Then, the PCIs and TCIs are connected together through their input and output port interfaces. These connections mimic the arc connections between the places and the transitions of the input PN.

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

1.3 Verifying the HILECOP methodology

The HILECOP methodology is useful to design and implement critical digital systems. The use of Petri nets as the base model of the process is one of its major advantage. All the analysis tools that accompany the Petri net formalism and that permit to prove the safety properties of the models qualify the HILECOP methodology as a formal method for the design and implementation of critical digital systems. However, even with input models that are proved to be safe and sound, the advantages provided by the use Petri nets would be lost if one of the transformation happening during the process changed the input definition of the circuit in a way that would alter its behavior. Thus, the engineers would have specified a perfectly correct digital system but would never obtain the expected circuit on a physical FPGA device. Therefore, in order to reinforce the confidence in the HILECOP methodology as a reliable formal method to produce critical digital systems, the goal of this thesis is to verify, i.e. to establish the formal proof, that the model-to-text transformation from Step 2 to Step 3 (i.e. the framed part with red dotted lines in Figure 1.2) preserves the behavior of input models into the generated VHDL designs. We choose to carry out this task as a verification task (see Section 4.1 for an explanation on the difference between validation and verification tasks). It means that we want to prove a behavior preservation theorem stating that: for all input models of our transformation the generated output designs act similarly at the execution time. Chapter 4 formally presents our behavior preservation theorem, and thus, what we mean about the similarity of execution between our input and output representation.

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

- The transformation from Step 1 to Step 2 changes the structure of the component-based input model. However, if we had to formalize the semantics of the HILECOP high-level models

then it would probably be through the definition of the global PN obtained by transformation at Step 2 (i.e. a kind of translational semantics). Therefore, we argue that there is no need to verify that the transformation from Step 1 to Step 2 is semantic preserving.

- The transformation from Step 3 to Step 4 is performed by industrial tools. We rely on these tools because they are widely used in the industry for the development of critical systems.

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

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

Even though answering this question is interesting, however, its formulation as a question that one can answer only by yes or no does not stimulate the scientific interest of the verification task. However, this task is really close to the verification of compilers for programming languages. Compiler verification has been widely explorated, and many works are accessible in the literature [19]. The major source of inspiration of this thesis has been the work done on the CompCert certified C compiler [36]. Thus, we argue here that the scientific interest of our research comes from the comparison between the methods used to perform our verification task and the methods used to perform similar verification task in other domains such as compiler verification. Thus, we can complement our research question with the following ones:

- What are the similarities and the differences between the HILECOP transformation and other transformation situations (compilers, model transformations...)?
- Is there a strategy to perform the verification of the HILECOP transformation?
- How far the correspondence holds between this strategy and the strategy used in other transformation situations such as compiler verification?

To achieve the formal verification of HILECOP, our approach is similar to what has been done for the CompCert compiler. The idea is to formalize the semantics of the source and target languages, and verify that the transformation preserves the semantics of any input model. In the thesis, we propose both to perform the formalization work on "paper" and to mechanize it within the Coq proof assistant [4].

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

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

To further motivate the necessity of the verification task, the development of neuroprotheses by the INRIA CAMIN team is at the base of the creation of the Neurrinov company[1]. The

[1] http://neurinnov.com/

Neurrinov company is now looking towards the industrial development of such neuroprotheses. We hope that once the verification performed on the HILECOP methodology, it will help to obtain the CE certification[2] necessary to qualify the neuroprotheses as eligible for the medical market.

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

FIGURE 1.6: A view of the HILECOP software implemented on top of the Eclipse framework.

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

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

Chapter 2

Implementation of the HILECOP Petri nets

In this chapter, we present the input formalism of our transformation function: Synchronously executed Interpreted Time Petri Nets with priorities (SITPNs). For the main part, the formalization of the SITPN structure and semantics is the result of two former Ph.D. theses [35, 40]. However, we contributed to the simplification of the definition of the SITPN structure and its semantics. Moreover, we added complementary definitions for the proof of behavior preservation. Our main contribution to this part lies in the implementation of the SITPN structure and semantics with the Coq proof assistant. This chapter is structured as follows: Section 2.1 is a reminder on the PN formalism and also gives an informal presentation of SITPNs; Section 2.2 lays out the formal definitions of the SITPN structure and semantics; Section 2.3 deals with the implementation of SITPNs with the Coq proof assistant.

2.1 Informal presentation of Synchronously executed Petri nets

Here, fundamentals on the Petri net formalism are outlined, and certain classes of Petri nets are described more precisely. Then, the specificities of the Petri nets used to design the behavior of electronic components in the HILECOP methodology are presented. For more information on the topic of Petri nets, the reader can refer to [20], [42], or [22].

2.1.1 Preliminary notions on Petri nets

Petri nets (PNs), invented by C. A. Petri [47], are used to model a broad range of *dynamic* systems: resource sharing between concurrent processes [20], behavior of agents in multi-agent systems [15], behavior of digital components [54]. A Petri net is a directed graph, composed of two types of node: place nodes (*circles*) and transition nodes (*squares* or *lines*). As shown in Figure 2.1, place nodes usually represent a part of the state of the modelled system, here, the states of two computer processes and a semaphore; transition nodes usually refer to events triggering the system evolution (or state changing). We will see later that there exists a lot of different classes of PNs. Figure 2.1 presents an example of the most simple form of PN, namely, the *place-transition* PN. In this chapter, when no precision is given on the class of PN considered, a PN refers to a *place-transition* PN.

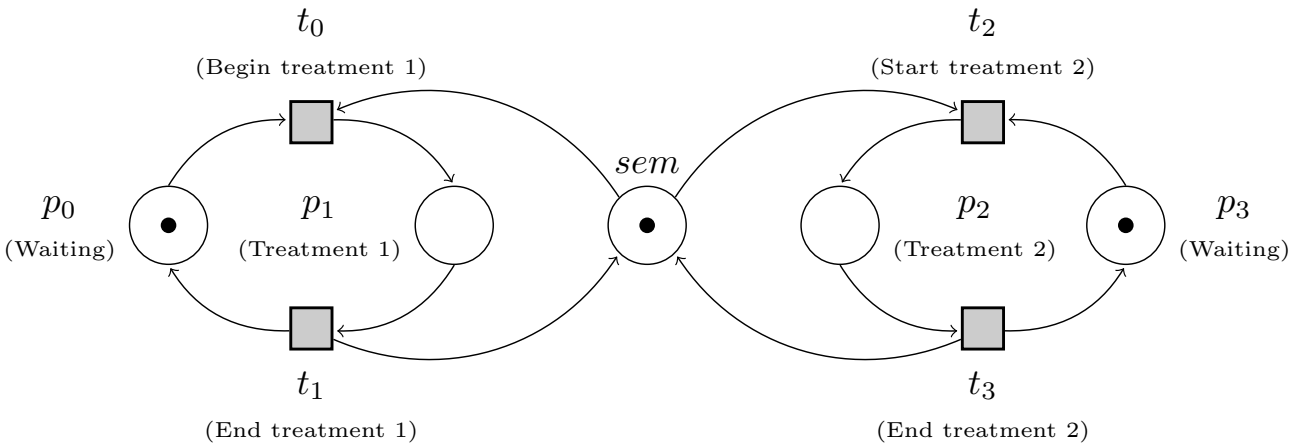


FIGURE 2.1: An example of Petri net. The semaphore place *sem* prevents the parallel execution of *Treatment 1* (place p_1) and *Treatment 2* (place p_2).

Edges

In a Petri net, directed edges link together places and transitions. Places cannot be linked to other places, and the same stands for transitions. There are two kinds of edges, *pre* or *incoming* edges, going from a place to a transition, and *post* or *outcoming* edges, going from a transition to a place. Places linked to a transition t by incoming (resp. outcoming) edges will be referred to as the *input* places (resp. *output* places) of t. The same stands for the transitions linked to a place p. For instance, in Figure 2.1, p_0 and *sem* are the input places of t_0, and p_1 is the output place of t_0; t_1 and t_3 are the input transitions of place *sem*, and t_0 and t_2 are the output transitions of *sem*. Some weight –a natural number– is associated to the edges of a Petri net. If no label appears on the edge then one is the default weight. Petri nets are said to be *generalized* when the weight of the edges are possibly greater than one.

Marking

In Figure 2.1, places p_0, p_3 and *sem* are marked with tokens, represented by little black circles. This means that places p_0, p_3 and *sem* are currently active. The distribution of tokens over places is called the *marking* of the net. The marking of a Petri net reflects the overall state of the modelled system at a certain moment in its activity cycle.

Transition firing

In a Petri net, the marking evolves based on a token consumption-production system. Transitions consume tokens from their input places, and produce tokens to their output places. This whole system is called *transition firing*. In order to be *firable*, a transition must be *sensitized* (or *enabled*), meaning that the number of tokens in each of its input places must be equal or greater than the weight of its incoming edges. For instance, in Figure 2.1, the transition t_0 is sensitized because the weight of the arc (p_0, t_0) is of one (default value), and place p_0 is marked with one token, and the same stands for the number of tokens in place *sem* and the weight of the arc

(*sem*, t_0). As a counter example, transition t_3 is not sensitized because there are no tokens in its input place p_2. Depending on the class of PNs that is considered, other parameters affect the *firability* of transitions (see interpreted Petri nets, time Petri nets and Section 2.1.2). When a sensitized transition is fired, tokens are retrieved from its input places (as many tokens as the weight of the arcs) and produced in its output places (as many tokens as the weight of the arcs). This process represents the occurrence of an event –denoted by the transition– triggering the evolution of the system from one state to another. Figure 2.2 shows the state of the PN of Figure 2.1 after the firing of the transition t_0.

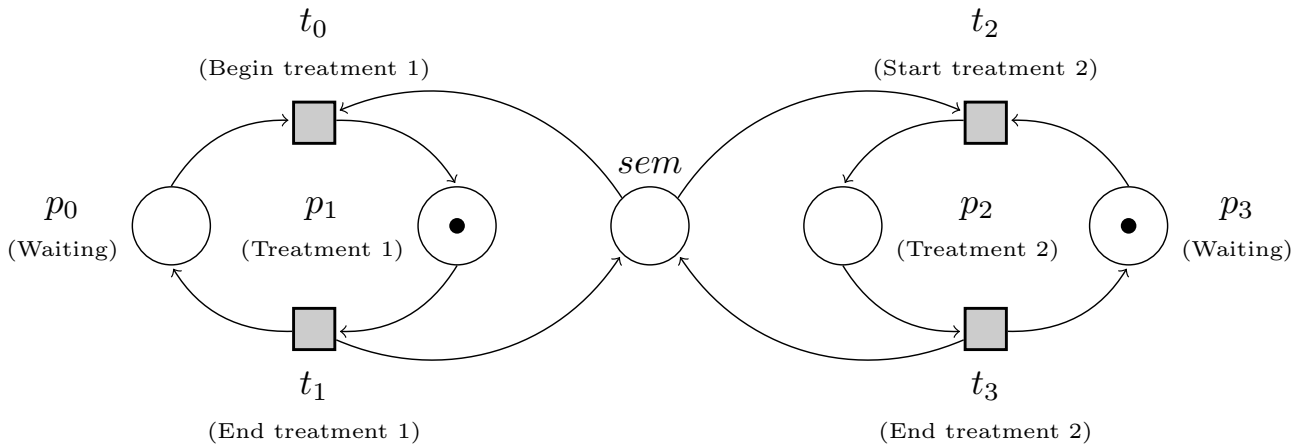


FIGURE 2.2: The PN of Figure 2.1 after the firing of transition t_0.

In Figure 2.2, the tokens in the input places of t_0, i.e. places p_0 and *sem* have been consumed, and one token has been produced in the output place p_1. The current marking indicates that the task "Treatment 1" is being performed (place p_1 is active).

In Figure 2.1, transition t_0 and t_2 are enabled at the same time. However, the *standard* semantics of PNs is such that only one transition can be fired in that case. Either t_0 consumes the token in place *sem* or t_2 does, but never both. Thus, the transition firing process in the standard PN semantics is an nondeterministic process. From the marking of Figure 2.1, two markings are reachable: the marking resulting of the firing of transition t_0 and the one resulting of the firing of transition t_2. Also, the transition firing process is asynchronous. As soon as a transition is enabled, the transition firing process can be triggered.

Extended Petri nets

The class of *extended* Petri nets introduces the inhibitor and test edges. As shown in Figure 2.3, test arc tips are black circles and inhibitor arc tips are white circles. Inhibitor and test edges are incoming edges, always coming from a place toward a transition.

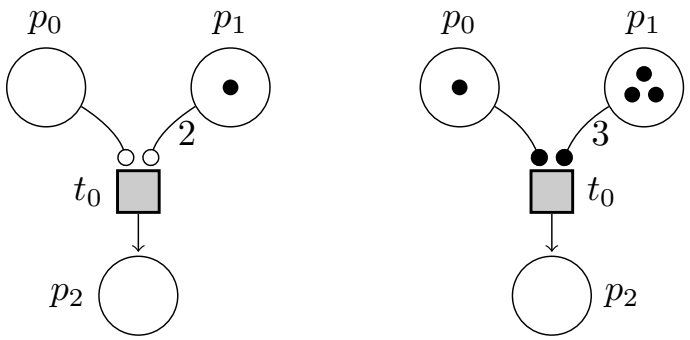


FIGURE 2.3: Two examples of extended Petri nets; on the left side, a PN with inhibitor arcs; on the right side, a PN with test arcs.

The particularity of the inhibitor and test edges is that they are not consuming tokens in input places after the firing of a transition. Indeed, they are just testing the number of tokens in incoming places to determine if the transition is enabled. Inhibitor arcs ensure that the number of tokens in input places is strictly lower than their weights; test arcs ensure that the number of tokens in incoming places is equal or greater than their weights. Therefore, on the left side of Figure 2.3, transition t_0 is sensitized because there is strictly less than one token in place p_0 and strictly less than two tokens in place p_1. On the right side of Figure 2.3, transition t_0 is sensitized because there is at least one token in place p_0 and three tokens in place p_1.

Interpreted Petri nets

Interpreted Petri nets (IPN) [20] describe the interaction between a system and its outside environment. This class of PN introduces three new concepts:

- Continuous actions, associated to the places of a Petri net. Actions associated to a place p are activated as long as p is marked. For instance, when modelling a controller with a IPN, actions can correspond to the setting of a electric signal controlling some actuator (e.g, maintaining a LED on).
- Functions (or discrete actions), associated to the transitions of a Petri net. When a transition t is fired, all functions associated to t are executed. Functions can be any kind of discrete operations –variable incrementation, for instance– manipulating both internal variable and external signal values.
- Conditions, associated to the transitions of a Petri net. Conditions are boolean expressions receiving their values from the environment of the PN. In an IPN, a transition is firable only if all its associated conditions are `true` (or `false` in the case where an inverse condition is associated).

Conditions represent inputs from the environment; actions and functions describe the influence of the modelled system on its environment. There is an interaction between actions, functions and conditions. The execution of actions and functions modify the environment of the system, and the modification of the environment will be measured through the value of

conditions. However, in an IPN, the interaction between actions, functions and conditions is not described as it would require to model the dynamics of the environment. Figure 2.4 illustrates the use of actions, functions and conditions in an interpreted Petri net.

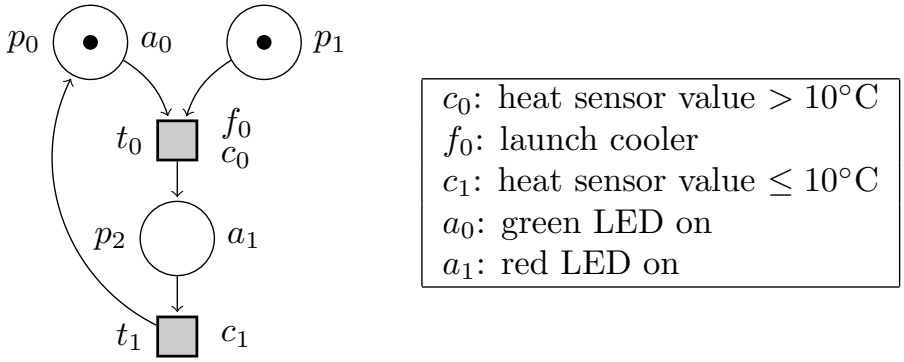


FIGURE 2.4: An example of Interpreted Petri net; on the left side, the interpreted Petri net; on the right side, examples of tests associated to conditions and operations associated to actions and functions.

In Figure 2.4, the action a_0 is activated as place p_0 is marked by one token. Also, function f_0 will be executed at the firing of t_0, that is if condition c_0 is `true` and t_0 is sensitized. On the right side of Figure 2.4, we associate a semantics to conditions, actions and functions in terms of concrete tests or operations. However, when considering the semantics of IPNs, what is of interest to us is the value of conditions and the execution state of actions and functions. We are not interested in interpreting the Boolean expressions associated to conditions but only to retrieve their value. Likewise, we are only interested in the fact that a given action/function is activated/executed but not in what is its effect on the environment.

Time Petri nets

In a time Petri net (TPN), time intervals are associated to transitions. The goal is to constrain the firing of a transition to a certain time window. As shown in Figure 2.5, time intervals are of the form $[a, b]$, where $a \in \mathbb{N}^*$ and $b \in \mathbb{N}^* \sqcup \{\infty\}$. Other definitions of time intervals exist for TPNs (e.g. with real numbers), but here we will only consider the latter definition. In Figure 2.5, time counters are represented in red between diamond brackets. The current value of time counters is part of the state of the TPN, along with its current marking, whereas time intervals are part of the static structure of the TPN.

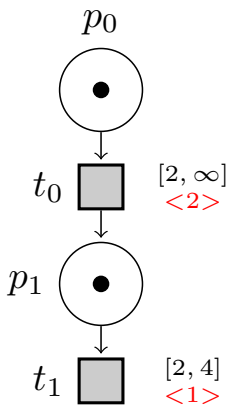


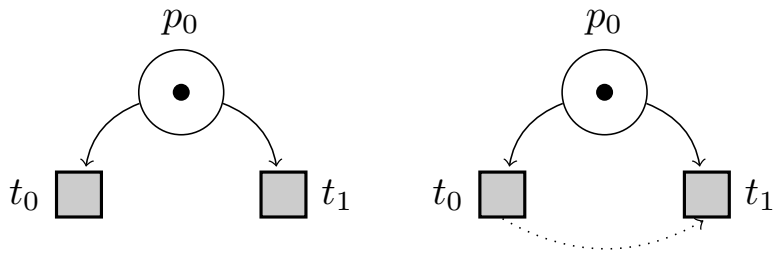
FIGURE 2.5: An example of time Petri net. The value of time counters appears in red.

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

When a transition associated with a time interval is fired or disabled, a reset order is sent to the transition to set its time counter to zero. The value of reset orders (Boolean values) is also a part of the TPN state. In time Petri nets, a transition is firable only if its time counter value is within its time interval. For instance, in Figure 2.5, only transition t_0 is firable. Moreover, there are several possible firing policies for TPNs. Here, we will only consider the *imperative* firing policy: as soon as a time counter reaches the lower bound of a time interval, the associated transition must be fired.

Petri nets with priorities

Two transitions are in structural conflict if they have a common input place connected through a *basic* arc (i.e. neither inhibitor nor test). When two transitions in structural conflict are firable at the same time and if the firing of one of the transitions disables the other, then, the conflict becomes *effective*. In a Petri net with priorities, it is possible to specify a firing priority in the case where the conflict between two transitions becomes effective. In that case, the transition with the highest firing priority will always be fired first. Figure 2.6 illustrates the application of a priority relation to solve the effective conflict between two transitions.



(a) Without priorities. (b) With priorities.

FIGURE 2.6: An example of transitions in structural and effective conflict. In subfigure (b), the dotted arrow represents the priority relation between t_0 and $t1$. The transition with the highest firing priority is at the source of the arrow; here, transition t_0.

2.1.2 Particularities of SITPNs

Here, we will informally present the specificities of the Petri nets describing the internal behavior of the HILECOP high-level model components. These Petri nets are called: Synchronously executed, extended, generalized, Interpreted, Time Petri Nets with priorities or SITPNs. SITPNs are a combination of multiple classes of PNs, namely: extended PNs, generalized PNs, interpreted PNs, time PNs and PNs with priorities. These classes were presented in the above section. We will now talk about another aspect of SITPNs that constitutes the originality of the formalism compared to the standard PN semantics: its synchronous execution.

The class of interpreted Petri nets increases the expressiveness of the HILECOP high-level models. However, to ensure the safe execution of functions after the synthesis of the designed circuit on an FPGA card, the whole system must be synchronized with a clock signal [35]. As a consequence, a clock signal also regulates the evolution of SITPNS (i.e. it is a part of their semantics). The evolution of a SITPN is *synchronized* with two clock events: the rising edge and the falling edge of the signal. Figure 2.7 depicts the process of state evolution, following the clock signal.

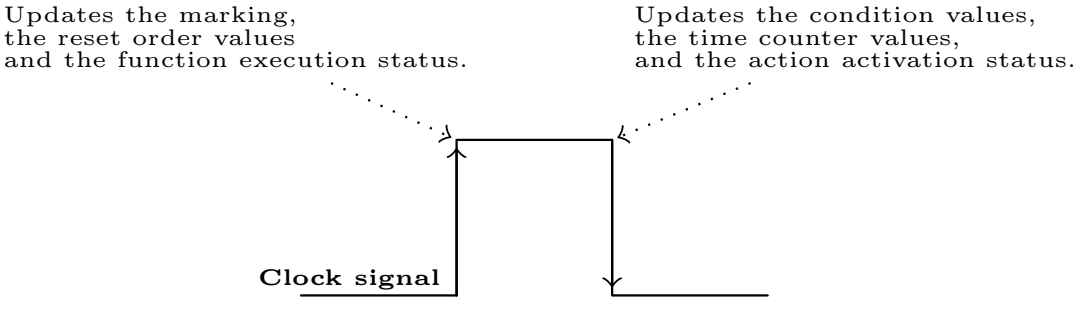


FIGURE 2.7: Evolution of an SITPN synchronized with a clock signal.

Considering the different classes of PNs that define SITPNs, the state of a SITPN is characterized by its marking, the value of time counters, the reset orders assigned to time counters,

the execution/activation status of actions/functions (Boolean values), and the value of conditions (also Boolean). As shown in figure 2.7, the state evolution process of a SITPN is divided into two steps. The rising edge of the clock signal triggers the marking update, which is the consequence of transition firing; all transitions that have been fired or disabled by the firing process receive reset orders; all functions associated with fired transitions are executed. Then, on the falling edge of the clock signal, the environment provides a new value to each condition. Also, the falling edge triggers the evolution of the time counter values; values are incremented, reset, or stalling (see the following remark on locked time counters). Finally, all actions associated with marked places are activated. Figure 2.8 gives an example of the evolution of the state of a given SITPN through one clock cycle. The aim of this figure and the explanation that follows is to give some hints to the reader about the semantics of SITPNs before giving its formal definition in Section 2.2.4.

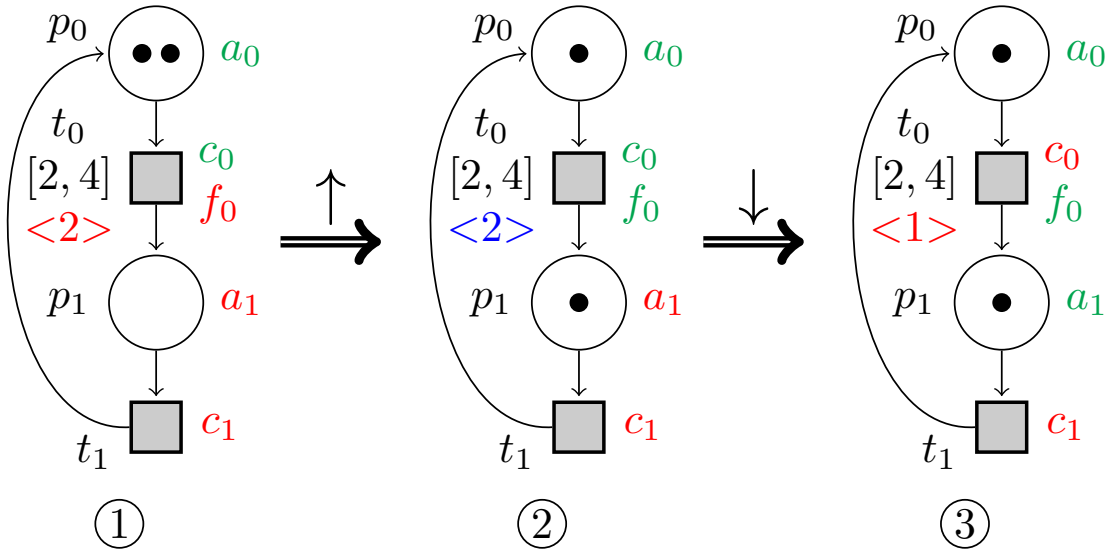


FIGURE 2.8: Evolution of a SITPN over one clock cycle. Conditions appear in green when their value is `true` and in red otherwise; actions and functions appear in green when they are activated/executed and in red otherwise; time counters appear in red and between diamond brackets; time counters appear in blue when they receive reset orders.

From Step 1 to Step 2, the rising edge of the clock signal triggers the SITPN state evolution. Here, transition t_0 is fired. At Step 1, transition t_0 gathers all the necessary conditions to trigger the firing process, namely: t_0 is enabled by the current marking, condition c_0 is `true` (appears in green), the value of t_0's time counter is within the time interval. As a consequence, one token is consumed in place p_0 and one token is produced in place p_1. Also, function f_0 is executed at Step 2 (appears in green) and a reset order is sent to the time counter of t_0 (appears in blue). From Step 2 to Step 3, the falling edge updates the action activation status: a_0 stays activated as place p_0 is still marked; a_1 becomes newly activated as p_1 is marked. The value of time counters are updated: t_0's time counter is set to zero as the transition previously received a reset order. However, as t_0 is still enabled by the new marking, its time counter is incremented. Thus, the

resulting time counter value at Step 3 is of one (i.e. result of reset plus increment). Also, the environment provides a new value to each condition. As a consequence, condition c_0 takes the value `false` and condition c_1 keeps the same value.

A remark on priorities

The semantics of synchronous execution is that all transitions are fired at the same time. In Figure 2.9, transitions t_0 and t_1 are both sensitized by place p_0, and consequently are both fired at the same time. The system acts as if two tokens were available in place p_0, one for the firing of t_0 and another for the firing of t_1.

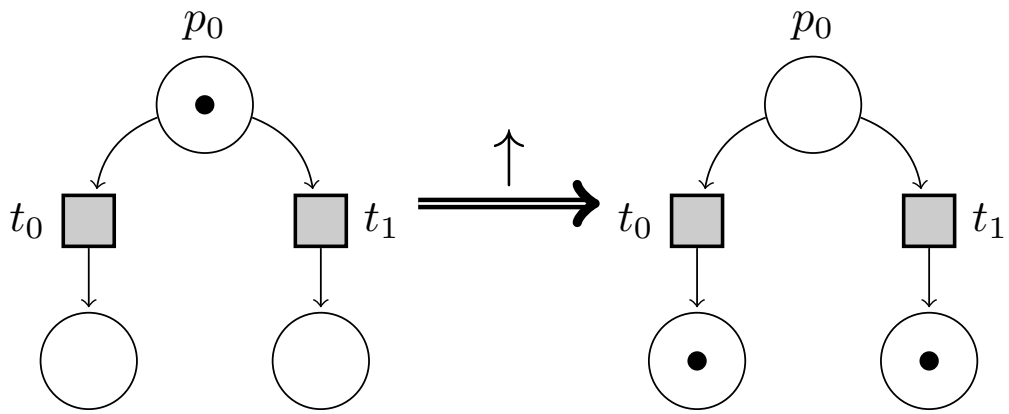


FIGURE 2.9: Double consumption of one token in a SITPN. On the left side, the current marking before the firing of t_0 and t_1; on the right side, the marking resulting of the firing of t_0 and t_1. The arrow indicates the occurrence of a rising edge that triggers the firing process.

In the context of a SITPN, a branching like the one of Figure 2.8, normally interpreted as a disjunctive branching, takes the semantics of a conjunctive branching when no priority are prescribed between the conflicting transitions. To avoid the phenomenon of "double consumption" of tokens, we enforce the resolution of any structural conflict by means of mutual exclusion or through the application of priorities. This policy about the resolution of structural conflicts is part of the definition of a well-defined SITPN presented in Section 2.2.6. The property of well-definition is mandatory to produce safe models of digital systems.

When a structural conflict between transitions is solved with priorities, the firing process follows a slightly different mechanism. As illustrated in Figure 2.10, to determine which transitions of t_0, t_1 and t_2 must be fired, a *residual marking* is computed by following the priority order. For each transition of the group t_0, t_1 and t_2, the residual marking represents the remnant of tokens in p_0 after the firing of transitions with a higher firing priority. Thus, in the semantics of SITPNs, we add an extra condition to the firing of a transition: to be fired, a transition must be enabled by the current marking, must have all its conditions valuated to `true`, must have its time counter within its time interval *and* must be enabled by the residual marking. The computation of the residual marking only involves the consumption phase of the firing process; tokens are withdrawn from places, but none are generated.

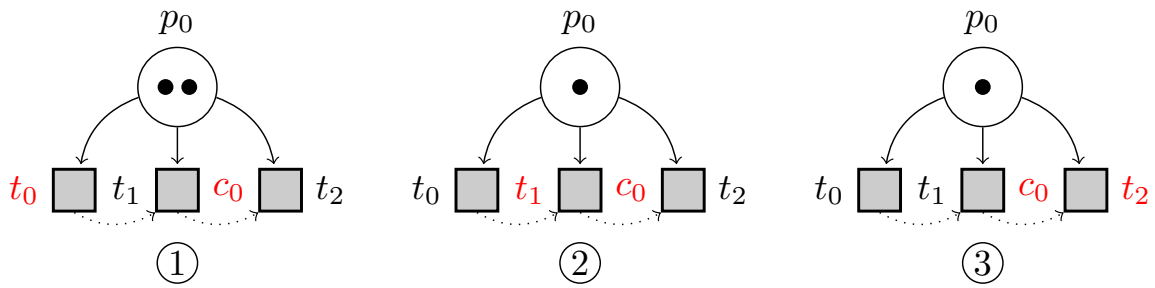


FIGURE 2.10: Computation of the residual marking for a group of conflicting transitions. At ① (resp. ② and ③), the residual marking for transition t_0 (resp. t_1 and t_2). Condition c_0 is in red to indicate that its currentl value is `false`.

In Figure 2.10, the residual marking for t_0 corresponds to the marking obtained after the firing of all transitions with a higher priority. As t_0 is the transition with the highest firing priority, the residual marking for t_0 is equal to the current marking. Transition t_0 gathers all the conditions to be firable and is enabled by the residual marking; thus, t_0 will be fired on the next rising edge. The residual marking for t_1 is the marking obtained after the firing of t_0, i.e. the only transition with a higher priority. As illustrated at ②, t_1 is enabled by the residual marking. However, t_1 does not gather all the conditions to be firable as the value of condition c_0 is `false`. Thus, t_1 will not be fired on the newt rising edge. The residual marking for t_2 is obtained after the firing of t_0 only. Even though transition t_1 has a higher firing priority than t_2, t_1 is not a member of the set of fired transitions. Thus, t_1 is not taken into account in the computation of the residual marking for t_2. The residual marking at ③ enables transition t_2, and as t_2 gathers all the conditions to be firable, then t_2 will be fired on the next rising edge.

Locked time counters

SITPNs inherit the properties of time PNs and interpreted PNs. The phenomenon of *locked* time counters is a consequence of this inheritance. As illustrated in Figure 2.11, the value of a time counter can overreach the upper bound of its associated time interval. This situation can only arise if a condition hinders the firing of a given transition while the considered transition is still enabled by the marking. As a consequence, the time counter will be incremented at every clock cycle until the upper bound of the time interval is overreached. Then, at this point, the time counter is said to be *locked* and its value will no more evolve.

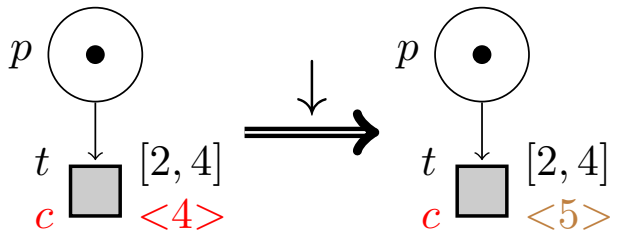


FIGURE 2.11: An example of locked time counter. Condition c is equal to `false` and thus appears in red.

In Figure 2.11, condition c is valuated to `false` before the falling edge of the clock signal. Thus, transition t can not be fired but is still enabled by the marking. On the next falling edge, the time counter of transition t is incremented and overreaches the upper bound of interval $[2, 4]$ and thus becomes locked.

2.2 Formalization of the SITPN structure and semantics

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

2.2.1 SITPN structure

The structure of SITPNs is formally defined as follows:

Definition 1 (SITPN). *A synchronously executed, extended, generalized, interpreted, and time Petri net with priorities is a tuple* $<P, T, pre, post, M_0, \succ, \mathcal{A}, \mathcal{C}, \mathcal{F}, \mathbb{A}, \mathbb{C}, \mathbb{F}, I_s>$, *where we have:*

1. $P = \{p_0, \ldots, p_n\}$, *a finite set of places.*
2. $T = \{t_0, \ldots, t_m\}$, *a finite set of transitions.*
3. $pre \in P \to T \nrightarrow (\mathbb{N}^* \times \{\texttt{basic}, \texttt{inhib}, \texttt{test}\})$, *the function associating a weight to place-transition edges.*
4. $post \in T \to P \nrightarrow \mathbb{N}^*$, *the function associating a weight to transition-place edges.*
5. $M_0 \in P \to \mathbb{N}$, *the initial marking of the SITPN.*
6. $\succ \subseteq (T \times T)$, *the priority relation which is a partial order over the set of transitions.*
7. $\mathcal{A} = \{a_0, \ldots, a_i\}$, *a finite set of continuous actions.*
8. $\mathcal{F} = \{f_0, \ldots, f_k\}$, *a finite set of functions (discrete actions).*

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

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

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

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

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

2.2.2 SITPN State

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

Definition 2 (SITPN State)**.** *For a given* $sitpn \in SITPN$, *let* $S(sitpn)$ *be the set of possible states of* $sitpn$. *An SITPN state* $s \in S(sitpn)$ *is a tuple* $<M, I, reset_t, ex, cond>$, *where:*

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

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

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

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

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

In Items 2 and 3 of Definition 2, *time* transitions refer to transitions with a time interval, i.e. the transitions belonging to the domain of I_s.

2.2.3 Preliminary definitions and fired transitions

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

Notation 1 (Relations between markings)**.** *For all relation* $\mathcal{R}$ *existing between two marking functions* M *and* M', *the expression* $\mathcal{R}(M, M')$ *is a notation for* $\forall p \in P,\ \mathcal{R}(M(p), M'(p))$. *For instance,* $M' = M - \sum_{t_i \in Pr(t)} pre(t_i)$ *is a notation for* $\forall p \in P,\ M'(p) = M(p) - \sum_{t_i \in Pr(t)} pre(p, t_i)$.

Notation 2 (Sum expressions and arc types)**.** *Many times in this document, we need to express the number of tokens coming in or out of places, after the firing of a certain subset of transitions. To do so, we use two kinds of sum expression:*

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

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

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

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

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

   Here, we add the weight of the arc from t to p only if there exists such an arc; we add 0 to the sum otherwise.

Therefore, in the remainder of the document, we will use the conciser notation $\sum_{t \in T'} pre(p,t)$ to denote an output token sum, and $\sum_{t \in T'} post(t,p)$ to denote an input token sum.

We give the formal definition of the sensitization (see Section 2.1.1 for an informal definition) of a transition by a given marking as follows:

Definition 3 (Sensitization)**.** *A transition $t \in T$ is said to be sensitized, or enabled, by a marking M, which is noted $t \in Sens(M)$, if $\forall p \in P, \omega \in \mathbb{N}^*$, $\big(pre(p,t) = (\omega, \texttt{basic}) \vee pre(p,t) = (\omega, \texttt{test})\big) \Rightarrow M(p) \geq \omega$, and $pre(p,t) = (\omega, \texttt{inhib}) \Rightarrow M(p) < \omega$.*

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

Definition 4 (Firability)**.** *A transition $t \in T$ is said to be firable at a state $s = <M, I, reset_t, ex, cond>$, which is noted $t \in Firable(s)$, if $t \in Sens(M)$, and $t \notin T_i$ or $I(t) \in I_s(t)$, and $\forall c \in \mathcal{C}, \mathbf{C}(t,c) = 1 \Rightarrow cond(c) = 1$ and $\mathbf{C}(t,c) = -1 \Rightarrow cond(c) = 0$.*

As explain in Section 2.1.2, the firability conditions are not sufficient for a transition to be fired. A transition must also be enabled by the residual marking to go through the firing process. Definition 5 gives the formal definition of a fired transition at a given SITPN state:

Definition 5 (Fired)**.** *A transition $t \in T$ is said to be fired at the SITPN state $s = <M, I, reset_t, ex, cond>$, which is noted $t \in Fired(s)$, if $t \in Firable(s)$ and $t \in Sens\big(M - \sum_{t_i \in Pr(t)} pre(t_i)\big)$, where $Pr(t) = \{t_i \mid t_i \succ t \wedge t_i \in Fired(s)\}$.*

One can notice that the definition of the set of fired transitions is recursive. Indeed, to compute the residual marking necessary to the definition of a fired transition, the Pr set must be defined. For a given transition t, the Pr set of t represents all the transitions with a higher firing priority than t that are also fired transitions; hence the recursive definition.

In Definition 5, the marking $M - \sum_{t_i \in Pr(t)} pre(t_i)$ formally qualifies the residual marking for a given transition t and at a given SITPN state s.

2.2.4 SITPN Semantics

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

Definition 6 (SITPN Semantics)**.** *The semantics of a given $sitpn \in SITPN$ is the transition system $<L, E_c, \rightarrow>$ where:*

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

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

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

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

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

    (1) $cond'$ is the function giving the (Boolean) values of conditions that are extracted from the environment E_c at the clock count τ, i.e.:
    $\forall c \in \mathcal{C},\ cond'(c) = E_c(\tau, c)$.

    (2) All the actions associated with at least one marked place in the marking M are activated, i.e.:
    $\forall a \in \mathcal{A},\ ex'(a) = \sum_{p \in marked(M)} \mathbb{A}(p, a)$ where $marked(M) = \{p' \in P \mid M(p') > 0\}$.

    (3) All the time transitions that are sensitized by the marking M and received the order to reset their time intervals, have their time counter reset and incremented, i.e.:
    $\forall t \in T_i,\ t \in Sens(M) \wedge reset_t(t) = \texttt{true} \Rightarrow I'(t) = 1$.

    (4) All the time transitions that are sensitized by the marking M, and did not receive a reset order, increment their time counters if time counters are still active, i.e.:
    $\forall t \in T_i,\ t \in Sens(M) \wedge reset_t(t) = \texttt{false} \wedge (I(t) \leq upper(I_s(t)) \vee upper(I_s(t)) = \infty) \Rightarrow I'(t) = I(t) + 1$.

(5) *All the time transitions verifying the same conditions as above, but with locked counters, keep having locked counters (values are stalling), i.e.:*
$\forall t \in T_i,\ t \in Sens(M) \wedge reset_t(t) = \texttt{false} \wedge I(t) > upper(I_s(t)) \wedge upper(I_s(t)) \neq \infty \Rightarrow I'(t) = I(t)$.

(6) *All the time transitions disabled by the marking M have their time counters set to zero, i.e.:*
$\forall t \in T_i,\ t \notin Sens(M) \Rightarrow I'(t) = 0$.

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

(7) *M′ is the new marking resulting from the firing of all the transitions contained in Fired(s), i.e.:*
$\forall p \in P,\ M'(p) = M(p) - \sum_{t \in Fired(s)} pre(p,t) + \sum_{t \in Fired(s)} post(t,p)$.

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

$$\begin{aligned}
&\forall t \in T_i, t \in Fired(s) \\
&\quad \vee \big(\exists p \in P, \omega \in \mathbb{N}^*,\ pre(p,t) = (\omega, \texttt{basic}) \vee pre(p,t) = (\omega, \texttt{test}) \\
&\qquad \wedge \sum_{t_i \in Fired(s)} pre(p,t_i) > 0 \\
&\qquad \wedge M(p) - \sum_{t_i \in Fired(s)} pre(p,t_i) < \omega\big) \Rightarrow reset'_t(t) = \texttt{true}, \\
&\quad \text{and } reset'_t(t) = \texttt{false} \text{ otherwise.}
\end{aligned}$$

(9) *All functions associated with at least one fired transition are executed, i.e:*
$\forall f \in \mathcal{F},\ ex'(f) = \sum_{t \in Fired(s)} \mathbb{F}(t,f)$.

Rules (1) to (6) describe the SITPN state evolution at the falling edge of the clock signal. Rules (1) and (2) pertain to the update of condition values and to the update of the activation status of actions. Rules (3), (4), (5) and (6) focus on the update of time counter values. In Rule (4) of the SITPN semantics, the *active* time counters refer to the time counters that have not yet overreached the upper bound of their associated time interval. Of course, a time counter is always active when the upper bound is infinite. In Rule (5), the *locked* time counters refer to the time counters that have overreached the upper bound of their associated time interval. Of course, time counters can never be locked in the presence of an infinite upper bound. In Rules (4) and (5), for a given time interval i, $upper(i)$ denotes the upper bound of the time interval, and $lower(i)$ denotes the lower bound of the time interval.

Rules (7) to (9) describe the SITPN state evolution at the rising edge of the clock signal. Rule (7) corresponds to the marking update. The computation of the new marking uses the set of fired transitions at state s, i.e. $Fired(s)$. Rule (9) pertains to the update of the function execution status. Rule (8) computes the reset orders for time transitions. There are two cases where a time transition receives the order to reset its time counter. First, if the transition is one of the fired transitions at state s, then its time counter must be reset on the next falling edge.

Second, if the transition is disabled in a *transient* manner, then its time counter must also be reset. Figure 2.12 illustrates the case of a transition disabled by the *transient* marking, i.e. the marking obtained after the consumption phase of the firing process.

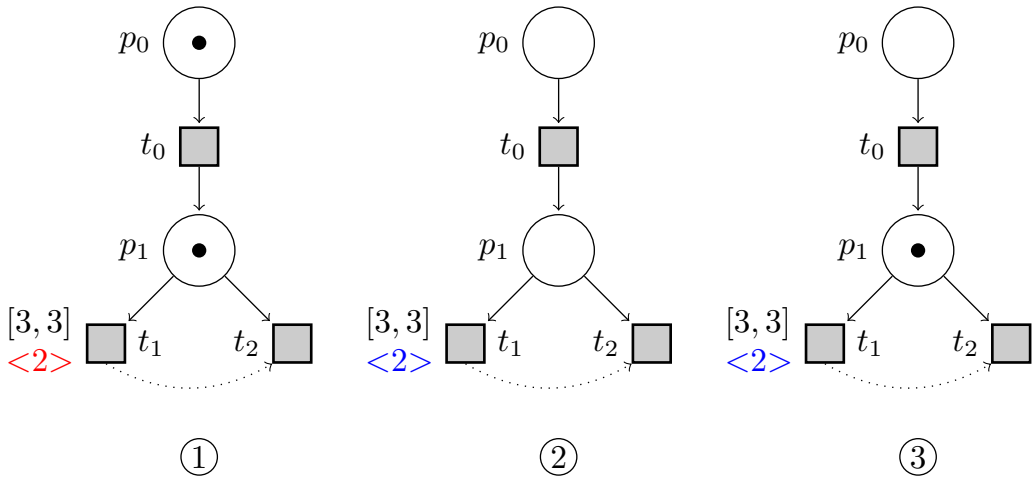


FIGURE 2.12: An example of transition that receives a reset order after being disabled by the transient marking. At ①, the marking before the firing of transitions t_0 and t_2; at ②, the transient marking; at ③, the marking at the end of the firing process.

In Figure 2.12, the situation at ① describes the state of the SITPN before a rising edge. Judging by the current SITPN state at ①, transition t_0 and t_2 will be fired on the next clock event. Situation ② depicts the marking obtained after the consumption phase of the firing process, i.e. the so-called *transient* marking. Situation ③ corresponds to the marking at the end of the firing process. At ③, transition t_1 is enabled by the marking. However, at ②, the transient marking disables t_1 and thus t_1 must receive a reset order (represented by a blue time counter).

2.2.5 SITPN Execution

As a part of the SITPN semantics, we define here the SITPN execution and SITPN full execution relations. These relations bind a given SITPN to the execution trace, i.e. a time-ordered list of states, that it produces when executed over a given number of clock cycles. These definitions are additional elements corresponding to our own contribution to the formalization of the SITPN semantics.

Definition 7 (SITPN execution). *For a given $sitpn \in SITPN$, a starting state $s \in S(sitpn)$, a clock cycle count $\tau \in \mathbb{N}$, and an environment $E_c \in \mathbb{N} \to \mathcal{C} \to \mathbb{B}$, $sitpn$ yields the execution trace θ from starting state s, written $E_c, \tau \vdash sitpn, s \to \theta$, by following the two rules below:*

EXECUTIONEND

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

EXECUTIONLOOP

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

Definition 8 (SITPN full execution). *For a given $sitpn \in SITPN$, a clock cycle count $\tau \in \mathbb{N}$, and an environment $E_c \in \mathbb{N} \rightarrow \mathcal{C} \rightarrow \mathbb{B}$, $sitpn$ yields the execution trace θ starting from its initial state $s_0 \in S(sitpn)$ (as defined in Definition 6), written $E_c, \tau \vdash sitpn \rightarrow \theta$, by following the two rules below:*

FULLEXEC0

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

FULLEXECCONS

$$\frac{E_c, \tau \vdash s_0 \xrightarrow{\uparrow_0} s_0 \quad E_c, \tau \vdash s_0 \xrightarrow{\downarrow} s \quad E_c, \tau - 1 \vdash sitpn, s \rightarrow \theta_s}{E_c, \tau \vdash sitpn \xrightarrow{full} (s_0 :: s_0 :: s :: \theta_s)} \; \tau > 0$$

The FULLEXECCONS rule of the SITPN full execution relation (Definition 8) appeals to the SITPN execution relation (Definition 7). However, the definition of the SITPN full execution relation is necessary because the first cycle of execution, starting from the initial state s_0, is particular. As shown in the premises of Rule FULLEXECCONS, the first rising edge is idle. We consider that no transitions are fired during the first rising edge. Thus, the first rising edge does not change the initial state s_0, and we denote the particular first rising edge with the sign $\uparrow_0$ over the SITPN transition relation.

2.2.6 Well-definition of a SITPN

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

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

Definition 9 (Conflict). *For a given $sitpn \in SITPN$, two transitions $t, t' \in T$ are in conflict if there exists a place $p \in P$ such that $p \in input(t) \cap input(t')$ and there exist $\omega, \omega' \in \mathbb{N}^*$ such that $pre(p, t) = (\omega, \texttt{basic})$ and $pre(p, t') = (\omega', \texttt{basic})$.*

A conflict group qualifies a finite set of transitions that are all in conflict with each other through the same place. In Figure 2.13, the set $\{t_0, t_1\}$ is a conflict group. The formal definition of a conflict group is as follows:

Definition 10 (Conflict Group). *For a given $sitpn \in SITPN$, $T_c \subseteq T$ is a conflict group if there exists a place p such that $\forall t \in T, \left(\exists \omega \in \mathbb{N}^*,\ pre(p,t) = (\omega, \texttt{basic})\right) \Leftrightarrow t \in T_c$.*

Contrary to the statement made in [35, p. 67], we no more consider the notion of conflict as being transitive. To illustrate this, Figure 2.13 shows two conflict groups: $\{t_0, t_1\}$ and $\{t_1, t_2\}$. In a well-defined *SITPN* (see Section 2.2.6), all conflicts in a conflict group must be dealt with, i.e. for all pair of transitions in the group the conflict must be solved. However, we no more consider transitions t_0 and t_2 as in conflict. We argue that even when no conflict resolution technique is applied between transitions in the same situation as t_0 and t_2, the execution of the *SITPN* can neither result in the double-consumption of a token, nor in the case where a transition is not elected to be fired even though it ought to be. Therefore, we no more consider the construction of merged conflict group (i.e, conflict groups must be merged into one if their intersection is not empty; e.g, $\{t_0, t_1, t_2\}$ in Figure 2.13) as being necessary.

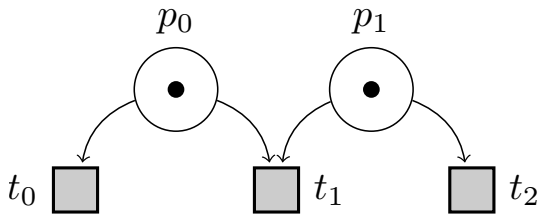


FIGURE 2.13: An example of two separate conflict groups, namely: $\{t_0, t_1\}$ and $\{t_1, t_2\}$.

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

Definition 11 (Mutual exclusion with complementary conditions). *Given two conflicting transitions t_0 and t_1, t_0 and t_1 are in mutual exclusion with complementary conditions if there exists $c \in \mathcal{C}$ such that $(\mathbb{C}(t_0, c) = 1 \wedge \mathbb{C}(t_1, c) = -1)$ or $(\mathbb{C}(t_0, c) = -1 \wedge \mathbb{C}(t_1, c) = 1)$.*

Definition 12 (Mutual exclusion with an inhibitor arc). *Given two conflicting transitions t_0 and t_1, t_0 and t_1 are in mutual exclusion with an inhibitor arc if there exists $p \in P$ and $\omega \in \mathbb{N}^*$ such that $(pre(p, t_0) = (\omega, \texttt{basic}) \vee pre(p, t_0) = (\omega, \texttt{test})) \wedge pre(p, t_1) = (\omega, \texttt{inhib})$ or $(pre(p, t_1) = (\omega, \texttt{basic}) \vee pre(p, t_1) = (\omega, \texttt{test})) \wedge pre(p, t_0) = (\omega, \texttt{inhib})$.*

Figure 2.14 illustrates the two means of mutual exclusion that can be applied to solve a conflict between two transitions.

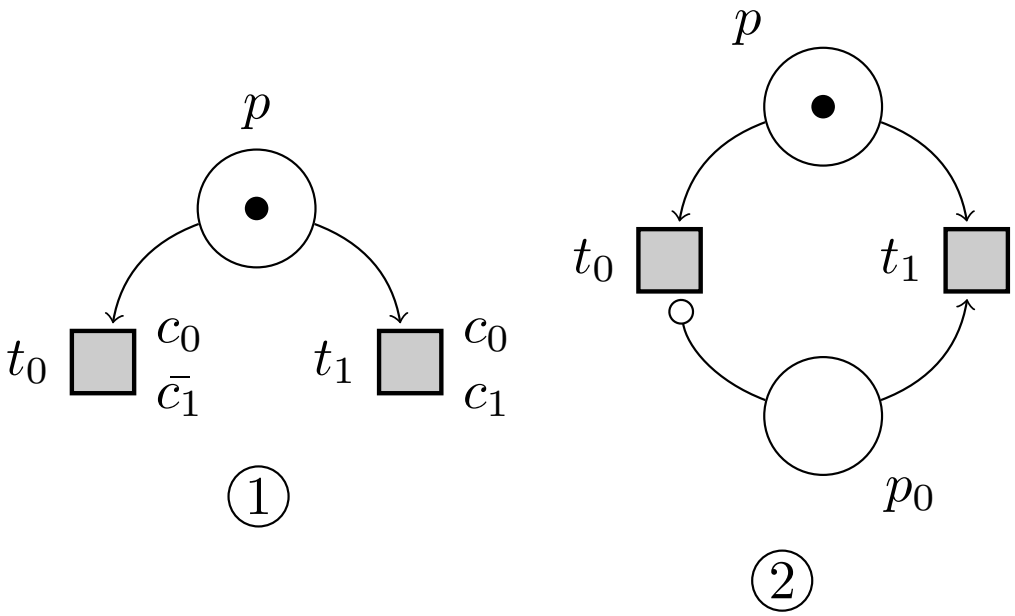


FIGURE 2.14: Examples of conflicting transitions in mutual exclusion. At ①, an example of mutual exclusion with complementary conditions; at ②, an example of mutual exclusion with an inhibitor arc.

In Figure 2.14, in situation ①, condition c_1 is associated to t_1 and the complementary condition is associated to t_0 thus creating the mutual exclusion. In situation ②, the arc (p_0, t_0) and (p_0, t_1) ensure the mutual exclusion between transitions t_0 and t_1. Note that in the structure of mutual exlcusion with an inhibitor arc, the weight of the inhibitor arc and of the basic or test arc must be the same; otherwise, the mutual exclusion is not effective.

A given $sitpn \in SITPN$ is well-defined if it enforces some properties needed on the HILECOP source models before the transformation into VHDL. If the properties, layed out in Definition 13, are not ensured, they will lead to compile-time errors during the transformation of the SITPN into a VHDL design.

Definition 13 (Well-defined SITPN). *A given $sitpn \in SITPN$ is well-defined if:*

- $T \neq \emptyset$, *the set of transitions must not be empty.*
- $P \neq \emptyset$, *the set of places must not be empty.*
- *There is no isolated place, i.e, a place that has neither input nor output transitions:* $\nexists p \in P,\ input(p) = \emptyset \wedge output(p) = \emptyset$, *where* $input(p)$ *(resp.* $output(p)$*) denotes the set of input (resp. output) transitions of* p.
- *There is no isolated transition, i.e, a transition that has neither input nor output places:* $\nexists t \in T,\ input(t) = \emptyset \wedge output(t) = \emptyset$, *where* $input(t)$ *(resp.* $output(t)$*) denotes the set of input (resp. output) places of* t.

- *For all conflict group as defined in Definition 10, either all conflicts (i.e. for all pair of transitions in the conflict group) are solved by one of the mean of mutual exclusion, or, the priority relation is a strict total order over the transitions of the conflict group.*

2.3 Implementation of the SITPN structure and semantics

In this section, we present our mechanization of the SITPN structure and semantics with the Coq proof assistant. The source code is available to the reader at the address https://github.com/viampietro/ver-hilecop. More precisey, the implementation of the SITPN structure and semantics is to be found under the `sitpn/dp` directory. We have made a first implementation of SITPNs without the use of dependent types. For this first version, we have also implemented a SITPN interpret (a so-called *token player*) and proved that the interpret was sound and complete w.r.t the SITPN semantics. This first implementation of the SITPNs and the formal proof of soundness and completeness are available at https://github.com/viampietro/sitpns.

2.3.1 Implementation of the SITPN and the SITPN state structure

Listing 2.1 presents the implementation of the SITPN structure as a Coq record type. The implementation is almost similar to the formal definition of the SITPN structure given in Definition 1.

```
1  Record Sitpn := BuildSitpn {
2
3    places : list nat;
4    transitions : list nat;
5    P := { p | (fun p0 ⇒ In p0 places) p };
6    T := { t | (fun t0 ⇒ In t0 transitions) t };
7
8    pre : P → T → option (ArcT * ℕ*);
9    post : T → P → option ℕ*;
10   M₀ : P → nat;
11   Iₛ : T → option TimeInterval;
12
13   conditions : list nat;
14   actions : list nat;
15   functions : list nat;
16   𝒞 := { c | (fun c0 ⇒ In c0 conditions) c };
17   𝒜 := { a | (fun a0 ⇒ In a0 actions) a };
18   ℱ := { f | (fun f0 ⇒ In f0 functions) f };
19
20   ℂ : T → 𝒞 → MOneZeroOne;
21   𝔸 : P → 𝒜 → bool;
22   𝔽 : T → ℱ → bool;
23
24   pr : T → T → Prop;
25
26 }.
```

LISTING 2.1: Implementation of the SITPN structure in Coq.

We use lists of natural numbers, i.e. `list nat` in Coq, to define the finite sets of places (Line 3), transitions (Line 4), actions (Line 14), conditions (Line 13) and functions (Line 15) in the `Sitpn` record. We want to use these finite sets in the signature of functions appearing in the structure (e.g use the finite set of places P in the signature of the initial marking $M_0 \in P \to \mathbb{N}$). To do son we leverage the Coq `sig` type to define subsets of elements verifying a certain property. Thus, we define the finite set P as the subset of natural numbers that are members of the `places` list (Line 5). We use the `In` relation defined in the Coq standard library to express the membership of a natural number regarding the elements of `places` list. Also, the `ArcT` type (Line 8) implements the set $\{\texttt{inhib}, \texttt{test}, \texttt{basic}\}$; the `TimeInterval` type (Line 11) implements the set $\mathbb{I}^+$ of time intervals, and the `MOneZeroOne` type (Line 20) implements the set $\{0, 1, -1\}$. The priority relation is implemented by the `pr` function (Line 24) taking two transitions in parameter and projecting to the type of logical propositions, i.e. the `Prop` type.

Listing 2.2 presents the implementation of the SITPN state structure as a Coq record type.

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

LISTING 2.2: Implementation of the SITPN state structure in Coq.

The `SitpnState` type definition depends on a given SITPN passed as a parameter; it is an example of dependent type. Projection functions are automatically generated to access the attributes of a record at the declaration of a type with the `Record` keyword. Thus, in Listing 2.2, we can refer to the set of places of `sitpn` with the term `P sitpn`. The term T_i `sitpn` denotes the set of time transitions of `sitpn`. The set of time transitions for a given SITPN is declared as a `sig` type qualifying to the subset of transitions with an associated time interval.

2.3.2 Implementation of the SITPN semantics

Here, we present our implementation of the SITPN semantics. In Listing 2.3, we give an excerpt of the implementation of the SITPN state transition relation, i.e. the core of the SITPN semantics.

```
1  Inductive SitpnStateTransition
2    (sitpn : Sitpn) (E_c : nat → C sitpn → bool) (τ : nat) (s s' : SitpnState sitpn) :
3    Clk → Prop :=
4  | SitpnStateTransition_falling :
5
```

```
6       (* Rule (2) *)
7       (forall a marked sum,
8          Sig_in_List (P sitpn) (fun p ⇒ M s p > 0) marked →
9          BSum (fun p ⇒ 𝔸 p a) marked sum →
10         ex s' (inl a) = sum) →
11
12      (* Rules (3), (4), (5) and (6) *)
13      (forall (t : Ti sitpn), ∼Sens (M s) t → I s' t = 0) →
14      (forall (t : Ti sitpn), Sens (M s) t → reset s t = true → I s' t = 1) →
15      (forall (t : Ti sitpn),
16          Sens (M s) t →
17          reset s t = false →
18          (TcLeUpper s t ∨ upper t = i+) → I s' t = S (I s t)) →
19      (forall (t : Ti sitpn),
20          Sens (M s) t →
21          reset s t = false →
22          (upper t <> i+ ∧ TcGtUpper s t) → I s' t = S (I s t)) →
23
24      (** Conclusion *)
25      SitpnStateTransition E_c τ s s' ↓
26
27  | SitpnStateTransition_rising:
28
29      (** Rule (7) *)
30      (forall fired, IsNewMarking s fired (M s')) →
31
32      (* Rule (9) *)
33      (forall f fired sum,
34         IsFiredList s fired →
35         BSum (fun t ⇒ 𝔽 t f) fired sum →
36         ex s' (inr f) = sum) →
37
38      (* Conclusion *)
39      SitpnStateTransition E_c τ s s' ↑.
```

LISTING 2.3: Excerpt of the implementation of the SITPN state transition relation in Coq.

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

In the construction case `SitpnStateTransition_falling`, we give the implementation of Rules (2), (3), (4), (5) and (6) defined in the SITPN semantics. The sum term of Rule (2), i.e. $\sum_{p \in marked(M)} \mathbb{A}(p,a)$, is implemented by Lines 8 and 9. At Line 8, the `Sig_in_List` predicate states that all the inhabitant of the `P sitpn` type (i.e. the places of `sitpn`) that verifies the

property `(fun p => M s p > 0)` (i.e. the marking of a place is greater than zero at state `s`) are members of the `marked` list. Because we can not iterate over the elements of a given `sig` type, we use the `Sig_in_List` relation to convert a `sig` type into a list. Lists are iterable by definition. At Line 9, the `BSum` relation states that `sum` is the Boolean sum obtained by applying the function `(fun p => 𝔸 p a)` to the elements of the `marked` list. Rules (3), (4), (5) and (6) are almost similar in their implementation to the description of Definition 6. The Coq term `Sens (M s) t` implements the term $t \in Sens(M)$. Due to the particular nature of the upper bound of a time interval, i.e. defined over the set $\mathbb{N}^* \sqcup \{\infty\}$, the test that the current time counter of a given transition t is less than or equal to the upper bound is implemented by a separate predicate `TcLeUpper`. Similarly, the `TcGtUpper` predicate implements the inverse test.

In the construction case `SitpnStateTransition_rising`, we give the implementation of Rules (7) and (9) defined in the SITPN semantics. In the implementation of Rule (7), the `IsNewMarking` predicate hides away the expression:

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

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

2.4 Conclusion

The class of SITPNs is a particular class of PNs used to model the behavior of components in the HILECOP high-level models. The synchronous evolution of SITPNs constitutes the originality of the model compared to the standard PNs semantics. In this chapter, we gave an informal and formal presentation of SITPNs and their execution semantics. Two previous Ph.D. theses contributed, for the most part, to the formalization of the SITPN structure and semantics. However, we helped to simplify the semantics of SITPNs. We passed from 14 rules in the definition of the SITPN semantics given in [35] to 9 rules in our current definition of semantics. Also, we completed some rules when they happened to be insufficient to prove the theorem of behavior preservation. Finally, we defined the execution relations for the SITPN semantics and formalized the well-definition property for the SITPN structure.

Our other contribution was to implement the SITPN structure and semantics with the Coq proof assistant. There are two implementations: one with and one without dependent types. For the version without dependent types, we implemented a SITPN interpret or token player. We also proved a soundness and completeness theorem between the interpret and the formalized SITPN semantics. The first implementation of the SITPNs in Coq represents 5000 lines of specification and 7000 lines of proof. The second implementation leverages dependent types. This implementation is more concise and closer to the formal definition given within the set

theory. We chose this implementation to mechanize the proof of the behavior preservation theorem (see Chapter 4).

Chapter 3

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

The main purpose of this chapter is to present the target language of the HILECOP transformation, i.e. the VHDL language. The formalization and the implementation of the VHDL language syntax and semantics is mandatory to reason about the programs generated by the HILECOP model-to-text transformation. Thus, we want the reader to clearly understand the structure and the semantics of the language to be able to fully grab the proof of semantic preservation presented in Chapter 4. Specifically, we present here the $\mathcal{H}$-VHDL language, a synthesizable subset of the VHDL language. This subset permits to encode the programs generated by the HILECOP transformation. We devise a formal semantics for $\mathcal{H}$-VHDL which is a simplification of the simulation semantics of the VHDL language. The formalization of the $\mathcal{H}$-VHDL semantics and its implementation is one contribution of this thesis to the many formalization of the VHDL language found in the literature. The chapter is structured as follows. In Section 3.1, we give an informal presentation of the VHDL language syntax and semantics. In Section 3.2, we present the state of the art pertaining to the formalization of the VHDL language semantics. In Section 3.3, 3.4, 3.5 and 3.6, we give the full formalization of the $\mathcal{H}$-VHDL language, a subset of the VHDL language. Section 3.7 illustrates the formal $\mathcal{H}$-VHDL semantics with an example. Finally, Section 3.8 outlines the implementation of the $\mathcal{H}$-VHDL syntax and semantics with the Coq proof assistant.

As explained in Chapter 1, the HILECOP transformation generates a VHDL design implementing an input SITPN model. To do so, the transformation generates and connects the component instances of two previously defined VHDL designs: the `place` design, i.e. a VHDL implementation of a SITPN place, and the `transition` design, i.e. a VHDL implementation of a SITPN transition. These designs were defined by the INRIA CAMIN team at the creation of the HILECOP methodology. In the following sections, we will be using excerpts of the definition of the `place` and `transition` designs to illustrate the content of VHDL programs and the rules of the VHDL language semantics. The reader will find the source code of the `place` and `transition` designs in concrete and abstract syntax in Appendices A and B.

3.1 Presentation of the VHDL language

The intent here is to give an overview of the VHDL language, its purpose, its main syntactal constructs, and an informal description of its semantics as presented in the Language Reference Manual (LRM) [32]. The VHDL language offers a lot of possibility in terms of hardware (and even software) description. Here, we are not trying to be exhaustive in our presentation of the language. We will only maintain our description of the VHDL concepts in the scope that is of interest to us. The readers that are interested in learning more about the VHDL language can refer to [32], [2] and [46].

3.1.1 Main concepts

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

A top-level VHDL program is called a *design*. A VHDL design is composed of two descriptive parts. The first part is called the entity and describes the interfaces of a circuit, namely: the input and output ports, and the generic constants. Listing 3.1 is an excerpt of the `transition` design's entity that defines the generic constants, the input and output port interfaces of the design. The generic clause of the entity holds the declaration of the generic constants. The purpose of generic constants is either to represent some dimensions of the design (e.g. the size of ports, internal signals...) or to represent constant values used throughout the design. In Listing 3.1, one can see that the `conditions_number` generic constant gives a dimension to the type of the `input_conditions` input port, which is an array of Boolean values with indexes ranging from 0 to `conditions_number-1` (that is the meaning of `std_logic_vector (conditions_number-1 downto 0)`. The port clause holds the declaration of input and output ports of the design. The `in` keyword indicates the declaration of an input port and the `out` indicates the declaration of an output port.

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

```
19  end transition;
```

LISTING 3.1: The entity part of the transition design in concrete VHDL syntax.

Figure 3.1 is a visual representation of the interfaces of the `transition` design.

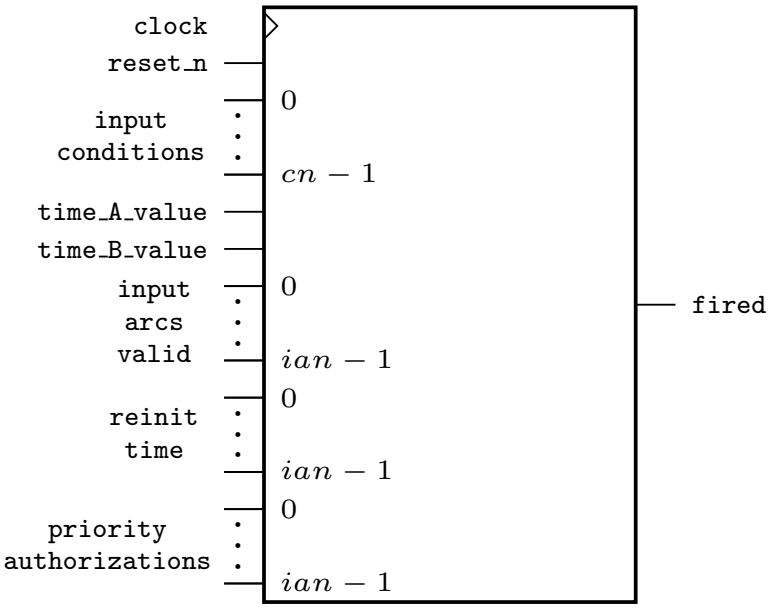


FIGURE 3.1: A representation of the `transition` design entity. On the left side, the input port interface of the transition design; *cn* stands for *conditions_number* and *ian* stands for *input_arcs_number*, i.e. two of the generic constants declared in the generic clause of the `transition` design entity; the numbers at the right of the input pins represent the pin indexes. On the right side, the output port interface of the `transition` design.

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

Behavior specification with processes

The first way to specify the behavior of a design is through the description of processes. Processes are concurrent statements that describes the wiring or the operations performed on the signals of a given design. These operations are described by sequential statements in the body of processes. A process declares a sensitivy list that corresponds to the signals read in the process statement body; also, it possibly declares internal variables. Listing 3.2 gives an excerpt of the `transition` design architecture containing the declarative part of the architecture (i.e. the declaration of internal signals) and three of the processes describing the `transition` design

behavior, namely: the `condition_evaluation` process, the `firable` process and the `fired_evaluation` process. In Listing 3.2, Lines 2 to 8 correspond to the declaration of the internal signals of the `transition` design. Line 11 starts the declaration of the `condition_evaluation` process. The sensitivity list of the `condition_evaluation` process holds one signal, the `input_conditions` input port. The value of the `input_conditions` input port is read is the process body. Then, as a design rule, it must be declared in the sensitivity list. The process defines a local variable `v_internal_condition` at Line 12. At Line 13, the `begin` keyword starts the declaration of the process body, i.e. the block of sequential statements performing operations on the signals of the `transition` design.

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

```
38
39  end transition_architecture;
```

LISTING 3.2: An excerpt of the architecture part of the transition design in concrete VHDL syntax.

In the statement body of a process, the designer can use control flow statements common to most of the generic programming languages (if statement, for loops...), and also statements that are specific to the VHDL language. The most representative statement, and the one of interest to us, is the *signal assignment* statement. The signal assignment statement relate a given signal identifier to a source expression. For instance, at Line 20 of Listing 3.2, the signal assignment statement, represented with the $\Leftarrow$ operator, assigns the value of the internal variable `v_internal_condition` to the `s_condition_evaluation` signal. The `v_internal_variable` that itself holds the Boolean product between the subelements of the `input_conditions` input port performed in the `for` loop of Lines 16 to 18.

When considering a VHDL design in the point of view of hardware synthesis, a signal assignment statement specifies a wiring between a target signal identifier and other source signals. Figure 3.2 gives a synthesis-oriented view of the processes described in Listing 3.2.

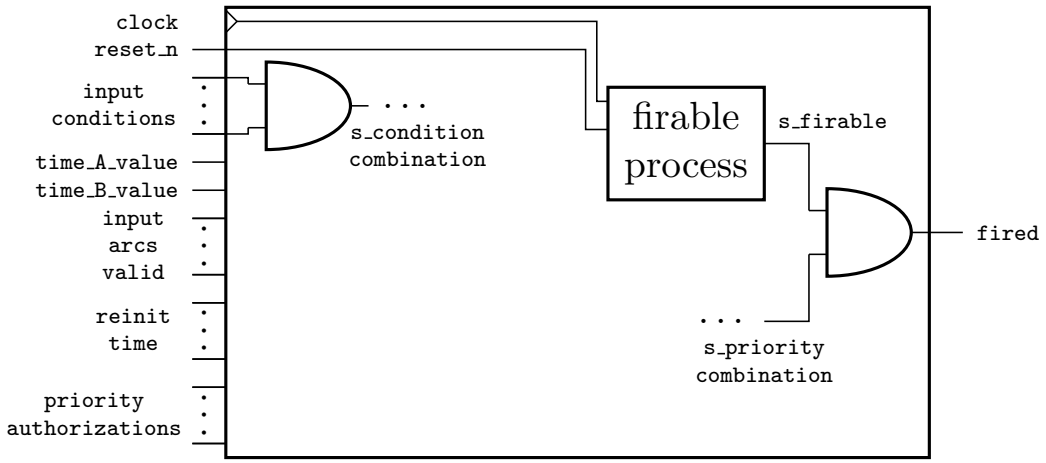


FIGURE 3.2: A representation of a part of the transition design architecture comprising three processes. On the left side, the `condition_evaluation` process connecting the `input_conditions` input port to the `s_condition_combination` internal signal; the `firable` process in the middle; on the right side, the `fired_evaluation` process connecting the `s_firable` and the `s_priority_combination` signals to the `fired` output port.

In Figure 3.2, the `condition_evaluation` process is represented as an `and` port performing the product over the elements of the `input_conditions` input port. The `fired_evaluation` process is a simple `and` gate connecting the `fired` output port to the `s_firable` and `s_priority_combination` internal signals. The `fired_evaluation` and the `condition_evaluation` processes are combinational processes. They describe the value of an output signal based on the value of input signals. For instance, the value of the `s_condition_combination` signal is a function of the value of the `input_conditions` input port, i.e `s_condition_combination`=

f(`input_conditions`). This equation always holds, and we refer to it as a combinational equation.

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

To complete the presentation of the statements to be found in the body of processes, the VHDL language is also equipped with `timing` constructs, i.e. statements that explicitly specify an amount of time in a given time unit. The signal assignement statement possibly specifies a time clause indicating when the assignment must be performed. For instance, the signal assignment statement specifying that the value of signal `b` must be assigned to signal `a` in 3 milliseconds takes the form: $a \Leftarrow b$ `in 3 ms`. When no time clause is specified for a signal assignment statement, we talk about a δ-delay signal assignment, i.e. the application of the signal assignment is related to some δ interval corresponding the time of propagation through a wire. When a time clause is specified, we talk about an unit-delay signal assignment. δ-delay signal assignments are synthetizable, meaning they have an equivalent implementation on a physical device, whereas, unit-delay signal assignments can not be synthetized. Unit-delay signal assignments do not appear neither in the VHDL designs generated by HILECOP transformation nor in the declaration of the place and transition designs. We are only mentioning their existence because they play a part in the simulation algorithm described in Section 3.1.2.

Behavior specification with design instances

The second way to specify the behavior of a design is to use other designs, or rather instances of other designs, as subcomponents. In that case, the design is said to be composite as it embeds instances of other designs in its own behavior. Also, a design at the highest level of embedding, i.e. that is not instantiated as a part of another design's behavior, is called a *top-level* design. The design instantiation, or component instantiation, statement permits to instantiate a design in an embedding architecture. When instantiating a design with a design instantiation statement, the designer provides the component instance with an identifier. Then, the design instance must be dimensioned; this is performed through a generic map that associates the generic constants of the design being instantiated to a static value. Finally, the designer specifies how the component instance is connected to the other elements of the architecture. A port map associates the input ports and output ports of the component instance to expressions or to the signals of the embedding architecture. Listing 3.3 shows an example of instantiation of the HILECOP's `transition` design. This instance is involved in the definition of the behavior of an embedding design called `toplevel`.

```
1 architecture toplevel_architecture of toplevel is
2 begin
```

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

LISTING 3.3: An example of design instantiation statement in the architecture of the `toplevel` design. Here, the design being instantiated is the transition design.

In Listing 3.3, the `transition` component instance (TCI) has the identifier id_t. Following the `entity` keyword is the name of the design being instantiated; here, the `transition` design. Then, the generic map associates the generic constants of the transition design (i.e. the left side of the arrow, also called the *formal* part) to static values (i.e. the right side of the arrow called the *actual* part). This permits the dimensiniong the component instance. For example, remember that the `input_arcs_number` generic constant value determines the number of elements in the composite input ports `input_arcs_valid`, `priority_authorizations` and `reinit_time` (cf. Figure 3.1). The port map associates the input ports of the `transition` design to expressions. For instance, the `time_A_value` input port is connected to the constant value 0, and the `input_conditions` input port is connected to the internal signal id_0 at index 0. The port map also associates the output ports with signal identifiers. Contrary to the association of input ports, output ports can not be associated to expressions. An output port association describes a direct wiring. In the port map described in Listing 3.3, the association `fired` $\Rightarrow id_3$ expresses that the `fired` output port is connected to the signal id_3, where signal id_3 is defined in the embedding design. Figure 3.3 illustrates the `transition` component instance id_t and the wiring of its input and output port interfaces inside the `toplevel` design.

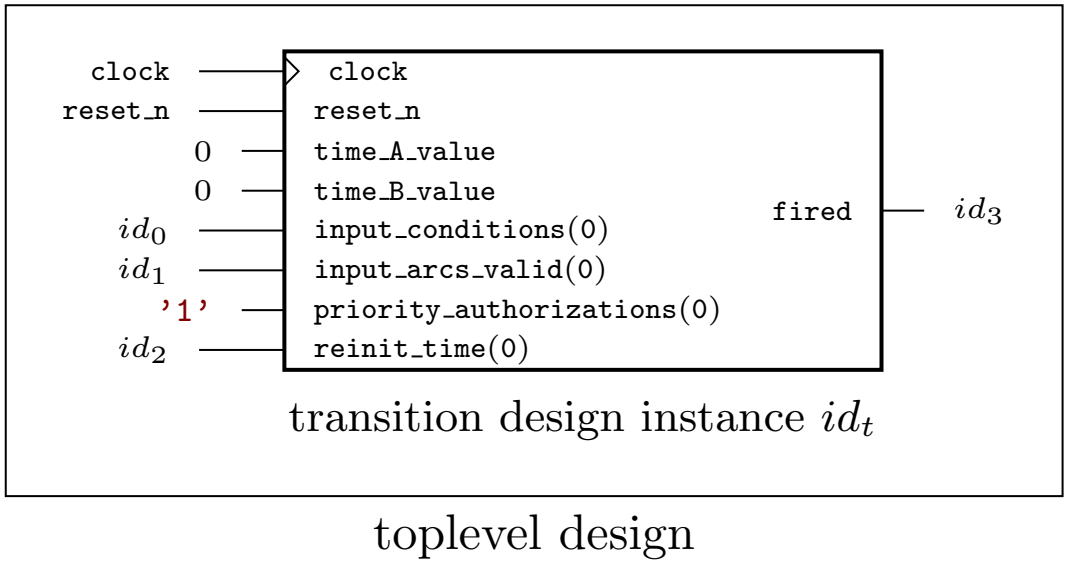


FIGURE 3.3: Visual representation of a design instantiation statement. Here, the figure represents the transition design instance described in Listing 3.3.

3.1.2 Informal semantics of the VHDL language

There are two approaches to the description of circuits with the VHDL language. The first aims at the simulation of the described circuits, and the second aims at the synthesis of described circuits on physical supports. These two approaches arise from the practice and the use of the VHDL language by electronicians. Even though, in practice, there are two ways to consider a VHDL design, i.e. a synthesis-oriented way and a simulation-oriented way, the LRM does not define a synthesis-oriented semantics for the VHDL language. A synthesis-oriented semantics gives an interpretation to a design by describing an equivalent in a lower level formalism, closer to the physical circuit. For instance, the Verilog language gives a synthesis-oriented semantics to its programs by defining an equivalent RTL level description [33]. The LRM gives an informal semantics to VHDL designs through the definition of a simulation algorithm [32, p.167]. The purpose of simulation is to compute the evolution of the values of signals during a certain time interval. Through the simulation process, the designer is able to control the behavior of the modeled circuits and to detect flaws in the evolution of the signal values.

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

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

After the elaboration, the top-level design, or rather its corresponding execution model, is ready to be simulated. Two entities are involved in the simulation: the *sea* of processes obtained after the elaboration of the top-level design's behavior, and a *kernel* process. The kernel process orchestrates the simulation; it handles the time of the simulation, i.e. it holds a variable describing the current time of the simulation, and controls the execution of processes. Figure 3.4, which is an excerpt from [13], describes the structure of the VHDL simulation algorithm.

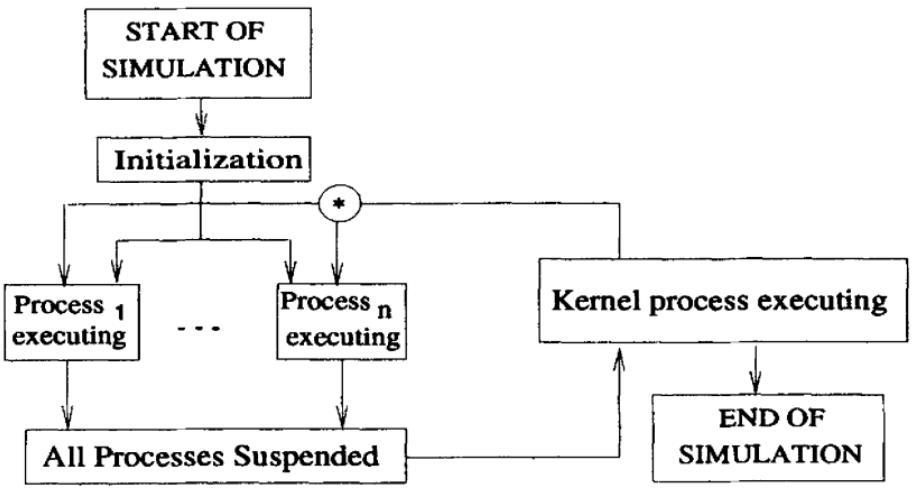


FIGURE 3.4: The VHDL simulation loop. Excerpt from [13].

The simulation starts with an initialization phase. During the initialization phase all processes are run exactly once. Then, the simulation cycles are structured as follows. All processes execute their statement body concurrently. New transactions are posted in the drivers of signals for every interpreted signal assignment statement. The execution goes on until all processes have executed their statement body and then have reached a suspension state. When, all processes are suspended, the kernel process takes over. Figure 3.5 shows the activity diagram associated with the kernel process.

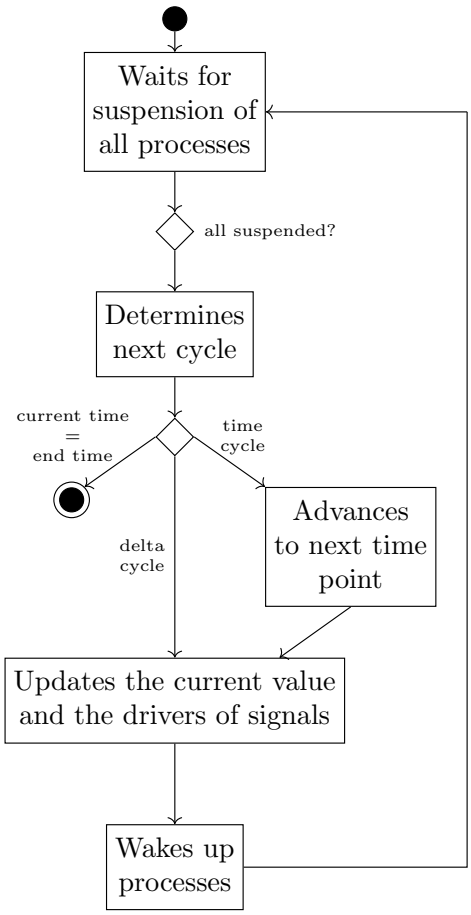


FIGURE 3.5: The activity diagram of the kernel process. Square boxes represent activities, diamond nodes are decision nodes. The black circle at the top represents the starting point of the activities; the other black circle in the middle of the diagram represents the end of all activities.

As shown in Figure 3.5, after the suspension of all processes, the kernel process will then determine the kind of simulation cycle that will be performed next. There are two kinds of cycles: delta cycles or time cycles. If the value of a signal changes at the current time point, i.e. its driver holds a transaction at the current time point with a new value, then a delta cycle must be performed. Then, the simulation time does not change. The kernel process updates the current value of signals and their drivers, and wakes up the processes sensitive to the signals that obtained new values. The repetition of multiple delta cycles corresponds to the stabilization of signal values, i.e. the propagation of values through the wires, that takes effect in a negligeable δ time. If all signal values are stable at the current time point, then a time cycle must be performed. The kernel process looks up the drivers for the next time point where the value of a given signal will change. Then, the kernel process advances the simulation time to this next time point before updating the signal values and resuming the execution of processes. The simulation goes on like this, alternating between delta and time cycles, until the current time value reaches the time specified for the end of the simulation.

3.2 Choosing a formal semantics for VHDL

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

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

3.2.1 Specifying our needs: HILECOP and VHDL

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

The need for coverage

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

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

- component instantiation statements with generic constant and port mapping
- entity's generic constant clauses (declaration of generic constants in a design entity)

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

Regarding component instantiation statements, the VHDL LRM describes a way to transform these statements into equivalent process statements and block constructs [32, p. 141] which are a part of the elaboration of the design. However, we want to preserve the hierarchical structure provided by the component instantiation statements arguing that it will be easier to compare the state of a given SITPN model with a VHDL design state with an explicit hierarchical structure. Indeed, there exists a mapping between places and transitions of an SITPN and their mirror (generated by the transformation) place and transition component instances (PCIs and TCIs). This one-to-one correspondence might turn out to be handy to perform the proof of behavior preservation. Obviously, the semantics must cover the evaluation of process statements which are the core concurrent statements of VHDL programs.

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

The need for a synchronous execution

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

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

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

Other considerations

Considering the kind of proof that needs to be established, we would rather consider an operational semantics for VHDL. The reason is that, in the CompCert project [36], which is one of our major inpsiration source, the whole C compiler toolchain is verified by reasoning over the operational semantics of the source and target languages. A last consideration pertains to whether or not the VHDL semantics must explicitly handle errors. As the SITPN semantics does not include the production of error values, the handling of errors by the VHDL semantics is not a mandatory aspect.

Qualifying criterions

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

- *Synchronism*. We distinguish three levels for this criterion:
  - Synchronism is not expressible in the considered VHDL semantics.
  - Synchronism is expressible in the considered VHDL semantics. Synchronism is expressible if time-steps are handle in the semantics, at least to be able to represent clock events.
  - Synchronism is explicit, i.e. the simulation loop is built around the occurrences of clock events.

  We will foster the semantics that explicitly formalize a synchronized execution of a VHDL design.
- *Component instantiation*. Either the semantics handle the component instantiation statement in its simulation rules, or component instantiation statements must be transformed in order to be executed. We will foster the semantics that handle component instantiation statements without transformation.
- *Elaboration*. This criterion expresses the ability of the semantics to handle constrained types, i.e. arrays and natural ranges, and generic constant clauses that are both dealt with during *elaboration* phase. Either the semantics handle these constructs or it does not. Of course, we will foster the first kind of semantics.

3.2.2 Looking for an existing formal semantics

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

Denotational semantics

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

In [27], the authors give a denotational semantics to the VHDL language within the Focus [21] framework, a method for the development of distributed systems. Signal values and their evolution through time are represented as streams of values. Statements are denoted as stream-processing functions. Processes are stream-processing functions that takes input signal streams (signals of the sensitivity list) and yields transaction traces (i.e, waveforms) over output signals (i.e, signal that are written by the process). Transaction traces are merged together as the result of the concurrent execution of processes. The authors only consider 0-delay signal assignments in their semantics, stating that it is sufficient to "consider time at a logical level to model both synchronous and asynchronous designs". However, it necesitates some transformations on a design that has a synchronous execution to express it only with 0-delay signal assignments. Therefore, this semantics does not express synchronism of execution in an explicit manner. Moreover, the component instantiation statements are not dealt with, and no mention is made of the elaboration phase.

In [12], the authors give a denotational, yet relational, semantics for VHDL. A state of a VHDL design is represented by a function binding signals to values; a worldline is a time-ordered list of states. Statements (including processes) are denoted in the semantics by a relation that binds an input couple, composed of a time point and a worldline, to an output couple of the same type. Multiple input and output couples possibly satisfy the relation denoting a particular statement; thus, the semantics is undeterministic. The semantics tries to abstract from the formalization of the simulation cycle as it is done in the LRM. The authors want to establish a semantics that is abstract enough to be able to compare all other works of formalization with the authors semantics. The authors also give an axiomatic semantics (i.e, in the Hoare logic style) which is proved to be sound and complete with the first denotational semantics. A Prolog [17] implementation of the axiomatic semantics is given. Regarding our needs, the semantics only deals with unit-delay signal assignments. However, this semantics enables the representation of a δ-delay signal assignment with a unit-delay signal assignment adorned with a "`after 0 ns`" time clause. The hierarchical structure of designs is not preserved, and, although expressible, the semantics does not explicitly express a synchronous simulation cycle.

The denotational semantics expressed in [45] uses interval temporal logic as an underlying model. Leveraging this underlying model, the authors are interested in proving some properties over VHDL designs to help compilers to optimize the code, for instance, by using rewrite rules proved to be valid against the model. Some of the proofs laid out by the authors are embedded in PVS [44]. The expression of the dynamic model uses many concepts described in the LRM, like drivers, port association, driving and effective values for signals. The semantics deals with both unit-delay and δ-delay signal assignments. The semantics works on fully-elaborated designs, therefore, it does not deal with component instantiation statements. Moreover, interval temporal logic is useful to reason on the VHDL designs in the presence of delays, however, it looses its interest for designs presenting only 0-delay assignments.

In [7], the author states that "denotational semantics is more adequate for mathematical reasoning". The author formalizes the VHDL semantics to prove the equivalence between VHDL programs (for instance, a specification and an implementation). What is of major interest regarding our needs is that the author has expressed a simulation cycle for synchronous designs. Therefore, a distinction is made between combinational and synchronous processes in the abstract syntax. Moreover, this work formalizes the elaboration part of a VHDL design former to the simulation; also, the elaboration keeps the hierarchical setting of the VHDL design, that is component instantiation statements are not replaced by processes. Due to the time abstraction, the semantics only deals with 0-delay. It is explained by the fact that the reference time-unit is the clock period (i.e, the only known time-step), and the advancing of time, happening during the simulation cycle as described in the LRM, is captured within the setting of the simulation cycle. Also, the semantics takes primary inputs into account (i.e, input ports of the top-level design); to preserve a synchronous behavior for the simulated design, the hypothesis is made that the values of the primary inputs are stable between two clock events.

Operational semantics

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

In [11], a formal description of a *functional* semantics for VHDL is laid out based on stream-processing functions. The semantics is expressed with the functional programming language Gofer [34], thus enabling the computation of execution traces, that is, the computation of the streams representing the values taken by signals over time. As in the former work of the same author [12], only unit-delay signal assignments are dealt with, however, this time the author describes a deterministic operational semantics. Regarding our needs, this work is neither interested in preserving the hierarchical structure of VHDL designs, and no mention is made regarding how a design is elaborated, nor in expressing an explicit synchronous simulation cycle.

In [13], the authors formalize the simulation loop of the LRM using Evolving Algebra machines (EA-machines). All important constructs of the VHDL language are represented as records; processes are represented as concurrent agents running pseudo-codes, and the simulation control flow is passed to and fro between the kernel process (i.e, the simulation orchestrator) and the rest of the processes that execute the design behavior. This semantics implements closely the simulation loop as described in the LRM. Therefore, it is very rich and deals with most of the VHDL constructs, including the two time paradigms of the language (i.e. δ time and unit time). Moreover, the semantics works on fully-elaborated designs, therefore, component instantiation statements are omitted. However, a synchronous execution is fully expressible even if not explicitly embedded in the expression of the simulation loop.

In [52], the author presents a natural semantics for VHDL. The simulation loop is expressed by inference rules, and the execution of processes is based on the events over signals of their corresponding sensitivity lists. The execution of statements computes transaction traces, that is, the projected waveforms for signals over the future of the simulation. The semantics deals both with unit and delta delays. Regarding our needs, this semantics covers does not entirely

cover the subset of VHDL we are interested in. Component instantiation statements are not dealt with. A synchronous execution is expressible within the semantics, although it would be hidden in the inference rule formalizing the generic simulation loop. Also, the semantics does not provided its simulation loop with a simulation horizon (a maximum number of simulation cycle to be computed). The simulation ends when signal values evolve no more. The question of the influence of the environment, measured through the values of the primary inputs of a design, is not discussed.

In [28], the author presents an operational semantics for VHDL in the small-step style. The semantics follows closely the simulation cycle described in the LRM; however it is very concise and clear. The covered VHDL subset comprises both unit and delta-delay signal assignments. There is a interesting discussion about the non-determinism of VHDL, since it is a concurrent programming language: it entails that non-determinism is only existent at the processes level, that is, internal sequential statement of processes can be executed in an undeterministic manner (refered to by the author as A actions, that is, *internal* actions). But at every delta or time step (refered to as δ and T actions) of the execution, the design state can be computed in a deterministic manner, since all processes have reached a suspension point at the end of their inner body. The author is interested in comparing the behaviors of two VHDL designs by proving that some relation of equivalence holds between the two. He describes two strategies to compare VHDL programs. The first one is bisimulation; it is based on the comparison of the sequence of actions (either A, δ or T actions) performed by the two programs. The second one is observational equivalence; it is based on the observation of the value of the output signals of two VHDL programs (the observees), that receive values in their input signals from another VHDL program (the observer). The observer stimulates the entries of the observees and reaches a success state based on its observations of the value of the outputs. Regarding our needs, this semantics permits the description of our synchronous simulation cycle. However, like most of the semantics presented here, the component instantiation statement is not supported as it stands, but it is rather transformed into the equivalent processes statements. Small-step semantics is not needed in our case because we are only interested in the values of signals at the delta and time steps (for us, time steps correspond to clock events). We are not interested in capturing the design states in the middle of the execution of a process body. We are more interested in "weak bisimulation", therefore forsaking the internal actions performed by a VHDL design. In [51], the authors extend the work of [28], especially by handling shared variables, in the presence of which a VHDL program can have a concrete underterministic behavior. The authors are also interested in the equivalence between two VHDL programs, and they are interested in determining a unique meaning property for VHDL programs. The unique meaning property states that the execution of a VHDL design in the presence of shared variables is unique. This work is interesting as it points out the fact that VHDL is not only subject to benign undeterminism. However, we are not interested in dealing with constructs so advanced as shared variables, therefore, this work is not really relevant to us.

Translational semantics

Another kind of semantics, called “translational”, formalizes the VHDL language semantics by translating a VHDL design into another formal model. Thus, the semantics of VHDL is

modeled by the translation and the formal semantics of the target model. The target model has the ability to model concurrency, which is one of the specificity of VHDL. Moreover, target models are chosen because of the tools that they provide for analysis, and thus, a translational semantics for VHDL is often related to model checking considerations.

In [48], the author expresses the formal semantics of VHDL by translating a VHDL design into a corresponding *flowgraph*. All VHDL constructs, ranging from sequential statements to concurrent processes, are expressed with individual flowgraphs that are then composed together through their interfaces. The simulation cycle of VHDL is also encoded by means of connected flow graphs: one for the "execution part" of the semantics, that is, all processes run until suspension, and one for the update part (i.e, the kernel process in the semantics of [13]). Flowgraphs come with a large amount of tools for analysis, and this translational semantics is involved in the setting of a framework to reason about VHDL programs using multiple technics (automatic theorem proving, model checking...). All these technics lean on the flowgraph formalism.

In [24], the author introduces a translational semantics for VHDL based on deterministic finite-state automatons. Again, the reason for using such automatons lies in the existence of many analysis tools. Moreover, forcing the generation of deterministic automatons improves the time execution of model-checking technics. The translation is performed on an elaborated VHDL design; a data space stores the values of signals and variables, and automatons represent the control-flow of VHDL statements. Each VHDL statement is associated to a specific automaton; sequence of statements are achieved by automaton composition. The simulation kernel is also represented by a specific automaton. Processes are composed together with respect to synchronization states, i.e. states that permit to pass the control from one process to another, therefore achieving determinism in the control flow of the overall automaton.

In [43], the author presents a translation from VHDL to Coloured Petri Nets (CPNs) thus giving a formal semantics to VHDL constructs. The author approach to VHDL semantics is a strict translation of the "event-based" VHDL simulator by means of Petri nets. The author translates VHDL execution models (sea of processes) into CPNs, and also translates the kernel process into a CPN. The kernel process has previously been expressed as a VHDL process so that the translation into CPN is similar to the translation of other processes. Signals are not represented in the subnets, instead, three shared variables depict the signal states: one variable for the driving, one for the effective and one for the current value of a given signal (see [32, p.167] for the details on the values associated with signals during the simulation). Colour domains of places in the subnets represent the different types of VHDL domains. Variables are represented by tokens. Values in drivers are represented by sequences of transactions (equivalent to waveforms); the author defines a set of functions that are convenient to handle sequences of transactions. Sequential statements are partitioned into two kinds: control flow (if, loop, case...) and notation (operations on signals and variables) nets. Processes subnets are made by the fusing of each sequential statements in the process body. There is a special *Resume* place that can be set by the kernel process to resume the activity of a process. Concurrency is not discussed here, as the Petri net models are inherently concurrent models. The kernel process is a broad CPN having some of its places interfaced with the process subnets. The decoloration of the Petri net enables the analysis of the model and the detection of dead-locks.

In [23], the author gives a formal semantics to VHDL by transforming a VHDL design into

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

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

Table 3.1 summarizes the analysis of the VHDL semantics encountered during our literature review. Table 3.1 compares the different VHDL semantics in relation to our qualifying criterions (see Section 3.2.1).

		Fuchs and Mendler [27]	Breuer et al. [12]	Pandey et al. [45]	Borrione and Salem [7]	Breuer et al. [11]	Börger et al. [13]	Van Tassel [52]	Goossens [28]	Reetz and Kropf [48]	Döhmen and Herrmann [24]	Olcoz [43]	Déharbe and Borrione [23]
Semantics Description	Kind	D	D, A	D	D	O	O	O	O	T	T	T	T
	Purpose	AR, ATP	AR	AR	AR	SS	SS	SS, ITP	SS, MC	ATP, MC, ITP	MC, ITP	MC	MC
Qualifying Criterions	Component Instantiation	T	T	T	N	T	T	T	T	T	T	T	N
	Synchronism	NE	NE	NE	Ex	E	E	E	E	E	E	E	NE
	Elaboration	×	×	×	✓	×	×	✓	×	×	×	×	✓
Extra. Informations.	Impl. Technology	Focus [21]	Prolog [17]	PVS [44]	?	Gofer [34]	?	HOL [31]	?	HOL [31]	?	?	?
	Particular Model or Data Types	Stream Processing	No	Interval Temporal Logic	No	Stream Processing	Evolving Algebra Machines	Natural Semantics (big-step)	Structural Semantics (small-step)	Flow Graphs	Finite-State Automatons	Colored Petri Nets	Abstract Machines and Decision Diagrams

TABLE 3.1: A comparative summary on VHDL formal semantics.

- Kind : D (Denotational) - A (Axiomatic) - O (Operational) - T (Translational).
- Purpose : AR (Abstract Reasoning) - ATP (Automatic Theorem Proving) - SS (Simulator Specification) - ITP (Interactive Theorem Proving) - MC (Model Checking).
- Component Instantiation : T (statement is *Transformed* into equivalent processes) - N (statement is *Natively* taken into account in the semantics).
- Synchronism : E (Expressible within the semantics) - NE (Not Expressible within the semantics) - Ex (Explicitly built in the semantics).

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

In Table 3.1, cells are colored in green when the cell's content foster the adoption of the semantics, in yellow when the content does not go towards the adoption of the semantics but is not disqualifying, and red when the content is a disqualifying criterion. With regards to the semantics adoption, cells are labelled in light grey when their content is neutral. Now comparing the entries of Table 3.1 with the expression of our needs, we can discard the semantics with a cell labelled in red, that is most of the denotational semantics; moreover, all translational semantics are let aside for the reasons cited before. The candidate semantics are the operational semantics, plus the denotational semantics by Borrione and Salem [7], the only semantics that formalizes an explicitly synchronous simulation cycle. The semantics that is the most likely to be adopted is the Borrione and Salem's semantics. However, we prefer an operational setting for our semantics because it is more fit to our task. To lower down the complexity of proofs, we really need a semantics that builds the synchronism into its simulation cycle, therefore putting aside all the intricacies of the full-blown VHDL simulation cycle. Moreover, the big-step style for an operational semantics is more relevant to us; as stated before, we are not interested in the intermediary states of computation that a small-step style semantics considers. Based on these observations, we have decided to formalize our own VHDL semantics inspired from the semantics of Borrione and Salem's [7] and Van Tassel's [52]. The following sections are dedicated to the presentation of the syntax and semantics of a subset of VHDL called $\mathcal{H}$-VHDL. $\mathcal{H}$-VHDL embeds the subset of VHDL that we are interested in when considering the VHDL designs generated by the HILECOP transformation.

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

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

3.3.1 Design declaration

Similarly to [52], we define the *design* construct in the $\mathcal{H}$-VHDL's abstract syntax which has no equivalent in the concrete syntax of VHDL.

In the above entry, id_e indicates the entity identifier and id_a the architecture identifier of the declared design. The gens entry corresponds to the generic clause, i.e. the declaration list

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

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

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

3.3.2 Concurrent statements

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

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

Process statement

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

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

Component instantiation statement

The VHDL LRM defines two kinds of component instantiation statement: the instantiation of a component instance [32, p.139] and the instantiation of a design entity [32, p.141]. The component instantiation statement used in the $\mathcal{H}$-VHDL abstract syntax corresponds to the instantiation of a design entity.

cistmt	::=	`comp` (id$_c$, id$_e$, gmap, ipmap, opmap)
gmap	::=	assoc$_g^*$
ipmap	::=	assoc$_{ip}^*$
opmap	::=	assoc$_{op}^*$
assoc$_g$	::=	(id,e)
assoc$_{ip}$	::=	(name,e)
assoc$_{op}$	::=	(id,(name | `open`)) | (id(e),name)

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

3.3.3 Sequential statements

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

The ss entry defines the sequential statements that compose the body of processes. The signal assignment statement is represented with the $\Leftarrow$ operator; the variable assignment statement with the := operator. Also, we devise three control flow statements that have no equivalent in the VHDL syntax: the `falling` block statement, the `rising` block statement and the `rst` block (or reset) block statement. The `falling` statement (resp. `rising` ss) declares a block of sequential statements to be executed only at the falling edge (resp. rising edge) of the clock signal (see Section 3.6.5). Also, the `rst` statement declares two blocks, the first one must be executed during the initialization phase of the simulation; otherwise, the second one is executed (see Section 3.6.4). These invented constructs are equivalent to specific if-else statements that are commonly used in the body of a synchronous process (see Section **??** for an example of transcription of a specific if-else statement into one of these constructs).

3.3.4 Expressions, names and types

e	::=	e `and` e | e `or` e | `not` e | e = e | e $\neq$ e | e < e | e <= e | e > e | e >= e | e + e | e - e | name | natural | boolean | `(`e$^+$`)`
name	::=	id | id`(` e `)`
boolean	::=	`true` | `false`
τ	::=	`boolean` | `natural (`e`,` e`)` | `array (`τ`,` e`,` e`)`

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

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

3.4 Preliminary definitions

3.4.1 Semantic domains

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

TABLE 3.2: The *type* and *value* semantic types.

type	::=	$bool$ | $nat(n, n)$ | $array(type, n, n)$
a *value*	::=	b | n | arr
b	::=	'$\top$' | '$\bot$'
n	::=	`0` | `1` | ... | `NATMAX`
arr	::=	$(value^+)$

In Table 3.2, the *type* type is in any way similar to the τ entry of the $\mathcal{H}$-VHDL abstract syntax. However, all constraint bounds, that were taking the form of expressions in the constrained

natural and the array type indications, have been evaluated to natural numbers. `NATMAX` denotes the maximum value for a natural number. The `NATMAX` value depends on the implementation of the VHDL language; `NATMAX` must at least be equal to $2^{31} - 1$. Note that the *array* value contains at least one value as an array's index range contains at least one index.

Notation 3 (Partial functions)**.** *In the following sections, when the context is free from any ambiguity, we adopt the following notations:*

- *For all $f \in A \nrightarrow B$, $x \in f$ states that x is in the domain of function f.*
- *For all $f \in A \nrightarrow B$ and $g \in A \nrightarrow C$, $f \subseteq g$ states that the domain of f is a subset of the domain of g.*
- *For all $X \subset A$ and $f \in A \nrightarrow B$, $X \subseteq f$ states that X is a subset of the domain of f.*

3.4.2 Elaborated design and design state

Now, let us define the structure of an elaborated design which is a structure bound to a given $\mathcal{H}$-VHDL design and to a design store, i.e a global environment mapping identifiers to $\mathcal{H}$-VHDL designs. Only the designs referenced into the global design store can be instantiated as component instances in the behavior of a given design. The elaborated design structure is built during the elaboration phase (see Section 3.5). Then, the elaborated design will act as a runtime environment in the expression of the simulation rules. Let $ElDesign(d, \mathcal{D})$ be the set of the elaborated designs for a given $\mathcal{H}$-VHDL design d and a design store $\mathcal{D}$. An elaborated design is a composite environment built out of multiple sub-environments. Each sub-environment is a table, represented as a function, mapping identifiers of a certain category of constructs (e.g, input port identifiers) to their declaration information (e.g, type indication for input ports). We represent an elaborated design as a record where the fields are the sub-environments. An elaborated design is defined as follows:

Definition 14 (Elaborated Design)**.** *For a given $\mathcal{H}$-VHDL design $d \in design$ s.t. $d =$ `design` id_e id_a gens ports sigs behavior and a given design store $\mathcal{D} \in$ entity-id $\nrightarrow$ design, an elaborated design $\Delta \in ElDesign(d, \mathcal{D})$ is a record <Gens, Ins, Outs, Sigs, Ps, Comps> where:*

- *$Gens \in$ generic-id $\rightarrow (type \times value)$ where generic-id $= \{id \mid (id, \tau, e) \in gens\}$, is the function yielding the type and the value of generic constants.*
- *$Ins \in input\text{-}id \rightarrow type$ where input-id $= \{id \mid (\texttt{in}, id, \tau) \in ports\}$, is the function yielding the type of input ports.*
- *$Outs \in output\text{-}id \rightarrow type$ where output-id $= \{id \mid (\texttt{out}, id, \tau) \in ports\}$, is the function yielding the type of output ports.*
- *$Sigs \in$ declared-signal-id $\rightarrow type$ where declared-signal-id $= \{id \mid (id, \tau) \in sigs\}$, is the function yelding the type of declared signals.*
- *$Ps \in$ process-id $\rightarrow$ (variable-id$(id_p) \rightarrow (type \times value)$) where process-id $= \{id_p \mid \texttt{process}(id_p, sl, vars, ss) \in behavior\}$, is the function associating processes*

to their local environment. Local environments are functions mapping local variable identifiers to their corresponding type and value. Therefore, each set of local variable identifiers variable-id(id_p) depends on the process identifier (represented by id_p) passed as the first argument of the Ps function.

– *Comps* $\in$ *component-id* $\rightarrow$ *ElDesign*$(d_e, \mathcal{D})$, *where*
component-id $= \{id_c \mid \texttt{comp}(id_c, id_e, gm, ipm, opm) \in behavior\}$, *is the function mapping component instance identifiers to their own elaborated design version. The set* $ElDesign(d_e, \mathcal{D})$ *depends on the design* d_e *from which the component identifier* id_c, *passed as the first argument of the Comps function, is an instance. Design* d_e *is retrieved from the design store* $\mathcal{D}$ *s.t.* $d_e = \mathcal{D}(id_e)$.

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

Let $\Sigma(\Delta)$ be the set of design states for a given elaborated design Δ. A design state of Δ is defined as follows:

Definition 15 (Design state)**.** *A design state* $\sigma \in \Sigma(\Delta)$, *for a given design* $d \in design$, *a given design store* $\mathcal{D}$ *and an elaborated design* $\Delta \in ElDesign(d, \mathcal{D})$, *is a record* $<\mathcal{S}, \mathcal{C}, \mathcal{E}>$ *where:*

– $\mathcal{S} \in$ *signal-id* $\rightarrow$ *value, is the function yielding the current values of the design's signals (ports and declared signals).*

– $\mathcal{C} \in$ *component-id* $\rightarrow \Sigma(\Delta_c)$, *is the function yielding the current state of component instances, where* $\Delta_c = \Delta(id_c)$ *and* $id_c \in$ *component-id is the component identifier passed to the* $\mathcal{C}$ *function.*

– $\mathcal{E} \subseteq$ *signal-id* $\sqcup$ *component-id, is the set of signal and component instance identifiers that generated an event at the current design state.*

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

Notation 4 (No events design state)**.** *For a given* $\mathcal{H}$*-VHDL design* d, *a design store* $\mathcal{D}$, *and an elaborated design* $\Delta \in ElDesign(d, \mathcal{D})$, *the function* $NoEv \in \Sigma(\Delta) \rightarrow \Sigma(\Delta)$ *returns a design state similar to the one passed in parameter but with an empty set of events. I.e, for all design state* $\sigma \in \Sigma(\Delta)$ *s.t.* $\sigma = <\mathcal{S}, \mathcal{C}, \mathcal{E}>$, $NoEv(\sigma) = <\mathcal{S}, \mathcal{C}, \emptyset>$.

3.5 Elaboration rules

The goal of the elaboration phase is to build an elaborated design Δ along with a *default* state σ_e, out of a $\mathcal{H}$-VHDL design d and for a given design store $\mathcal{D}$. The elaboration relation performs type-checking operations over the declarative and behavioral parts of the design. Even though the elaboration of a design is described in the LRM, the formalization of this phase has been performed in few works only [7, 23, 52], and never in a setting that covers both syntactical well-formedness and type-checking of the designs. We are interested in the formalization of the elaboration phase because we are interested in the *well-formedness* of the programs generated by the HILECOP transformation. Here, the term well-formedness refers to a syntactically valid design, w.r.t. the syntactic rules of the VHDL language, and to a well-typed design, w.r.t. the typing rules defined in the LRM. Formalizing the elaboration phase is also a way to define how the runtime environment and the runtime state of the simulation are built. For now, we haven't tackle down the proof that the $\mathcal{H}$-VHDL designs generated by HILECOP are elaborable, i.e. syntactically well-formed and well-typed. As explained in Chapter 4, this task is foreseen in our work perspectives. In our own formalization of the elaboration phase, and contrary to what is prescribed by the LRM [32, p. 166], we are not dealing with the transformation of the component instantiation statements into block statements. We prefer to preserve the hierarchical structure of the design (i.e. its composite structure) during its elaboration. We argue that dealing with component instantiation statements instead of block statements does not complexify the semantics of the $\mathcal{H}$-VHDL simulation rules.

In the following sections, the green frames give additional explanations about the premises, and the red frames bring additional explanations about the side conditions of the inference rules qualifiying the relations of the $\mathcal{H}$-VHDL semantics.

3.5.1 Design elaboration

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

Definition 16 (Design store)**.** *A design store* $\mathcal{D} \in$ *entity-id* $\rightharpoonup\!\!\!\!\!\!*$ design *is a partial function mapping design identifiers (i.e. the entity identifier of designs) to their corresponding representation in abstract syntax. As a prerequisite to the elaboration of HILECOP-generated designs (i.e, resulting from the transformation of a SITPN into an $\mathcal{H}$-VHDL design), a particular design store $\mathcal{D}_\mathcal{H}$ is defined. Design store $\mathcal{D}_\mathcal{H}$ binds the* `transition` *and* `place` *identifiers to the definition of the* `place` *and* `transition` *designs in $\mathcal{H}$-VHDL abstract syntax:*

$$\mathcal{D}_\mathcal{H} := \{(\texttt{transition}, \texttt{design transition transition\_architecture } gens_t\ ports_t\ sigs_t\ cs_t),\ (\texttt{place}, \texttt{design place place\_architecture } gens_p\ ports_p\ sigs_p\ cs_p)\}$$

The full definition of the `place` *and* `transition` *designs in abstract syntax are given in Appendices A and B.*

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

DESIGNELAB

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

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

3.5.2 Generic clause elaboration

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

1. Transforming the type indication associated with the constant into a semantic type.
2. Checking that the default value, and/or the value associated with the constant in the dimensioning function, is well-typed.
3. Adding the couple constant identifier and *(type,value)* to the *Gens* sub-environment of Δ.

Premises

- $etype_g$ transforms a type indication, specifically attached to a generic constant declaration, into a *type* instance and checks its well-formedness (see Section 3.5.5).
- The e relation links an expression e to its value v in a given context (see Section 3.6.9). The context of evaluation for an expression is composed of a given elaborated design, a given design state, and given local environment. We omit symbols at the left of the thesis when they refer to empty structures. For instance, $\vdash e \xrightarrow{e} v$ is a notation for $\Delta_\emptyset, \sigma_\emptyset, \Lambda_\emptyset \vdash \text{e} \xrightarrow{e} v$.
- SE_l states that an expression is *locally* static (see Section 3.5.9).

- $v \in_c T$ and $\mathcal{M}(\mathrm{id}_g) \in_c T$ checks that the default value and the value of yielded by the dimensioning function belongs to the type of the declared generic constant (see Section 3.5.8).

Side conditions

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

GENELABDIMEN

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

The GENELABDEFAULT states that the value of declared generic constant is defined by its default value when no value is specified by the dimensioning function $\mathcal{M}$.

GENELABDEFAULT

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

GENELABCOMP

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

3.5.3 Port clause elaboration

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

Premises

- The *etype* relation associates a type indication to its corresponding semantic type and checks its well-formedness (see Section 3.5.5).
- The $defaultv$ relation associates a given semantic type to its implicit *default* value.

InPortElab

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

OutPortElab

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

PortElabComp

$$\frac{\Delta, \sigma \vdash \mathrm{pdecl} \xrightarrow{eports} \Delta', \sigma' \qquad \Delta', \sigma' \vdash \mathrm{ports} \xrightarrow{eports} \Delta'', \sigma''}{\Delta, \sigma \vdash \mathrm{pdecl}, \mathrm{ports} \xrightarrow{eports} \Delta'', \sigma''}$$

3.5.4 Architecture declarative part elaboration

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

SigElab

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

SigElabComp

$$\frac{\Delta, \sigma \vdash \mathrm{sdecl} \xrightarrow{esigs} \Delta', \sigma' \qquad \Delta', \sigma' \vdash \mathrm{sigs} \xrightarrow{esigs} \Delta'', \sigma''}{\Delta, \sigma \vdash \mathrm{sdecl}, \mathrm{sigs} \xrightarrow{esigs} \Delta'', \sigma''}$$

3.5.5 Type indication elaboration

The *etype* relation checks the well-formedness of a type indication τ, and transforms it into a semantic *type* (as defined in Table 3.2). A type indication τ is well-formed in the context Δ if τ denotes the `boolean` keyword or the `natural` or `array` keywords with a *well-formed* constraint, and a well-formed element type in the `array` case.

ETypeBool

$$\frac{}{\Delta \vdash \texttt{boolean} \xrightarrow{etype} bool}$$

ETypeNat

$$\frac{\Delta \vdash (\mathrm{e}, \mathrm{e}') \xrightarrow{econstr} (v, v')}{\Delta \vdash \texttt{natural}(\mathrm{e}, \mathrm{e}') \xrightarrow{etype} nat(v, v')}$$

$$\textsc{ETypeArray}\quad \frac{\Delta \vdash \tau \xrightarrow{etype} T \qquad \Delta \vdash (\mathrm{e}, \mathrm{e}') \xrightarrow{econstr} (v, v')}{\Delta \vdash \texttt{array}(\tau, \mathrm{e}, \mathrm{e}') \xrightarrow{etype} array(T, v, v')}$$

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

- its bounds are globally static expressions [32, p.36] conforming to the $nat(0, \texttt{NATMAX})$ type after evaluation.
- its lower bound value is inferior or equal to its upper bound value.

Remark 1 (Type of constraints)**.** *As the VHDL language reference stays unclear about the type of range and index constraints [32, p.33], we add the restriction that range and index constraints must have bounds of the $nat(0, \texttt{NATMAX})$ type, i.e. the interval of natural numbers representable with the VHDL language.*

Premises

- The $\in_c$ relation states that a given value conforms to a given type (see Section 3.5.5).
- The SE_g relation states that an expression is *globally* static (see Section 3.5.9).

$$\textsc{EConstr}\quad \frac{\begin{array}{lll}\Delta \vdash SE_g(\mathrm{e}) & \Delta \vdash \mathrm{e} \xrightarrow{e} v & v \in_c nat(0, \texttt{NATMAX}) \\ \Delta \vdash SE_g(\mathrm{e}') & \Delta \vdash \mathrm{e}' \xrightarrow{e} v' & v' \in_c nat(0, \texttt{NATMAX})\end{array}}{\Delta \vdash (\mathrm{e}, \mathrm{e}') \xrightarrow{econstr} (v, v')}\ v \leq v'$$

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

$$\textsc{ETypeGBool}\quad \frac{}{\vdash \texttt{boolean} \xrightarrow{etype} bool} \qquad \textsc{ETypeGNat}\quad \frac{\Delta \vdash (\mathrm{e}, \mathrm{e}') \xrightarrow{econstr_g} (v, v')}{\vdash \texttt{natural}(\mathrm{e}, \mathrm{e}') \xrightarrow{etype} nat(v, v')}$$

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

- its bounds are locally static expressions [32, p.36] conforming to the $nat(0, \texttt{NATMAX})$ type after evaluation.
- its lower bound value is inferior or equal to its upper bound value.

ECONSTRG

$$\frac{\begin{array}{lll} SE_l(\mathrm{e}) & \vdash \mathrm{e} \xrightarrow{e} v & v \in_c nat(0, \texttt{NATMAX}) \\ SE_l(\mathrm{e}') & \vdash \mathrm{e}' \xrightarrow{e} v' & v' \in_c nat(0, \texttt{NATMAX}) \end{array}}{\vdash (\mathrm{e}, \mathrm{e}') \xrightarrow{econstr_g} (v, v')} \; v \leq v'$$

3.5.6 Behavior elaboration

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

Elaboration of concurrent statements

The elaboration of the composition of concurrent statements is performed in a sequential manner.

CSPARELAB

$$\frac{\mathcal{D}, \Delta, \sigma \vdash \mathrm{cs} \xrightarrow{ebeh} \Delta', \sigma' \qquad \mathcal{D}, \Delta', \sigma' \vdash \mathrm{cs}' \xrightarrow{ebeh} \Delta'', \sigma''}{\mathcal{D}, \Delta, \sigma \vdash \mathrm{cs} \;||\; \mathrm{cs}' \xrightarrow{ebeh} \Delta'', \sigma''}$$

CSNULLELAB

$$\frac{}{\mathcal{D}, \Delta, \sigma \vdash \texttt{null} \xrightarrow{ebeh} \Delta, \sigma}$$

Process statement elaboration

To elaborate a process statement, the *ebeh* relation associates the process identifier with a local environment in the *Ps* sub-environment of Δ. The *ebeh* builds the local environment from the process's local variable declaration list (see the *evars* relation). The *ebeh* relation also checks that the sequential statements composing the body of the process are well-typed (see the $valid_{ss}$ relation in Section 3.5.11).

Premises

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

Side conditions

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

PsElab

$$\frac{\Delta, \Lambda_\emptyset \vdash \text{vars} \xrightarrow{evars} \Lambda \qquad \Delta, \sigma, \Lambda \vdash \texttt{valid}_{ss}(\text{ss})}{\mathcal{D}, \Delta, \sigma \vdash \texttt{process}\ (\text{id}_p, \text{sl}, \text{vars}, \text{ss}) \xrightarrow{ebeh} \Delta \cup (id_p, \Lambda), \sigma} \quad \begin{array}{l} \text{id}_p \notin \Delta \\ \text{sl} \subseteq Ins(\Delta) \cup Sigs(\Delta) \end{array}$$

Process declarative part elaboration

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

VarElab

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

VarElabComp

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

Component instantiation statement elaboration

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

Premises

- The *emapg* relation builds a function $\mathcal{M} \in$ *generic-id* $\rightharpoonup$ *value* out of a generic map (see definition below).

- $\texttt{valid}_{ipm}$ (resp. $\texttt{valid}_{opm}$) states that an input port map (resp. output port map) is valid, i.e well-formed and well-typed (see Section 3.5.10).

Side conditions

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

CompElab

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

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

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

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

AssocGElab

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

GMElab

$$\frac{\mathcal{M} \vdash \text{assoc}_g \xrightarrow{emapg} \mathcal{M}' \quad \mathcal{M}' \vdash \text{gmap} \xrightarrow{emapg} \mathcal{M}''}{\mathcal{M} \vdash \text{assoc}_g,\ \text{gmap} \xrightarrow{emapg} \mathcal{M}''}$$

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

3.5.7 Implicit default value

According to the VHDL LRM, at the declaration of a port, a signal or a variable, these items must receive an implicit default value depending on their types [32, p.61, 64, 173]. The $defaultv$ relation determines the default value for a given type.

$$\frac{}{\texttt{bool} \xrightarrow{defaultv} \bot}\ \textsc{DefaultVBool} \qquad \frac{}{nat(n,m) \xrightarrow{defaultv} n}\ \textsc{DefaultVCNat}\ n \leq m$$

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

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

3.5.8 Typing relation

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

$$\frac{}{b \in_c bool}\ \textsc{IsBool}\ b \in \mathbb{B} \qquad \frac{}{n \in_c nat(l,u)}\ \textsc{IsCNat}\ n \in [l,u] \qquad \frac{v_i \in_c T}{\Delta \vdash (v_1,\ldots,v_n) \in_c array(T,l,u)}\ \textsc{Array}\ \begin{array}{l} i = 1,\ldots,n \\ n = (u-l)+1 \end{array}$$

3.5.9 Static expressions

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

Locally static expressions

An expression is *locally* static if:

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

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

$$\frac{}{SE_l(\mathrm{n})}\ \textsc{LSENat}\ \mathrm{n} \in \mathbb{N} \qquad \frac{}{SE_l(\mathrm{b})}\ \textsc{LSEBool}\ \mathrm{b} \in \mathbb{B} \qquad \frac{SE_l(\mathrm{e})}{SE_l(\texttt{not e})}\ \textsc{LSENot} \qquad \frac{SE_l(\mathrm{e}) \quad SE_l(\mathrm{e}')}{SE_l(\mathrm{e}\ \texttt{op}\ \mathrm{e}')}\ \textsc{LSEBinOp}\ \texttt{op} \in \{+,-,=,\neq,<,\leq,>,\geq,\texttt{and},\texttt{or}\}$$

Globally static expressions

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

- It is a generic constant.
- It is an array aggregate composed of globally static expressions.
- It is a locally static expression.

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

$$\frac{SE_l(\mathrm{e})}{\Delta \vdash SE_g(\mathrm{e})}\ \textsc{GSELocal} \qquad \frac{}{\Delta \vdash SE_g(\mathrm{id}_g)}\ \textsc{GSEGen}\ \ \mathrm{id}_g \in Gens(\Delta) \qquad \frac{\Delta \vdash SE_g(\mathrm{e}_i)}{\Delta \vdash SE_g((\mathrm{e}_1, \ldots, \mathrm{e}_n))}\ \textsc{GSEAggregate}\ \ i = 1, \ldots, n$$

3.5.10 Valid port map

Valid input port map

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

- All ports defined in Δ_c are exactly mapped once in the input port map.
- For each input port map entry, the formal and actual part are of the same type.

Premises

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

$$\frac{\Delta, \Delta_c, \sigma, \mathcal{L}_\emptyset \vdash \mathrm{ipmap} \xrightarrow{list_{ipm}} \mathcal{L} \qquad \texttt{check}_{pm}(Ins(\Delta_c), \mathcal{L})}{\Delta, \Delta_c, \sigma \vdash \texttt{valid}_{ipm}(\mathrm{ipmap})}\ \textsc{ValidIPM}$$

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

Side conditions

- $\mathrm{id}_f \in Ins(\Delta_c)$ checks that the identifier id_f is an input port identifier of Δ_c.
- $\mathrm{id}_f \notin \mathcal{L}$ checks that the port identifier id_f is not already mapped, i.e. it is not already referenced in the $\mathcal{L}$ set.

LISTIPMSIMPLE

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

Premises

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

Side conditions

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

LISTIPMPARTIAL

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

LISTIPMCONS

$$\frac{\Delta, \Delta_c, \sigma, \mathcal{L} \vdash \mathrm{assoc}_{ip} \xrightarrow{list_{ipm}} \mathcal{L}' \qquad \Delta, \Delta_c, \sigma, \mathcal{L}' \vdash \mathrm{ipmap} \xrightarrow{list_{ipm}} \mathcal{L}''}{\Delta, \Delta_c, \sigma, \mathcal{L} \vdash \mathrm{assoc}_{ip}, \mathrm{ipmap} \xrightarrow{list_{ipm}} \mathcal{L}''}$$

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

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

Valid output port map

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

- An output port identifier appears at most once in the output port map.
- Two different output port identifiers cannot be connected to the same signal.
- For each output port map entry, the formal and the actual part are of the exact same type.

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

Premises

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

VALIDOPM

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

Side conditions

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

LISTOPMSIMPLETOSIMPLE

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

Side conditions

$Outs_c(\mathrm{id}_f) = T$ and $Sigs(\mathrm{id}_a) = \mathtt{array}(T, n, m)$ checks that the type of id_f and the type of the elements of id_a are the same. Note that $\mathrm{id_a}$ must denote an array as id_f is mapped to one subelement of id_a.

ListOPMSimpleToPartial

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

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

ListOPMSimpleToOpen

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

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

Remark 3 (Unconnected output port.). *We forbid the case where an indexed formal part corresponding to the subelement of a composite output port is unconnected, i.e* $(\mathrm{id}_f(e_i), \mathtt{open})$*, as it could lead to the case where some subelements of a composite output port are connected while others are not (error case in [32, p.7]).*

ListOPMPartialToSimple

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

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

ListOPMPartialToPartial

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

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

ListOPMCons

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

3.5.11 Valid sequential statements

The $\mathtt{valid}_{ss}$ predicate states that a sequential statement is well-typed in the context Δ, σ, Λ.

Well-typed signal assignment

Premises

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

WTSig

$$\frac{\Delta, \sigma, \Lambda \vdash e \xrightarrow{e} v \qquad v \in_c T}{\Delta, \sigma, \Lambda \vdash \texttt{valid}_{ss}(\mathrm{id}_s \Leftarrow \mathrm{e})} \quad \begin{array}{l} \mathrm{id}_s \in \mathit{Sigs}(\Delta) \cup \mathit{Outs}(\Delta) \\ \Delta(\mathrm{id}_s) = T \end{array}$$

WTIdxSig

$$\frac{\begin{array}{ll} \Delta, \sigma, \Lambda \vdash e \xrightarrow{e} v & v \in_c T \\ \Delta, \sigma, \Lambda \vdash e_i \xrightarrow{e} v_i & v_i \in_c \mathit{nat}(n, m) \end{array}}{\Delta, \sigma, \Lambda \vdash \texttt{valid}_{ss}(\mathrm{id}_s(\mathrm{e}_i) \Leftarrow \mathrm{e})} \quad \begin{array}{l} \mathrm{id}_s \in \mathit{Sigs}(\Delta) \cup \mathit{Outs}(\Delta) \\ \Delta(\mathrm{id}_s) = \mathit{array}(T, n, m) \end{array}$$

Well-typed variable assignment

WTVar

$$\frac{\Delta, \sigma, \Lambda \vdash \mathrm{e} \xrightarrow{e} v \qquad v \in_c T}{\Delta, \sigma, \Lambda \vdash \texttt{valid}_{ss}(\mathrm{id}_v := \mathrm{e})} \quad \begin{array}{l} \mathrm{id}_v \in \Lambda \\ \Lambda(\mathrm{id}_v) = (T, \mathit{val}) \end{array}$$

WTIdxVar

$$\frac{\begin{array}{ll} \Delta, \sigma, \Lambda \vdash e \xrightarrow{e} v & v \in_c T \\ \Delta, \sigma, \Lambda \vdash e_i \xrightarrow{e} v_i & v_i \in_c \mathit{nat}(n, m) \end{array}}{\Delta, \Lambda \vdash \texttt{valid}_{ss}(\mathrm{id}_v(\mathrm{e}_i) := \mathrm{e})} \quad \begin{array}{l} \mathrm{id}_v \in \Lambda \\ \Lambda(\mathrm{id}_v) = (\mathit{array}(T, n, m), \mathit{val}) \end{array}$$

Well-typed if statements

WTIf

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

WTIfElse

$$\frac{\Delta, \sigma, \Lambda \vdash \mathrm{e} \xrightarrow{e} v \qquad v \in_c \texttt{bool} \qquad \begin{array}{l} \Delta, \sigma, \Lambda \vdash \texttt{valid}_{ss}(\mathrm{ss}) \\ \Delta, \sigma, \Lambda \vdash \texttt{valid}_{ss}(\mathrm{ss}') \end{array}}{\Delta, \sigma, \Lambda \vdash \texttt{valid}_{ss}(\texttt{if (e) ss ss}')}$$

Well-typed loop statement

WTLOOP

$$\frac{\Delta,\sigma,\Lambda \vdash \mathrm{e} \xrightarrow{e} v \qquad v \in_c nat(0,\texttt{NATMAX}) \quad \Delta,\sigma,\Lambda \vdash \mathrm{e}' \xrightarrow{e} v' \qquad v' \in_c nat(0,\texttt{NATMAX}) \qquad \Delta,\sigma,\Lambda' \vdash \texttt{valid}_{ss}(\mathrm{ss})}{\Delta,\sigma,\Lambda \vdash \texttt{valid}_{ss}(\texttt{for}\ (\mathrm{id}_v, \mathrm{e}, \mathrm{e}')\ \mathrm{ss})} \ \Lambda' = \Lambda \cup (\mathrm{id}_v, (nat(v,v'), v))$$

Well-typed rising and falling edge blocks

WTRISING

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

WTFALLING

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

Well-typed rst blocks

WTRST

$$\frac{\Delta,\sigma,\Lambda \vdash \texttt{valid}_{ss}(\mathrm{ss}) \qquad \Delta,\sigma,\Lambda \vdash \texttt{valid}_{ss}(\mathrm{ss}')}{\Delta,\sigma,\Lambda \vdash \texttt{valid}_{ss}(\texttt{rst}\ \mathrm{ss}\ \mathrm{ss}')}$$

Well-typed null statement

WTNULL

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

3.6 Simulation rules

In this section, we formalize a specific simulation algorithm for the $\mathcal{H}$-VHDL designs. This algorithm is much simpler than the one presented in the LRM. This is mostly due to the fact that $\mathcal{H}$-VHDL is a subset of VHDL that aims at the description of synthesizable and synchronous designs. Synthesizable designs mean that the only kind of signal assignment used to describe the design behaviors are δ-delay signal assignments. Leaving apart the synchronous side, we only need a simulation algorithm that performs *delta cycles* (see Section 3.1.2) to simulate such synthesizable designs. However, $\mathcal{H}$-VHDL designs are also synchronous designs. As such, an $\mathcal{H}$-VHDL design is equipped with a clock input port. The value of the clock input port changes from 0 to 1 and inversely at constant rate, i.e. the clock rate. One can see the changing of the value of the clock input port as the result of the execution of a unit-delay signal assignment where the time clause is equal to half the clock period. Listing 3.4 illustrates how a $\mathcal{H}$-VHDL design `t1` can be embedded in another top-level design with a process regulating the value of a clock signal by using a unit-delay signal assignment. Listing 3.4 presents the behavioral part the embedding top-level design.

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

LISTING 3.4: An architecture to simulate a synchronous design. The architecture `toplevel_arch` is composed of the `clkp` process, that simulates a clock signal, and of an instance of the design `tl` named id_{tl}, i.e. the design under simulation.

In Listing 3.4, the `clkp` process assigns the clock signal with its inverse value after τ unit of time where τ corresponds to half the clock period. Of course, the clock period is specified by the designer of the circuit. The component instance id_{tl} corresponds to the instantiation of the $\mathcal{H}$-VHDL design `tl`, i.e. the one we want to simulate. The `clock` input port of id_{tl} is connected to the `clock` signal of the embedding design. Thus, when the value of the clock signal changes every half clock period, the processes that react to the changes of the clock signal, i.e. the so-called *synchronous* processes, are executed in the internal behavior of the component instance id_{tl}. Then, it is the turn of *combinational* processes to be executed until stabilization of all signal values. Using the terms of the LRM simulation algorithm, what will happen when trying to simulate the design of Listing 3.4 will be an alternation between one time cycle to move to the next clock event and execute synchronous processes, followed by many delta cycles corresponding to the execution of combinational processes until stabilization. Thus, we choose to embed this alternation within the definition of our simulation algorithm.

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

Finally, Algorithm 1 gives an overview of our simulation algorithm in a pseudo-code language. This simulation algorithm is formalized in a small-step semantics style in the following sections. Here, we say small-step semantics because the different intermediary states of the design under simulation are detailled abd registered in a simulation trace θ. This simulation trace is built incrementally through the execution of simulation cycles, and is returned at the end of Algorithm 1. However, the execution of sequential statements in the body of processes

are expressed with a big-step operational semantics.

Algorithm 1: Simulation(Δ, σ_e, cs, E_p, $nbOfCycles$)

```
   // Initialization phase.
1  σ'_e ← RunAllOnce(Δ,σ_e,cs)
2  σ ← Stabilize(Δ,σ'_e,cs)
   // Main loop.
3  T_c ← 0
4  θ ← [σ]
5  while T_c ≤ nbOfCycles do
6      σ_i ← Inject_↑(Δ,σ,E_p,T_c)
7      σ_↑ ← RisingEdge(Δ,σ_i,cs)
8      σ' ← Stabilize(Δ,σ_↑,cs)
9      σ'_i ← Inject_↓(Δ,σ',E_p,T_c)
10     σ_↓ ← FallingEdge(Δ,σ'_i,cs)
11     σ ← Stabilize(Δ,σ_↓,cs)
12     θ ← θ ++ [σ',σ]
13     T_c ← T_c + 1
14 return θ
```

Algorithm 1 defines an elaborated design Δ and a default design state σ_e as parameters. We assume that they are the result of the elaboration of the design being simulated. cs corresponds to the behavior of the design, i.e. the one that will be executed during the simulation. E_p is the environment that will provide values to the primary input ports. $nbOfCycles$ corresponds to the number of simulation cycles to be performed. Algorithm 1 begins with an initialization phase (following the LRM simulation algorithm); all processes are run excatly once (Line 1) followed by a stabilization phase (Line 2, multiple delta cycles). Line 3 initializes the variable T_c to zero. T_c represents the current count of simulation cycles. Line 4 initializes the variable θ with a singleton list holding state σ, i.e. the initial simulation state. Then, the same loop is performed until T_c reaches the prescribed number of simulation cycles. First, the values of primary input ports are retrieved from the environment E_p for the current count T_c and the current clock event (i.e. either $\uparrow$ or $\downarrow$); this is performed by the $\texttt{Inject}_\uparrow$ (resp. $\texttt{Inject}_\downarrow$) at the rising edge of the clock; then, all parts of cs that react to the rising edge (resp. falling edge) of the clock signal are executed; finally, the combinational parts of cs are executed until stabilization of all signals. At Line 12, the states obtained at the middle and at the end of the clock cycle are appended to the simulation trace θ. Note that we only register stable states in the simulation trace. To conclude the simulation cycle, the current count is incremented. After the execution of all simulation cycles, Algorithm 1 returns the simulation trace.

3.6.1 Full simulation

The full simulation process is decomposed in two steps. The first step is the elaboration phase that builds an elaborated version of a $\mathcal{H}$-VHDL design along with its default state. Previous to the elaboration phase, the top-level design receives a value for each of its generic constant; we refer to it as the *dimensioning* of the top-level design. The second step is the simulation phase

that executes the behavioral part of the top-level design starting from an initial state. The initial state is built by a specfic initialization phase. Then, the simulation is decomposed into simulation cycles. Each simulation cycle is divided in four parts entailed by the *synchronous* execution of $\mathcal{H}$-VHDL top-level designs, i.e designs whose behavior depend on a clock signal. The four parts are, first, the execution of concurrent statements responding to the rising edge of the clock signal, then, a phase of signal stabilization followed by the execution of concurrent statements responding to the falling edge of the clock signal, and finally another phase of signal stabilization. At each clock event, the value of the primary inputs of the design are captured and injected in the simulation; primary inputs receive values from the design environment. Here, the environment is represented by a function mapping input port identifiers to values depending on the current count of simulation cycles and the considered clock event.

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

FULLSIM

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

where:

- $\mathcal{M}_g \in Gens(\Delta) \rightharpoonup value$, the function yelding the values of generic constants for a given top-level design, refered to as the *dimensioning* function. Here, $Gens(\Delta)$ denotes the domain of $Gens(\Delta)$, i.e. the set of generic constant identifiers of Δ.
- $E_p \in (\mathbb{N} \times Clk) \rightarrow (Ins(\Delta) \rightarrow value)$, the function yelding a mapping from primary inputs (i.e, input ports of the top-level design) to values at a given simulation cycle count (i.e, the $\mathbb{N}$ argument), and a given clock event (i.e, the Clk argument, where $Clk = \{\uparrow, \downarrow\}$). Here, $Ins(\Delta)$ denotes the domain of $Ins(\Delta)$, i.e. the set of input port identifiers of Δ.
- τ, the number of simulation cycles to execute. The value of τ is decremented at each clock cycle until it reaches zero (see Section 3.6.2).

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

Hypothesis 1 (Stable primary inputs)**.** *The values of primary inputs (i.e, input ports of the top-level design) are captured at each clock event, and therefore are stable (i.e, their values do not change) between two contiguous clock events.*

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

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

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

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

3.6.2 Simulation loop

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

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

3.6.3 Simulation cycle

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

Notation 5 (Overriding union)**.** *For all partial function $f, f' \in X \nrightarrow Y$, $f \overleftarrow{\cup} f'$ denotes the overriding union of f and f' such that* $f \overleftarrow{\cup} f'(x) = \begin{cases} f'(x) & \text{if } x \in \mathtt{dom}(f') \\ f(x) & \text{otherwise} \end{cases}$

Notation 6 (Differentiated intersection domain)**.** *For all partial function $f, f' \in X \nrightarrow Y$, $f \overset{\neq}{\cap} f'$ denotes the intersection of the domain of f and f' for which f and f' yields different values. That is,* $f \overset{\neq}{\cap} f' = \{\, x \in \mathtt{dom}(f) \cap \mathtt{dom}(f') \mid f(x) \neq f'(x) \,\}$.

Definition 17 (Input port values update). *Given an $\mathcal{H}$-VHDL design $d \in design$, a design store $\mathcal{D} \in id \rightharpoonup design$, an elaborated design $\Delta \in ElDesign(d, \mathcal{D})$, a simulation environment $E_p \in (\mathbb{N} \times \{\uparrow, \downarrow\}) \rightarrow (Ins(\Delta) \rightarrow value)$, let us define the relation expressing the update of the values of the input ports of Δ at a given design state $\sigma \in \Sigma(\Delta)$, clock cycle count $\tau \in \mathbb{N}$, and clock event $clk \in \{\uparrow, \downarrow\}$, and thus resulting in a new state $\sigma_i \in \Sigma(\Delta)$. The relation is written $\texttt{Inject}_{clk}(\sigma, E_p, \tau, \sigma_i)$ and verifies that: $\sigma = <\mathcal{S}, \mathcal{C}, \mathcal{E}>$ and $\sigma_i = <\mathcal{S} \overleftarrow{\cup} E_p(\tau, clk), \mathcal{C}, \mathcal{E}>$.*

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

SIMCYC

$$\frac{\mathcal{D}, \Delta, \sigma_i \vdash \text{cs} \xrightarrow{\uparrow} \sigma_{\uparrow} \quad \mathcal{D}, \Delta, \sigma_{\uparrow} \vdash \text{cs} \xrightarrow{\rightsquigarrow} \sigma' \quad \mathcal{D}, \Delta, \sigma_i' \vdash \text{cs} \xrightarrow{\downarrow} \sigma_{\downarrow} \quad \mathcal{D}, \Delta, \sigma_{\downarrow} \vdash \text{cs} \xrightarrow{\rightsquigarrow} \sigma''}{\mathcal{D}, E_p, \Delta, \tau, \sigma \vdash \text{cs} \xrightarrow{\uparrow,\downarrow} \sigma', \sigma''} \quad \begin{array}{l} \texttt{Inject}_{\uparrow}(\sigma, E_p, \tau, \sigma_i) \\ \texttt{Inject}_{\downarrow}(\sigma', E_p, \tau, \sigma_i') \end{array}$$

3.6.4 Initialization rules

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

INIT

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

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

The *runinit* relation is defined by the Rules PSRUNINIT, COMPRUNINIT, PARRUNINIT and NULLRUNINIT which are detailled right below. The *stabilize* relation is defined in Section 3.6.6.

Evaluation of a process statement

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

Premises

- The i flag of the ss_i relation indicates that all sequential statements responding to the initialization phase (i.e, reset blocks) will be executed.
- The ss_i relation takes two states in its context, i.e. two σ. The first σ is the state used to evaluate expressions appearing in the process statement body; the second σ is the state that will be modified by the execution of signal assignment statements.

Side conditions

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

PSRUNINIT

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

Evaluation of a component instantiation statement

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

Premises

- The *mapip* relation evaluates the input port map i of id_c, thus modifying the internal state σ_c of id_c. The result is a new internal state σ_c'.
- The expression $\mathcal{D}(\mathrm{id}_e)$.cs refers to the internal behavior of the component instance id_c.
- State σ_c'' is the new internal state of component instance id_c resulting from the execution of its internal behavior.

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

Side conditions

- Δ_c is the elaborated version of the component instance id_c referenced in the *Comps* sub-environment of the embedding design Δ, i.e. $\Delta(id_c) = \Delta_c$.
- σ_c is the internal design state of the component instance id_c referenced in the component store of state σ, i.e. $\sigma(id_c) = \sigma_c$.
- The component store $\mathcal{C}''$ of state σ'' is equal to the component store $\mathcal{C}'$ of state σ' where the component instance id_c is assigned to its new internal state σ_c''.
- The expression $\mathcal{C} \overset{\neq}{\cap} \mathcal{C}''$ equals $\{\mathrm{id}_c\}$ if the internal state of the component instance id_c has changed after the evaluation of its input port map and its internal behavior. In other words, we register the component instance id_c as an eventful component instance if $\sigma_c \neq \sigma_c''$.

COMPRUNINIT

$$\frac{\begin{array}{c}\Delta, \Delta_c, \sigma, \sigma_c \vdash \mathrm{i} \xrightarrow{mapip} \sigma_c' \\ \mathcal{D}, \Delta_c, \sigma_c' \vdash \mathcal{D}(\mathrm{id_e}).\mathrm{cs} \xrightarrow{runinit} \sigma_c'' \\ \Delta, \Delta_c, \sigma, \sigma_c'' \vdash \mathrm{o} \xrightarrow{mapop} \sigma'\end{array}}{\mathcal{D}, \Delta, \sigma \vdash \mathtt{comp}\ (\mathrm{id}_c, \mathrm{id}_e, \mathrm{g}, \mathrm{i}, \mathrm{o}) \xrightarrow{runinit} \sigma''} \quad \begin{array}{l}\mathrm{id}_e \in \mathcal{D} \\ \Delta(\mathrm{id}_c) = \Delta_c, \sigma(\mathrm{id}_c) = \sigma_c \\ \sigma'' = <\mathcal{S}', \mathcal{C}'', \mathcal{E}' \cup (\mathcal{C} \overset{\neq}{\cap} \mathcal{C}'')> \\ \mathcal{C}'' = \mathcal{C}'(\mathrm{id}_c) \leftarrow \sigma_c''\end{array}$$

Evaluation of the composition of concurrent statements

Rule PARRUNINIT describes the evaluation of the parallel composition of two concurrent statements cs and cs′. The two concurrent statements are evaluated starting from the same state σ and they generate two different state σ' and σ''. The state resulting from the concurrent execution of cs and cs′ is the result of a merging between the starting state σ, and the two states σ' and σ''.

PARRUNINIT

$$\frac{\mathcal{D}, \Delta, \sigma \vdash \mathrm{cs} \xrightarrow{runinit} \sigma' \quad \mathcal{D}, \Delta, \sigma \vdash \mathrm{cs}' \xrightarrow{runinit} \sigma''}{\mathcal{D}, \Delta, \sigma \vdash \mathrm{cs} \,||\, \mathrm{cs}' \xrightarrow{runinit} \mathtt{merge}(\sigma, \sigma', \sigma'')} \quad \mathcal{E}' \cap \mathcal{E}'' = \emptyset$$

The `merge` function, defined in Listing in pseudo-Coq language, computes a new state based on the original state o, and the states s and s' yielded by the computation of two concurrent statements. In the resulting state, the signal value store $\mathcal{S}_m$ is a function merging together the

signal stores at state o, s and s'. S_m yields values from the signal store $\mathcal{S}$ (resp. $\mathcal{S}'$) for all signal that belongs to the set of events at state s (resp. s'), and yields values from the original signal store $\mathcal{S}_o$ for all eventless signals. The same goes for the merged component store C_m. The new set of events $\mathcal{E}_m$ is the union between the set of events at state s and the one at state s'. The `merge` function correctly merges the state o, s and s' only if the set of events of s and s' are disjoint. The PARRUNINIT rule, which appeals to the `merge` function, defines the condition of disjoint set of events as a side condition.

1 `Definition merge(o,s,s') :=`
2 `let o =` $<\mathcal{S}_o,\mathcal{C}_o,\mathcal{E}_o>$ `in`
3 `let s =` $<\mathcal{S},\mathcal{C},\mathcal{E}>$ `in`
4 `let s' =` $<\mathcal{S}',\mathcal{C}',\mathcal{E}'>$ `in`
5 `let` $\mathcal{S}_m(\mathrm{id}) = \begin{cases} \mathcal{S}(\mathrm{id}) & \texttt{if id} \in \mathcal{E} \\ \mathcal{S}'(\mathrm{id}) & \texttt{if id} \in \mathcal{E}' \texttt{ in} \\ \mathcal{S}_o(\mathrm{id}) & \textit{otherwise} \end{cases}$
6 `let` $\mathcal{C}_m(\mathrm{id}) = \begin{cases} \mathcal{C}(\mathrm{id}) & \texttt{if id} \in \mathcal{E} \\ \mathcal{C}'(\mathrm{id}) & \texttt{if id} \in \mathcal{E}' \texttt{ in} \\ \mathcal{C}_o(\mathrm{id}) & \textit{otherwise} \end{cases}$
7 `let` $\mathcal{E}_m = \mathcal{E} \cup \mathcal{E}'$ `in` $<\mathcal{S}_m,\mathcal{C}_m,\mathcal{E}_m>$.

LISTING 3.5: The `merge` function that fuses together an origin state o, with two states s and s' generated by the execution of two concurrent statements.

Remark 4 (No multiply-driven signals)**.** *For all states $\sigma = <\mathcal{S},\mathcal{C},\mathcal{E}>$ and $\sigma' = <\mathcal{S}',\mathcal{C}',\mathcal{E}'>$ resulting from the execution of two concurrent statements cs and cs', $\mathcal{E} \cap \mathcal{E}' = \emptyset$. Otherwise, there exists some multiply-driven signals, which are forbidden in our semantics.*

Rule NULLRUNINIT evaluates a null statement during the initialization phase. The evaluation of a null statement yields a state similar to the starting state.

NULLRUNINIT

$$\Delta, \sigma \vdash \texttt{null} \xrightarrow{runinit} \sigma$$

3.6.5 Clock phases rules

The following rules express the evaluation of concurrent statements at clock phases, i.e. the $\uparrow$ and $\downarrow$ phases. The clock signal, trigerring the evaluation of synchronous process statements, is represented by the reserved signal identifier `clk`. Thus, synchronous processes are processes containing the `clk` signal in their sensitivity list.

Evaluation of a process statement

The following rules describe the evaluation of a process statement at the occurrence of the rising or the falling edge of the clock signal. In the case where a process does not contain the

clk identifier in its sensitivity list, then its statement body is not executed during the clock phases (see Rules PsRENoClk and PsFENoClk). Otherwise, its statement body is executed. Depending on the considered clock event, falling blocks or rising blocks are executed when encountered in the body of a process (see Rules PsREClk and PsFEClk).

PsRENoClk

$$\frac{}{\mathcal{D}, \Delta, \sigma \vdash \texttt{process}\ (\mathrm{id}_p, \mathrm{sl}, \mathrm{vars}, \mathrm{ss}) \xrightarrow{\uparrow} \sigma} \quad \texttt{clk} \notin \mathrm{sl}$$

Premises

The $\uparrow$ flag in the $ss_{\uparrow}$ relation indicates that `rising` blocks will be executed.

PsREClk

$$\frac{\Delta, \sigma, \sigma, \Lambda \vdash \mathrm{ss} \xrightarrow{ss_{\uparrow}} \sigma', \Lambda'}{\mathcal{D}, \Delta, \sigma \vdash \texttt{process}\ (\mathrm{id}_p, \mathrm{sl}, \mathrm{vars}, \mathrm{ss}) \xrightarrow{\uparrow} \sigma'} \quad \begin{array}{l}\texttt{clk} \in \mathrm{sl} \\ \Delta(\mathrm{id}_p) = \Lambda\end{array}$$

PsFENoClk

$$\frac{}{\mathcal{D}, \Delta, \sigma \vdash \texttt{process}\ (\mathrm{id}_p, \mathrm{sl}, \mathrm{vars}, \mathrm{ss}) \xrightarrow{\downarrow} \sigma} \quad \texttt{clk} \notin \mathrm{sl}$$

Premises

The $\downarrow$ flag in the $ss_{\downarrow}$ relation indicates that `falling` blocks will be executed.

PsFEClk

$$\frac{\Delta, \sigma, \sigma, \Lambda \vdash \mathrm{ss} \xrightarrow{ss_{\downarrow}} \sigma', \Lambda'}{\mathcal{D}, \Delta, \sigma \vdash \texttt{process}\ (\mathrm{id}_p, \mathrm{sl}, \mathrm{vars}, \mathrm{ss}) \xrightarrow{\downarrow} \sigma'} \quad \begin{array}{l}\texttt{clk} \in \mathrm{sl} \\ \Delta(\mathrm{id}_p) = \Lambda\end{array}$$

Evaluation of a component instantiation statement

The following rules describe the evaluation of a component instantiation statement during clock phases. These rules are similar in every point to Rule CompRunInit that describes the evaluation of a component instantiation statement during the initialization phase. The only difference lies in the execution of the internal behavior of the component instance. During the clock phases, the falling relation, written $\xrightarrow{\downarrow}$, or the rising relation, written $\xrightarrow{\uparrow}$, evaluate the internal behavior of component instances.

COMPRE

$$\frac{\begin{array}{c}\Delta, \Delta_c, \sigma, \sigma_c \vdash \mathrm{i} \xrightarrow{mapip} \sigma_c' \\ \mathcal{D}, \Delta_c, \sigma_c' \vdash \mathcal{D}(id_e).\mathrm{cs} \xrightarrow{\uparrow} \sigma_c'' \\ \Delta, \Delta_c, \sigma, \sigma_c'' \vdash \mathrm{o} \xrightarrow{mapop} \sigma'\end{array}}{\mathcal{D}, \Delta, \sigma \vdash \mathtt{comp}\ (\mathrm{id}_c, \mathrm{id}_e, \mathrm{g}, \mathrm{i}, \mathrm{o}) \xrightarrow{\uparrow} \sigma''} \quad \begin{array}{l} id_e \in \mathcal{D} \\ \Delta(\mathrm{id}_c) = \Delta_c, \sigma(\mathrm{id}_c) = \sigma_c \\ \sigma'' = <\mathcal{S}', \mathcal{C}'', \mathcal{E}' \cup (\mathcal{C} \overset{\neq}{\cap} \mathcal{C}'')> \\ \mathcal{C}'' = \mathcal{C}'(\mathrm{id}_c) \leftarrow \sigma_c'' \end{array}$$

COMPFE

$$\frac{\begin{array}{c}\Delta, \Delta_c, \sigma, \sigma_c \vdash \mathrm{i} \xrightarrow{mapip} \sigma_c' \\ \mathcal{D}, \Delta_c, \sigma_c' \vdash \mathcal{D}(id_e).\mathrm{cs} \xrightarrow{\downarrow} \sigma_c'' \\ \Delta, \Delta_c, \sigma, \sigma_c'' \vdash \mathrm{o} \xrightarrow{mapop} \sigma'\end{array}}{\mathcal{D}, \Delta, \sigma \vdash \mathtt{comp}\ (\mathrm{id}_c, \mathrm{id}_e, \mathrm{g}, \mathrm{i}, \mathrm{o}) \xrightarrow{\downarrow} \sigma''} \quad \begin{array}{l} id_e \in \mathcal{D} \\ \Delta(\mathrm{id}_c) = \Delta_c, \sigma(\mathrm{id}_c) = \sigma_c \\ \sigma'' = <\mathcal{S}', \mathcal{C}'', \mathcal{E}' \cup (\mathcal{C} \overset{\neq}{\cap} \mathcal{C}'')> \\ \mathcal{C}'' = \mathcal{C}'(\mathrm{id}_c) \leftarrow \sigma_c'' \end{array}$$

Evaluation of the composition of concurrent statements

The following rules describe the evaluation of the composition of concurrent statements and the evaluation of null statements during the clock phases. These rules are similar to the ones described for the initialization phase. Thus, the reader can refer to Section 3.6.4 for more details.

PARFE

$$\frac{\mathcal{D}, \Delta, \sigma \vdash \mathrm{cs} \xrightarrow{\downarrow} \sigma' \quad \mathcal{D}, \Delta, \sigma \vdash \mathrm{cs}' \xrightarrow{\downarrow} \sigma''}{\mathcal{D}, \Delta, \sigma \vdash \mathrm{cs} \mid\mid \mathrm{cs}' \xrightarrow{\downarrow} \mathtt{merge}(\sigma, \sigma', \sigma'')} \ \mathcal{E}' \cap \mathcal{E}'' = \emptyset$$

NULLFE

$$\frac{}{\Delta, \sigma \vdash \mathtt{null} \xrightarrow{\downarrow} \sigma}$$

PARRE

$$\frac{\mathcal{D}, \Delta, \sigma \vdash \mathrm{cs} \xrightarrow{\uparrow} \sigma' \quad \mathcal{D}, \Delta, \sigma \vdash \mathrm{cs}' \xrightarrow{\uparrow} \sigma''}{\mathcal{D}, \Delta, \sigma \vdash \mathrm{cs} \mid\mid \mathrm{cs}' \xrightarrow{\uparrow} \mathtt{merge}(\sigma, \sigma', \sigma'')} \ \mathcal{E}' \cap \mathcal{E}'' = \emptyset$$

NULLRE

$$\frac{}{\Delta, \sigma \vdash \mathtt{null} \xrightarrow{\uparrow} \sigma}$$

3.6.6 Stabilization rules

The following rules describe the evaluation of concurrent statements, representing a design's behavior, during a stabilization phase. The stabilization phase triggers the execution of the combinational parts of the behavior by appealing to the *comb* relation. When the execution of the combinational parts of the behavior does not change the design state anymore, then we have reached a stable state and the stabilization phase ends (Rule STABILIZEEND). When the execution of the combinational parts produces some events, i.e. it changes the value of signals or the internal state of component instances, then the stabilization phase must continue until a stable state is reached (Rule STABILIZELOOP). In the formalization of the $\mathcal{H}$-VHDL simulation

algorithm, the set of events of a design state is useful to merge the states resulting from the execution of multiple concurrent statements (see Definition 3.5), and to determine if a stable state has been reachd. In the LRM simulation algorithm, the kernel process uses the set of events to resume the activity of processes. If one of the signal declared in a process' sensitivity list is registered in the current set of events, then the process body must be executed. We choose to disregard this aspect of the execution of process in the formalization of our simulation algorithm. Thus, all combinational processes are executed when a delta cycle is performed, i.e. through the use of the *comb* relation.

Side conditions

- In Rule StabilizeEnd, state σ is an eventless state, i.e. its event set $\mathcal{E}$ is empty.
- In Rule StabilizeLoop, state σ' is an eventful state and state σ'' is eventless.

StabilizeEnd

$$\frac{\mathcal{D}, \Delta, \sigma \vdash \text{cs} \xrightarrow{comb} \sigma}{\mathcal{D}, \Delta, \sigma \vdash \text{cs} \xrightarrow{\rightsquigarrow} \sigma} \; \mathcal{E} = \emptyset$$

StabilizeLoop

$$\frac{\mathcal{D}, \Delta, \sigma \vdash \text{cs} \xrightarrow{comb} \sigma' \qquad \mathcal{D}, \Delta, \sigma' \vdash \text{cs} \xrightarrow{\rightsquigarrow} \sigma''}{\mathcal{D}, \Delta, \sigma \vdash \text{cs} \xrightarrow{\rightsquigarrow} \sigma''} \; \begin{array}{l} \mathcal{E} \neq \emptyset \\ \mathcal{E}'' = \emptyset \end{array}$$

Evaluation of a process statement

Rule PsComb describes the execution of a process statement during a stabilization phase. Even synchronous processes can be executed during a stabilization phase, however, the falling and rising blocks are not interpreted. Thus, the evaluation of a *purely* synchronous process, defined only with falling or rising blocks and no combinational parts, does not change the design state during a stabilization phase.

Premises

- The c flag (for *combinational*) on the ss_c relation indicates that statements responding to clock events (i.e. `falling` and `rising` blocks) and statements executed during the initialization phase only (i.e. `rst` blocks) will not be considered.
- The set of events of state σ is emptied ($NoEv(\sigma)$, see Notation 4) before the evaluation of the process statement body. It corresponds to the consumption of the information brought by the event set. Once the information has been consumed, new events can be generated by executing the process body. Otherwise, the set of events is never empty, and a stable state is never reached.

PsComb

$$\frac{\Delta, \sigma, NoEv(\sigma), \Lambda \vdash \text{ss} \xrightarrow{ss_c} \sigma', \Lambda'}{\mathcal{D}, \Delta, \sigma \vdash \texttt{process}\ (\text{id}_p, \text{sl}, \text{vars}, \text{ss}) \xrightarrow{comb} \sigma'} \; \Delta(\text{id}_p) = \Lambda$$

Evaluation of a component instantiation statement

Rule CompComb describes the evaluation of a component instantiation statement during a stabilization phase. This rule is similar in every point to Rule CompRunInit, and Rules CompRE and CompFE, that describe the evaluation of a component instantiation statement during the initialization phase, and the clock phases. The only difference lies in the execution of the internal behavior of the component instance. During a stabilization, the *comb* relation evaluates the internal behavior of component instances. Otherwise, see Section 3.6.4 for more details about the premises and side conditions of Rule CompComb.

CompComb

$$\frac{\begin{array}{c}\Delta, \Delta_c, \sigma, \sigma_c \vdash \mathrm{i} \xrightarrow{mapip} \sigma_c' \\ \mathcal{D}, \Delta_c, \sigma_c' \vdash \mathcal{D}(id_e).\mathrm{cs} \xrightarrow{comb} \sigma_c'' \\ \Delta, \Delta_c, NoEv(\sigma), \sigma_c'' \vdash \mathrm{o} \xrightarrow{mapop} \sigma'\end{array}}{\mathcal{D}, \Delta, \sigma \vdash \mathtt{comp}\ (\mathrm{id}_c, \mathrm{id}_e, \mathrm{g}, \mathrm{i}, \mathrm{o}) \xrightarrow{comb} \sigma''} \quad \begin{array}{l} id_e \in \mathcal{D} \\ \Delta(\mathrm{id}_c) = \Delta_c, \sigma(\mathrm{id}_c) = \sigma_c \\ \sigma'' = <\mathcal{S}', \mathcal{C}'', \mathcal{E}' \cup (\mathcal{C} \overset{\neq}{\cap} \mathcal{C}'')> \\ \mathcal{C}'' = \mathcal{C}'(\mathrm{id}_c) \leftarrow \sigma_c'' \end{array}$$

Evaluation of the composition of concurrent statements

The following rules describe the evaluation of the composition of concurrent statements and the evaluation of null statements during a stabilization phase. These rules are similar to the ones describe for the initialization phase. Thus, the reader can refer to Section 3.6.4 for more details.

ParComb

$$\frac{\mathcal{D}, \Delta, \sigma \vdash \mathrm{cs} \xrightarrow{comb} \sigma' \qquad \mathcal{D}, \Delta, \sigma \vdash \mathrm{cs}' \xrightarrow{comb} \sigma''}{\mathcal{D}, \Delta, \sigma \vdash \mathrm{cs} \mid\mid \mathrm{cs}' \xrightarrow{comb} \mathtt{merge}(\sigma, \sigma', \sigma'')} \quad \mathcal{E}' \cap \mathcal{E}'' = \emptyset$$

NullComb

$$\frac{}{\Delta, \sigma \vdash \mathtt{null} \xrightarrow{comb} NoEv(\sigma)}$$

3.6.7 Evaluation of input and output port maps

Evaluation of an input port map

Here, we define the *mapip* relation that evaluates the input port map of a component instance. For each association of the input port map, the actual part is evaluated and the result is assigned to the formal part of the association, i.e. an input port (Rule MapipSimple) or an indexed input port (Rule MapipPartial) identifier. The following rules define the *mapip* relation.

MapipSimple

$$\frac{\Delta, \sigma \vdash \mathrm{e} \xrightarrow{e} v \qquad v \in_c T}{\Delta, \Delta_c, \sigma, \sigma_c \vdash (\mathrm{id}_s, \mathrm{e}) \xrightarrow{mapip} <\mathcal{S}', \mathcal{C}, \mathcal{E}>} \quad \begin{array}{l} \Delta_c(\mathrm{id}_s) = T \\ \sigma_c = <\mathcal{S}, \mathcal{C}, \mathcal{E}> \\ \mathcal{S}' = \mathcal{S}(id_s) \leftarrow v \end{array}$$

MAPIPPARTIAL

$$\frac{\begin{array}{ll}\Delta,\sigma \vdash \mathrm{e} \xrightarrow{e} v & v \in_c T \\ \vdash \mathrm{e}_i \xrightarrow{e} v_i & v_i \in_c nat(n,m)\end{array}}{\Delta,\Delta_c,\sigma,\sigma_c \vdash (\mathrm{id}_s(\mathrm{e}_i),\mathrm{e}) \xrightarrow{mapip} <\mathcal{S}',\mathcal{C},\mathcal{E}>} \quad \begin{array}{l}\Delta_c(\mathrm{id}_s) = array(T,n,m) \\ \sigma_c = <\mathcal{S},\mathcal{C},\mathcal{E}> \\ \mathcal{S}' = \mathcal{S}(id_s) \leftarrow \texttt{set\_at}(v,v_i,\mathcal{S}(id_s))\end{array}$$

MAPIPCOMP

$$\frac{\Delta,\Delta_c,\sigma,\sigma_c \vdash \mathrm{assoc}_{ip} \xrightarrow{mapip} \sigma_c' \qquad \Delta,\Delta_c,\sigma,\sigma_c' \vdash \mathrm{ipmap} \xrightarrow{mapip} \sigma_c''}{\Delta,\Delta_c,\sigma,\sigma_c \vdash \mathrm{assoc}_{ip},\ \mathrm{ipmap} \xrightarrow{mapip} \sigma_c''}$$

Evaluation of an output port map

Here, we define the *mapop* relation that evaluates the output port map of a component instance. For each association of the output port map, the formal part is evaluated and the result is assigned to the actual part of the association. There can be five kind of associations in an output port map:

- an output port identifier (of the component instance) is associated with the `open` keyword, thus denoting an unconnected output port in the output interface of the component instance
- an output port identifier is associated with an internal signal or an output port identifier of the embedding design (Rule MAPOPSIMPLETOSIMPLE)
- an output port identifier is associated with an indexed internal signal or an output port identifier of the embedding design (Rule MAPOPSIMPLETOPARTIAL)
- an indexed output port identifier is associated with an internal signal or an output port identifier of the embedding design (Rule MAPOPPARTIALTOSIMPLE)
- an indexed output port identifier is associated with an indexed internal signal or an output port identifier of the embedding design (Rule MAPOPPARTIALTOPARTIAL)

Remark 5 (Out ports and *e*)**.** *We can not use the e relation to interpret the values of output ports, because output ports are write-only constructs. We append the flag o to the e relation (i.e, e_o) to enable the evaluation of output port identifiers as regular signal identifier expressions.*

The e_o relation is only defined to retrieve the value of output ports from a store signal $\mathcal{S}$ under a design state $\sigma = <\mathcal{S},\mathcal{C},\mathcal{E}>$.

$$\frac{}{\Delta,\sigma \vdash \mathrm{id}_s \xrightarrow{e_o} \sigma(\mathrm{id}_s)} \quad \begin{array}{l}\mathrm{id}_s \in Outs(\Delta) \\ \mathrm{id}_s \in \sigma\end{array}$$

IDXOUTO

$$\frac{\vdash \mathrm{e}_i \xrightarrow{e} v_i \qquad v_i \in_c nat(n,m)}{\Delta,\sigma \vdash \mathrm{id}_s(\mathrm{e}_i) \xrightarrow{e_o} \texttt{get\_at}(i,\sigma(\mathrm{id}_s))} \quad \begin{array}{l}\mathrm{id}_s \in Outs(\Delta) \\ \mathrm{id}_s \in \sigma \\ \Delta(\mathrm{id}_s) = array(T,n,m) \\ i = v_i \texttt{ mod } n\end{array}$$

The following rules define the *mapop* relation.

MAPOPOPEN

$$\Delta, \Delta_c, \sigma, \sigma_c \vdash (\mathrm{id}_f, \mathtt{open}) \xrightarrow{mapop} \sigma$$

Side conditions

In the signal store $\mathcal{S}'$, value v is assigned to the signal identifier id_a. If this assignment changes the value of id_a, then an event on signal id_a must be registered. The expression $\mathcal{E} \cup \mathcal{S} \overset{\neq}{\cap} \mathcal{S}'$ represents the set of signals that have a different value in signal store $\mathcal{S}$ and $\mathcal{S}'$.

MAPOPSIMPLETOSIMPLE

$$\frac{\Delta_c, \sigma_c \vdash \mathrm{id}_f \xrightarrow{e_o} v \qquad v \in_c T}{\Delta, \Delta_c, \sigma, \sigma_c \vdash (\mathrm{id}_f, \mathrm{id}_a) \xrightarrow{mapop} <\mathcal{S}', \mathcal{C}, \mathcal{E}'>}$$

$\mathrm{id}_a \in Sigs(\Delta) \cup Outs(\Delta)$
$\Delta(\mathrm{id}_a) = T$
$\sigma = <\mathcal{S}, \mathcal{C}, \mathcal{E}>$
$\mathcal{S}' = \mathcal{S}(id_a) \leftarrow v, \mathcal{E}' = \mathcal{E} \cup (\mathcal{S} \overset{\neq}{\cap} \mathcal{S}')$

MAPOPSIMPLETOPARTIAL

$$\frac{\vdash \mathrm{e}_i \xrightarrow{e} v_i \qquad v \in_c T \qquad \Delta_c, \sigma_c \vdash \mathrm{id}_f \xrightarrow{e_o} v \qquad v_i \in_c nat(n,m)}{\Delta, \Delta_c, \sigma, \sigma_c \vdash (\mathrm{id}_f, \mathrm{id}_a(\mathrm{e}_i)) \xrightarrow{mapop} <\mathcal{S}', \mathcal{C}, \mathcal{E}'>}$$

$\mathrm{id}_a \in Sigs(\Delta) \cup Outs(\Delta)$
$\Delta(\mathrm{id}_a) = array(T, n, m)$
$\sigma = <\mathcal{S}, \mathcal{C}, \mathcal{E}>$
$\mathcal{S}' = \mathcal{S}(id_a) \leftarrow \mathtt{set\_at}(v, v_i, \mathcal{S}(id_a))$
$\mathcal{E}' = \mathcal{E} \cup (\mathcal{S} \overset{\neq}{\cap} \mathcal{S}')$

MAPOPPARTIALTOSIMPLE

$$\frac{\Delta_c, \sigma_c \vdash \mathrm{id}_f(\mathrm{e}'_i) \xrightarrow{e_o} v \qquad v \in_c T}{\Delta, \Delta_c, \sigma, \sigma_c \vdash (\mathrm{id}_f(\mathrm{e}'_i), \mathrm{id}_a) \xrightarrow{mapop} <\mathcal{S}', \mathcal{C}, \mathcal{E}'>}$$

$\mathrm{id}_a \in Sigs(\Delta) \cup Outs(\Delta)$
$\Delta(\mathrm{id}_a) = T$
$\sigma = <\mathcal{S}, \mathcal{C}, \mathcal{E}>$
$\mathcal{S}' = \mathcal{S}(id_a) \leftarrow v, \mathcal{E}' = \mathcal{E} \cup (\mathcal{S} \overset{\neq}{\cap} \mathcal{S}')$

MAPOPPARTIALTOPARTIAL

$$\frac{\vdash \mathrm{e}_i \xrightarrow{e} v_i \qquad v \in_c T \qquad \Delta_c, \sigma_c \vdash \mathrm{id}_f(\mathrm{e}'_i) \xrightarrow{e_o} v \qquad v_i \in_c nat(n,m)}{\Delta, \Delta_c, \sigma, \sigma_c \vdash (\mathrm{id}_f(\mathrm{e}'_i), \mathrm{id}_a(\mathrm{e}_i)) \xrightarrow{mapop} <\mathcal{S}', \mathcal{C}, \mathcal{E}'>}$$

$\mathrm{id}_a \in Sigs(\Delta) \cup Outs(\Delta)$
$\Delta(\mathrm{id}_a) = array(T, n, m)$
$\sigma = <\mathcal{S}, \mathcal{C}, \mathcal{E}>$
$\mathcal{S}' = \mathcal{S}(id_a) \leftarrow \mathtt{set\_at}(v, v_i, \mathcal{S}(id_a))$
$\mathcal{E}' = \mathcal{E} \cup (\mathcal{S} \overset{\neq}{\cap} \mathcal{S}')$

MAPOPCOMP

$$\frac{\Delta, \Delta_c, \sigma, \sigma_c \vdash \mathrm{assoc}_{po} \xrightarrow{mapop} \sigma' \qquad \Delta, \Delta_c, \sigma', \sigma_c \vdash \mathrm{opmap} \xrightarrow{mapop} \sigma''}{\Delta, \Delta_c, \sigma, \sigma_c \vdash \mathrm{assoc}_{po},\ \mathrm{opmap} \xrightarrow{mapop} \sigma''}$$

3.6.8 Evaluation of sequential statements

Here, we define the *ss* relation that evaluates the sequential statements defining the body of processes. The phases of a simulation cycle affect the evaluation of sequential statements. For instance, reset blocks are only evaluated during an initialization phase, falling blocks during a falling edge phase... Thus, we append a specific flag to the *ss* relation to enable the evaluation of specific sequential statements at particular phases of the simulation cycle. There are four different flags, the c flag to denote the execution of combinational statements only, the i flag to enable the execution of reset blocks, the $\uparrow$ (resp. $\downarrow$) flag to enable the execution of rising (resp. falling) blocks. Writing the *ss* relation with no flag indicates that the evaluation of a given sequential statement is the same for every phase of the simulation cycle. A flag is tranferred from the conclusion to the premises when an sequential statement is composed of inner sequential blocks.

Signal assigment statement

A signal assignment generates a new design state with a modified signal store and a new set of events. Note that there are two states on the left side of the thesis symbol. σ represents the state holding the current values of signals, and σ_w holds the new values of signals (i.e. the *written* state).

Side conditions

The expression $\mathcal{S} \overset{\neq}{\cap} \mathcal{S}'_w$ registers signal id_s as an eventful signal if its value after assignment, i.e. in the signal store $\mathcal{S}'_w$, is different from its current value at state σ, i.e. in the signal store $\mathcal{S}$.

SIGASSIGN

$$\frac{\Delta, \sigma, \Lambda \vdash \mathrm{e} \xrightarrow{e} v \qquad v \in_c T}{\Delta, \sigma, \sigma_w, \Lambda \vdash \mathrm{id}_s \Leftarrow \mathrm{e} \xrightarrow{ss} <\mathcal{S}'_w, \mathcal{C}_w, \mathcal{E}'_w>, \Lambda}$$

$$\begin{aligned} &\mathrm{id}_s \in \mathit{Sigs}(\Delta) \cup \mathit{Outs}(\Delta) \\ &\Delta(\mathrm{id}_s) = T \\ &\mathcal{S}'_w = \mathcal{S}_w(id_s) \leftarrow v \\ &\mathcal{E}'_w = \mathcal{E}_w \cup (\mathcal{S} \overset{\neq}{\cap} \mathcal{S}'_w) \end{aligned}$$

IDXSIGASSIGN

$$\frac{\begin{array}{ll}\Delta, \sigma, \Lambda \vdash \mathrm{e}_i \xrightarrow{e} v_i & v \in_c T \\ \Delta, \sigma, \Lambda \vdash \mathrm{e} \xrightarrow{e} v & v_i \in_c \mathit{nat}(n, m)\end{array}}{\Delta, \sigma, \sigma_w, \Lambda \vdash \mathrm{id}_s(\mathrm{e}_i) \Leftarrow \mathrm{e} \xrightarrow{ss} <\mathcal{S}'_w, \mathcal{C}_w, \mathcal{E}'_w>, \Lambda}$$

$$\begin{aligned} &\mathrm{id}_s \in \mathit{Sigs}(\Delta) \cup \mathit{Outs}(\Delta) \\ &\Delta(\mathrm{id}_s) = \mathit{array}(T, n, m) \\ &\mathcal{S}'_w = \mathcal{S}_w(id_s) \leftarrow \texttt{set\_at}(v, v_i, \mathcal{S}_w(id_s)) \\ &\mathcal{E}'_w = \mathcal{E}_w \cup (\mathcal{S} \overset{\neq}{\cap} \mathcal{S}'_w) \end{aligned}$$

Variable assignment statement

A variable assignment statement modifies the value of a variable defined in a local environment Λ.

VARASSIGN

$$\frac{\Delta, \sigma, \Lambda \vdash \mathrm{e} \xrightarrow{e} v \quad v \in_c T}{\Delta, \sigma, \sigma_w, \Lambda \vdash \mathrm{id}_v := \mathrm{e} \xrightarrow{ss} \sigma_w, \Lambda(id_v) \leftarrow (T, v)} \begin{array}{l} \mathrm{id}_v \in \Lambda \\ \Lambda(\mathrm{id}_v) = (T, val) \end{array}$$

IDXVARASSIGN

$$\frac{\begin{array}{cc} \Delta, \sigma, \Lambda \vdash \mathrm{e}_i \xrightarrow{e} v_i & v_i \in_c nat(n, m) \\ \Delta, \sigma, \Lambda \vdash \mathrm{e} \xrightarrow{e} v & v \in_c T \end{array}}{\Delta, \sigma, \sigma_w, \Lambda \vdash \mathrm{id}_v(\mathrm{e}_i) := \mathrm{e} \xrightarrow{ss} \sigma_w, \Lambda(id_v) \leftarrow (T, \texttt{set\_at}(v, v_i, val))} \begin{array}{l} \mathrm{id}_v \in \Lambda \\ \Lambda(\mathrm{id}_v) = (array(T, n, m), val) \end{array}$$

Remark 6 (Local variables and persistent values)**.** *In the LRM, the value of local variables is persistent through the multiple execution of a process. However, in the definition of the* `place` *and* `transition` *designs, and in the VHDL programs generated by HILECOP, all local variables are initialized by an assignment statement at the beginning of the body of processes. Thus, to simplify the $\mathcal{H}$-VHDL semantics, we choose not to consider local variables as persistent memory as their values are renewed at each execution of a process.*

If statement

Here, we present the classical evaluation of if and if-else statements.

IF$\top$

$$\frac{\Delta, \sigma, \Lambda \vdash \mathrm{e} \xrightarrow{e} \top \quad \Delta, \sigma, \sigma_w, \Lambda \vdash \mathrm{ss} \xrightarrow{ss} \sigma'_w, \Lambda'}{\Delta, \sigma, \sigma_w, \Lambda \vdash \texttt{if}\ (\mathrm{e})\ \mathrm{ss} \xrightarrow{ss} \sigma'_w, \Lambda'}$$

IF$\bot$

$$\frac{\Delta, \sigma, \Lambda \vdash \mathrm{e} \xrightarrow{e} \bot}{\Delta, \sigma, \sigma_w, \Lambda \vdash \texttt{if}\ (\mathrm{e})\ \mathrm{ss} \xrightarrow{ss} \sigma_w, \Lambda}$$

IFELSE$\top$

$$\frac{\Delta, \sigma, \Lambda \vdash \mathrm{e} \xrightarrow{expr} \top \quad \Delta, \sigma, \sigma_w, \Lambda \vdash \mathrm{ss} \xrightarrow{ss} \sigma'_w, \Lambda'}{\Delta, \sigma, \sigma_w, \Lambda \vdash \texttt{if}\ (\mathrm{e})\ \mathrm{ss}\ \mathrm{ss}' \xrightarrow{ss} \sigma'_w, \Lambda'}$$

IFELSE$\bot$

$$\frac{\Delta, \sigma, \Lambda \vdash \mathrm{e} \xrightarrow{e} \bot \quad \Delta, \sigma, \sigma_w, \Lambda \vdash \mathrm{ss}' \xrightarrow{ss} \sigma'_w, \Lambda'}{\Delta, \sigma, \sigma_w, \Lambda \vdash \texttt{if}\ (\mathrm{e})\ \mathrm{ss}\ \mathrm{ss}' \xrightarrow{ss} \sigma'_w, \Lambda'}$$

Loop statement

Here, we present the classical evaluation of for-loop statements.

LOOP$\bot$

$$\frac{\begin{array}{cc} & \Delta, \sigma, \sigma_w, \Lambda_i \vdash \mathrm{ss} \xrightarrow{ss} \sigma'_w, \Lambda' \\ \Delta, \sigma, \Lambda_i \vdash \mathrm{id}_v = \mathrm{e}' \xrightarrow{e} \bot & \Delta, \sigma, \sigma'_w, \Lambda' \vdash \texttt{for}\ (\mathrm{id}_v, \mathrm{e}, \mathrm{e}')\ \mathrm{ss} \xrightarrow{ss} \sigma''_w, \Lambda'' \end{array}}{\Delta, \sigma, \sigma_w, \Lambda \vdash \texttt{for}\ (\mathrm{id}_v, \mathrm{e}, \mathrm{e}')\ \mathrm{ss} \xrightarrow{ss} \sigma''_w, \Lambda''} \begin{array}{l} \mathrm{id}_v \in \Lambda \\ \Lambda(\mathrm{id}_v) = (T, val) \\ \Lambda_i = \Lambda(id_v) \leftarrow (T, val + 1) \end{array}$$

LOOP⊤

$$\frac{\Delta, \sigma, \Lambda_i \vdash \mathrm{id}_v = \mathrm{e}' \xrightarrow{e} \top}{\Delta, \sigma, \sigma_w, \Lambda \vdash \texttt{for}\ (\mathrm{id}_v, \mathrm{e}, \mathrm{e}')\ \mathrm{ss} \xrightarrow{ss} \sigma_w, \Lambda \setminus (id_v, \Lambda(id_v))} \quad \begin{array}{l} \mathrm{id}_v \in \Lambda \\ \Lambda(\mathrm{id}_v) = (T, val) \\ \Lambda_i = \Lambda(id_v) \leftarrow (T, val + 1) \end{array}$$

LOOPINIT

$$\frac{\Delta, \sigma, \Lambda \vdash \mathrm{e} \xrightarrow{e} v \quad \Delta, \sigma, \Lambda \vdash \mathrm{e}' \xrightarrow{e} v' \quad \Delta, \sigma, \sigma_w, \Lambda_i \vdash \texttt{for}\ (\mathrm{id}_v, \mathrm{e}, \mathrm{e}')\ \mathrm{ss} \xrightarrow{ss} \sigma'_w, \Lambda'}{\Delta, \sigma, \sigma_w, \Lambda \vdash \texttt{for}\ (\mathrm{id}_v, \mathrm{e}, \mathrm{e}')\ \mathrm{ss} \xrightarrow{ss} \sigma'_w, \Lambda'} \quad \begin{array}{l} \mathrm{id}_v \notin \Lambda \\ \Lambda_i = \Lambda \cup (id_v, (nat(v, v'), v)) \end{array}$$

Rising and falling edge block statements

Here, we define the execution of rising and falling blocks. Rising (resp. Falling) blocks are executed only during a rising (resp. falling) edge phase of a simulation cycle, i.e. when the flag ↑ (resp. ↓) is raised (Rule RISINGEGDEEXEC and FALLINGEDGEEXEC). Otherwise, the evaluation of these blocks is without effect on state σ_w and on the local environment Λ (Rules RISINGEDGEDEFAULT and FALLINGEDGEDEFAULT).

RISINGEDGEDEFAULT

$$\frac{}{\Delta, \sigma, \sigma_w, \Lambda \vdash \texttt{rising}\ \mathrm{ss} \xrightarrow{ss_f} \sigma_w, \Lambda}\ f \neq \uparrow$$

FALLINGEDGEDEFAULT

$$\frac{}{\Delta, \sigma, \sigma_w, \Lambda \vdash \texttt{falling}\ \mathrm{ss} \xrightarrow{ss_f} \sigma_w, \Lambda}\ f \neq \downarrow$$

RISINGEDGEEXEC

$$\frac{\Delta, \sigma, \sigma_w, \Lambda \vdash \mathrm{ss} \xrightarrow{ss_\uparrow} \sigma'_w, \Lambda'}{\Delta, \sigma, \sigma_w, \Lambda \vdash \texttt{rising}\ \mathrm{ss} \xrightarrow{ss_\uparrow} \sigma'_w, \Lambda'}$$

FALLINGEDGEEXEC

$$\frac{\Delta, \sigma, \sigma_w, \Lambda \vdash \mathrm{ss} \xrightarrow{ss_\downarrow} \sigma'_w, \Lambda'}{\Delta, \sigma, \sigma_w, \Lambda \vdash \texttt{falling}\ \mathrm{ss} \xrightarrow{ss_\downarrow} \sigma'_w, \Lambda'}$$

Rst block statement

Here, we define the evaluation of reset blocks. The first part of reset blocks is only evaluated during the initialization phase of a simulation, i.e. when the *i* flag is raised (Rule RSTEXEC). Otherwise, it is the second part of the reset block that is evaluated (Rule RSTDEFAULT). Remember that a reset block is the transcription of a if-else statement specifically devised for the $\mathcal{H}$-VHDL abstract syntax.

RSTDEFAULT

$$\frac{\Delta, \sigma, \sigma_w, \Lambda \vdash \mathrm{ss}' \xrightarrow{ss_f} \sigma'_w, \Lambda'}{\Delta, \sigma, \sigma_w, \Lambda \vdash \texttt{rst}\ \mathrm{ss}\ \mathrm{ss}' \xrightarrow{ss_f} \sigma'_w, \Lambda'}\ f \neq i$$

RSTEXEC

$$\frac{\Delta, \sigma, \sigma_w, \Lambda \vdash \mathrm{ss} \xrightarrow{ss_i} \sigma'_w, \Lambda'}{\Delta, \sigma, \sigma_w, \Lambda \vdash \texttt{rst}\ \mathrm{ss}\ \mathrm{ss}' \xrightarrow{ss_i} \sigma'_w, \Lambda'}$$

Composition of sequential statements and null statement

Here, we present the evaluation of the composition of sequential statements (Rule SEQSTMT) and of the null sequential statement (Rule NULLSTMT). When evaluating a sequence of statements, the same state σ holding the current value of signals is used to execute both part of the sequence. The written state σ_w is modified by the first part of the sequence, thus resulting in a state σ'_w. Then, σ'_w is used to evaluate the second part of the sequence.

$$\textsc{SeqStmt}\ \frac{\Delta, \sigma, \sigma_w, \Lambda \vdash \text{ss} \xrightarrow{ss} \sigma'_w, \Lambda' \qquad \Delta, \sigma, \sigma'_w, \Lambda' \vdash \text{ss}' \xrightarrow{ss} \sigma''_w, \Lambda''}{\Delta, \sigma, \sigma_w, \Lambda \vdash \text{ss}; \text{ss}' \xrightarrow{ss} \sigma''_w, \Lambda''} \qquad \textsc{NullStmt}\ \frac{}{\Delta, \sigma, \sigma_w, \Lambda \vdash \texttt{null} \xrightarrow{ss} \sigma_w, \Lambda}$$

3.6.9 Evaluation of expressions

Here, we present the evaluation of expressions used throughout the definition of $\mathcal{H}$-VHDL designs. Rules NAT, FALSE and TRUE describe the evaluation of natural number and boolean constants. Rule AGGREG describes the evaluation of an aggregate expression. Rule GEN presents the evaluation of a generic constant identifier. Rules SIG and VAR describe the evaluation of signal and variable identifiers. Rules IDXSIG and IDXVAR corresponds to the evaluation of indexed signal and indexed variable identifiers. In Rules IDXSIG and IDXVAR, $\texttt{get\_at}(i, a)$ is a function returning the i-th element of array a.

$$\textsc{Nat}\ \frac{}{\Delta, \sigma, \Lambda \vdash \text{n} \xrightarrow{e} n}\ \begin{array}{l}\text{n} \in \mathbb{N} \\ \text{n} \leq \texttt{NATMAX}\end{array} \qquad \textsc{False}\ \frac{}{\Delta, \sigma, \Lambda \vdash \texttt{false} \xrightarrow{e} \bot} \qquad \textsc{True}\ \frac{}{\Delta, \sigma, \Lambda \vdash \texttt{true} \xrightarrow{e} \top}$$

$$\textsc{Aggreg}\ \frac{\Delta, \sigma, \Lambda \vdash \text{e}_i \xrightarrow{e} v_i}{\Delta, \sigma, \Lambda \vdash (\text{e}_1, \ldots, \text{e}_n) \xrightarrow{e} (v_1, \ldots, v_n)}\ i = 1, \ldots, n$$

$$\textsc{Sig}\ \frac{}{\Delta, \sigma, \Lambda \vdash \text{id}_s \xrightarrow{e} \sigma(\text{id}_s)}\ \text{id}_s \in \mathit{Sigs}(\Delta) \cup \mathit{Ins}(\Delta) \qquad \textsc{Var}\ \frac{}{\Delta, \sigma, \Lambda \vdash \text{id}_v \xrightarrow{e} v}\ \begin{array}{l}\text{id}_v \in \Lambda \\ \Lambda(\text{id}_v) = (T, v)\end{array}$$

$$\textsc{Gen}\ \frac{}{\Delta, \sigma, \Lambda \vdash \text{id}_g \xrightarrow{e} v}\ \begin{array}{l}\text{id}_g \in \mathit{Gens}(\Delta) \\ \Delta(\text{id}_g) = (T, v)\end{array}$$

$$\textsc{IdxSig}\ \frac{\Delta, \sigma, \Lambda \vdash \text{e}_i \xrightarrow{e} v_i \qquad v_i \in_c \mathit{nat}(n, m)}{\Delta, \sigma, \Lambda \vdash \text{id}_s(\text{e}_i) \xrightarrow{e} \texttt{get\_at}(i, \sigma(\text{id}_s))}\ \begin{array}{l}\text{id}_s \in \mathit{Sigs}(\Delta) \cup \mathit{Ins}(\Delta) \\ \Delta(\text{id}_s) = \mathit{array}(T, n, m) \\ i = v_i \bmod n\end{array}$$

IDXVAR

$$\frac{\Delta, \sigma, \Lambda \vdash \mathrm{e}_i \xrightarrow{e} v_i \qquad v_i \in_c nat(n, m)}{\Delta, \sigma, \Lambda \vdash \mathrm{id}_v(\mathrm{e}_i) \xrightarrow{e} \texttt{get\_at}(i, v)} \begin{array}{l} \mathrm{id}_v \in \Lambda \\ \Lambda(\mathrm{id}_v) = (array(T, n, m), v) \\ i = v_i \bmod n \end{array}$$

Rule NATADD describe the evaluation of the addition between two expressions of the natural type. The operator $+_{\mathbb{N}}$ denotes the addition operator of natural numbers in the semantic world. We add as a side condition that the result of the addition between two natural numbers must not exceed the value of the NATMAX number (the greatest natural number representable in $\mathcal{H}$-VHDL). Rule NATSUB describes the evaluation of the substraction between two expressions of the natural type. Rule ORDOP describes the evaluation of the comparison between two expressions of the natural type. The result of the comparison is a Boolean value. Rules BOOLBINOP and NOTOP describes the evaluation of Boolean expressions. Rules EQOP and DIFFOP define the evaluation of the equality and difference between two expressions of the same type; the result is a Boolean value. Rule PARENTH describes the evaluation of a parenthesised expression.

NATADD

$$\frac{\Delta, \sigma, \Lambda \vdash \mathrm{e} \xrightarrow{e} v \qquad \Delta, \sigma, \Lambda \vdash \mathrm{e}' \xrightarrow{e} v'}{\Delta, \sigma, \Lambda \vdash \mathrm{e} \ + \ \mathrm{e}' \xrightarrow{e} v \ +_{\mathbb{N}} \ v'} \ v +_{\mathbb{N}} v' \leq \texttt{NATMAX}$$

NATSUB

$$\frac{\Delta, \sigma, \Lambda \vdash \mathrm{e} \xrightarrow{e} v \qquad \Delta, \sigma, \Lambda \vdash \mathrm{e}' \xrightarrow{e} v'}{\Delta, \sigma, \Lambda \vdash \mathrm{e} \ - \ \mathrm{e}' \xrightarrow{e} v \ -_{\mathbb{N}} \ v'} \ v \geq v'$$

ORDOP

$$\frac{\Delta, \sigma, \Lambda \vdash \mathrm{e} \xrightarrow{e} v \qquad \Delta, \sigma, \Lambda \vdash \mathrm{e}' \xrightarrow{e} v'}{\Delta, \sigma, \Lambda \vdash \mathrm{e}\ \mathrm{op}_{ordn}\ \mathrm{e}' \xrightarrow{e} v\ op_{ord\mathbb{N}}\ v'} \ \mathrm{op}_{ordn} \in \{<, \leq, >, \geq\}$$

BOOLBINOP

$$\frac{\Delta, \sigma, \Lambda \vdash \mathrm{e} \xrightarrow{e} v \qquad \Delta, \sigma, \Lambda \vdash \mathrm{e}' \xrightarrow{e} v'}{\Delta, \sigma, \Lambda \vdash \mathrm{e}\ \mathrm{op}_{bool}\ \mathrm{e}' \xrightarrow{e} v\ op_{\mathbb{B}}\ v'} \ \mathrm{op}_{bool} \in \{\texttt{and}, \texttt{or}\}$$

NOTOP

$$\frac{\Delta, \sigma, \Lambda \vdash \mathrm{e} \xrightarrow{e} v}{\Delta, \sigma, \Lambda \vdash \texttt{not}\ \mathrm{e} \xrightarrow{e} \neg v}$$

EQOP

$$\frac{\Delta, \sigma, \Lambda \vdash \mathrm{e} \xrightarrow{e} v \qquad \Delta, \sigma, \Lambda \vdash \mathrm{e}' \xrightarrow{e} v'}{\Delta, \sigma, \Lambda \vdash \mathrm{e} \ = \ \mathrm{e}' \xrightarrow{e} eq(v, v')}$$

DIFFOP

$$\frac{\Delta, \sigma, \Lambda \vdash \mathrm{e} \ = \ \mathrm{e}' \xrightarrow{e} v}{\Delta, \sigma, \Lambda \vdash \mathrm{e} \ \neq \ \mathrm{e}' \xrightarrow{e} \neg v}$$

PARENTH

$$\frac{\Delta, \sigma, \Lambda \vdash \mathrm{e} \xrightarrow{e} v}{\Delta, \sigma, \Lambda \vdash (\mathrm{e}) \xrightarrow{e} v}$$

In Rule EQOP, *eq* is the equality operator established for all types defined in the semantics. In the definition of *eq*, two natural numbers and two Booleans are compared with the Leibniz equality. Two values of an array type are equal if the sub-elements sharing the same index are equal w.r.t. the definition of the *eq* relation. Thus, to be equal, the two arrays must be of the same size.

3.7 An example of full simulation

In this section, we will demonstrate the full simulation of a $\mathcal{H}$-VHDL top-level design on the example of Listing 3.6. The aim here is to illustrate the formal rules of the $\mathcal{H}$-VHDL semantics. Listing 3.6 is the result of the transformation of the SITPN model presented in Figure 3.6 into a $\mathcal{H}$-VHDL design. To keep the examples within a reasonable size, Listing 3.6, and the other listings and derivation trees used in this section, refer to the generic constants, the ports and the internal signals of the `transition` and `place` designs by their short names. See Table C.1 for a correspondence between the short names and the full names of constants and signals of the `place` and `transition` designs. In Listing, the generated $\mathcal{H}$-VHDL design, named `tl`, declares its input and output ports at Line 7, and its internal signals at Line 10. The behavior of `tl` is defined by a `place` component instance id_p (Lines 15 to 23), a `transition` component instance id_t (Lines 28 to 16), and two processes, namely: the `marked` process (Lines 41 to 43), and the `fired` process (Lines 48 to 50).

```
1  design tl tla
2
3  -- Generic constants
4  ∅
5
6  -- Ports (ports_tl)
7  ((in, id_c0, boolean), (out, id_f0, boolean), (out, id_a0, boolean))
8
9  -- Declared (internal) signals (sigs_tl)
10 ((id_ft, boolean), (id_av, boolean), (id_rt, boolean),(id_m, boolean))
11
12 -- Behavior (cs_tl)
13
14 -- Place component instance id_p
15 comp (id_p, place,
16 -- Generic map
17 ((ian, 1), (oan, 1), (mm, 1)),
18
19 -- Input port map
20 ((im, 1), (iaw(0), 1), (oat(0), 0), (oaw(0), 1), (itf(0), id_ft), (otf(0), id_ft))
21
22 -- Output port map
23 ((oav(0), id_av), (pauths, open), (rtt(0), id_rt), (marked, id_m)))
24
```

```
25  ||
26
27  -- Transition component instance id_t
28  comp (id_t, transition,
29  -- Generic map
30  ((tt, 0), (ian, 1), (cn, 1)),
31
32  -- Input port map
33  ((ic(0), id_c0), (A, 0), (B, 0), (iav(0), id_av), (rt(0), id_rt), (pauths(0), true)),
34
35  -- Output port map
36  ((fired, id_ft)))
37
38  ||
39
40  -- The marked process
41  process (marked, {clk}, ∅,
42  (rst (id_a0 ⇐ false)
43       (falling (id_a0 ⇐ id_m or false))))
44
45  ||
46
47  -- The fired process
48  process (fired, {clk}, ∅,
49  (rst (id_f0 ⇐ false)
50       (rising (id_f0 ⇐ id_ft or false))))
```

LISTING 3.6: An example of $\mathcal{H}$-VHDL top-level design generated by the HILECOP transformation.

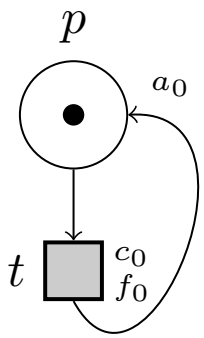


FIGURE 3.6: The SITPN model at the base of the generation of the top-level design presented in Listing 3.6.

The rule of Figure 3.7 states that the full simulation of the `tl` design (presented in Listing 3.6) over 1 clock cycle yields the simulation trace $(\sigma_0 :: \sigma_1 :: \sigma_2)$. The simulation over one clock cycle (the rightmost premise) yields a trace composed of two states: the state σ_1 at half the

clock period, and the state σ_2 at the end of the first cycle. The full simulation happens in the context of the HILECOP's design store $\mathcal{D}_{\mathcal{H}}$, the elaborated design Δ, an empty dimensioning function and an simulation environment E_p. Here, $ports_{tl}$ is an alias for the list of ports of `tl`, $sigs_{tl}$ for the list of internal signals of `tl`, and cs_{tl} for the behavior of `tl`. In what follows, we will detail the premises of the FULLSIM rule.

$$\frac{\vdots \qquad \vdots \qquad \vdots}{\mathcal{D}_{\mathcal{H}}, \emptyset \vdash \texttt{design tl} \ldots \xrightarrow{elab} \Delta, \sigma_e \qquad \mathcal{D}_{\mathcal{H}}, \Delta, \sigma_e \vdash cs_{tl} \xrightarrow{init} \sigma_0 \qquad \mathcal{D}_{\mathcal{H}}, E_p, \Delta, 1, \sigma_0 \vdash cs_{tl} \rightarrow (\sigma_1 :: \sigma_2)}\ \text{FULLSIM}$$

$$\mathcal{D}_{\mathcal{H}}, \Delta, \emptyset, E_p, 1 \vdash \texttt{design tl tla}\ \ ports_{tl}\ sigs_{tl}\ cs_{tl} \xrightarrow{full} (\sigma_0 :: \sigma_1 :: \sigma_2)$$

FIGURE 3.7: The FULLSIM rule applied to the `tl` design.

3.7.1 Elaboration of the `tl` design

The rule of Figure 3.8 states that the elaborated design Δ and the default design state σ_e are the result of the elaboration of the `tl` design. From left to right in the premises of the rule, the three first premises pertain to the elaboration of the declarative parts of the `tl` design, i.e. the generic constant declaration list, the port declaration list and the internal signal declaration list. The rightmost premise pertains to the elaboration of the behavior of the `tl` design.

$$\frac{\vdots \qquad \vdots \qquad \vdots \qquad \vdots}{\emptyset, \emptyset \vdash gens_{tl} \xrightarrow{egens} \Delta_0 \qquad \Delta_0, \emptyset \vdash ports_{tl} \xrightarrow{eports} \Delta_1, \sigma_{e1} \qquad \Delta_1, \sigma_{e1} \vdash sigs_{tl} \xrightarrow{esigs} \Delta_2, \sigma_{e2} \qquad \mathcal{D}_{\mathcal{H}}, \Delta_2, \sigma_{e2} \vdash cs_{tl} \xrightarrow{ebeh} \Delta, \sigma_e}$$

$$\mathcal{D}_{\mathcal{H}}, \emptyset \vdash \texttt{design tl tla}\ gens_{tl}\ ports_{tl}\ sigs_{tl}\ cs_{tl} \xrightarrow{elab} \Delta, \sigma_e$$

FIGURE 3.8: The DESIGNELAB rule applied to the `tl` design.

Elaboration of the declarative parts

The elaboration of the declarative parts populates the *Gens*, *Ins*, *Outs* and *Sigs* sub-environments of the elaborated design Δ. Here is the content of the *Gens*, *Ins*, *Outs* and *Sigs* sub-environments of Δ_2, where Δ_2 is the partial elaborated design after the elaboration of the declarative parts of the `tl` design (passed as a parameter of third and the fourth premises of the rule in Figure 3.8).

- $Gens(\Delta_2) := \emptyset$
- $Ins(\Delta_2) := \{(id_{c0}, bool)\}$
- $Outs(\Delta_2) := \{(id_{f0}, bool), (id_{a0}, bool)\}$
- $Sigs(\Delta_2) := \{(id_{ft}, bool), (id_{av}, bool), (id_{rt}, bool), (id_m, bool)\}$

The top-level design generated by the HILECOP transformation all have an empty list of generic constants (see Chapter **??** for more details about the transformation). Also, all ports and internal signals are of the Boolean type. Thus, there are no range constraint or index constraint to solve here. The `boolean` type indication is simply transformed into the *bool* semantic type.

The elaboration of the declarative parts also gives a default value to the signals in the signal store of the default design state σ_{e2}, where σ_{e2} is the partial default design state at the end of the elaboration of the declarative parts of the `tl` design (passed as a parameter of the third and the fourth premises of the rule in Figure 3.8). Here is the content of the signal store $\mathcal{S}$ of σ_{e2}.

- $\mathcal{S}(\sigma_{e2}) := \{(id_{c0}, \bot), (id_{f0}, \bot), (id_{a0}, \bot), (id_{ft}, \bot), (id_{av}, \bot), (id_{rt}, \bot), (id_m, \bot)\}$

The default value associated to the *bool* type is $\bot$, thus, all signals of the `tl design` are initialized to $\bot$ in the signal store of σ_{e2}.

Elaboration of the behavioral part

The behavior of the `tl` design contains two component instantiation statements and two process statements. Each one of these statements will be elaborated in sequence. First, we present the elaboration of the `marked` process to illustrate the elaboration of a process statement; then, we present the elaboration of the place component instance id_p to illustrate the elaboration of a component instantiation statement.

Elaboration of a process statement

The rule of Figure 3.9 presents the elaboration of the `marked` process defined in the behavior of the `tl` design.

$$\dfrac{\dfrac{}{\Delta_2, \emptyset \vdash \emptyset \xrightarrow{evars} \emptyset} \qquad \dfrac{\vdots}{\Delta_2, \sigma_{e2}, \emptyset \vdash \texttt{valid}_{ss}\left(\begin{array}{l}\texttt{rst}\ (id_{a0} \Leftarrow \texttt{false}) \\ (\texttt{falling}(id_{a0} \Leftarrow id_m\ \texttt{or}\ \texttt{false}))\end{array}\right)}\ \text{WTRST}}{\mathcal{D}_{\mathcal{H}}, \Delta_2, \sigma_{e2} \vdash \texttt{process}(\texttt{marked}, \texttt{clk}, \emptyset, \ldots) \xrightarrow{ebeh} \Delta_2 \cup (\texttt{marked}, \emptyset), \sigma_{e2}}\ \text{PSELAB}$$

FIGURE 3.9: The elaboration of the `marked` process defined in the behavior of the `tl` design.

The `marked` process is elaborated in the context $\mathcal{D}_{\mathcal{H}}, \Delta_2, \sigma_{e2}$ where Δ_2 and σ_{e2} are the partially-built elaborated design and default design state at a certain point of the elaboration of the behavioral part of the `tl` design. The elaboration of a process statement associates the process identifier to a local variable environment in the *Ps* sub-environment of the being-built elaborated design. The local variable environment is built out of the variable declaration list of the process. Here, the `marked` process has an empty variable declaration list. Thus, the binding (`marked`,$\emptyset$) is added in the *Ps* sub-environment of Δ_2.

The elaboration of process statement also performs static type-checking on the process statement body leveraging the $\texttt{valid}_{ss}$ relation. The rule of Figure 3.10 details the static type-checking of the statement body of the `marked` process (rightmost premise of the rule presented

in Figure 3.9). To keep the example within a reasonable size, we discard the context of some rules with it is not relevant. We annotate the rule names to describe the side conditions associated to a derivation.

$$\dfrac{\dfrac{}{\texttt{false} \xrightarrow{e} \bot}\ \textsc{False} \qquad \dfrac{}{\bot \in_c bool}\ \textsc{IsBool}}{\vdash \texttt{valid}_{ss}(id_{a0} \Leftarrow \texttt{false})}\ \textsc{WTSig}^1 \qquad \dfrac{\dfrac{\dfrac{}{\sigma_{e2} \vdash id_m \xrightarrow{e} \bot}\ \textsc{Sig}^2 \qquad \dfrac{}{\texttt{false} \xrightarrow{e} \bot}\ \textsc{False}}{\sigma_{e2} \vdash id_m\ \texttt{or}\ \texttt{false} \xrightarrow{e} \bot}\ \textsc{BoolBinOp} \qquad \dfrac{}{\bot \in_c bool}\ \textsc{IsBool}}{\sigma_{e2} \vdash \texttt{valid}_{ss}(id_{a0} \Leftarrow id_m\ \texttt{or}\ \texttt{false})}\ \textsc{WTSig}^1}{\sigma_{e2} \vdash \texttt{valid}_{ss}(\texttt{falling}(id_{a0} \Leftarrow id_m\ \texttt{or}\ \texttt{false}))}\ \textsc{WTFalling}$$

$$\textsc{WTRst}\quad \Delta_2, \sigma_{e2}, \emptyset \vdash \texttt{valid}_{ss}\left(\begin{array}{l}\texttt{rst}\ (id_{a0} \Leftarrow \texttt{false}) \\ (\texttt{falling}(id_{a0} \Leftarrow id_m\ \texttt{or}\ \texttt{false}))\end{array}\right)$$

(1) $\Delta_2(id_{a0}) = bool$ (2) $\sigma_{e2}(id_m) = \bot$

FIGURE 3.10: Static type-checking of the `marked` process statement body.

At the end of the elaboration of the `tl` design's behavior, the Ps sub-environment of Δ is as follows: $Ps(\Delta) := \{(\texttt{marked}, \emptyset), (\texttt{fired}, \emptyset)\}$

Elaboration of a component instantiation statement

The rule of Figure 3.11 presents the elaboration of the place component instance id_p belonging to the behavior of the `tl` design.

$$\dfrac{\vdots \quad \emptyset \vdash g_p \xrightarrow{emapg} \mathcal{M} \qquad \vdots \quad \mathcal{D}_\mathcal{H}, \mathcal{M} \vdash \texttt{design place} \ldots \xrightarrow{elab} \Delta_p, \sigma_p \qquad \vdots \quad \Delta_2, \Delta_p, \sigma_{e2} \vdash \texttt{valid}_{ipm}(i_p) \qquad \vdots \quad \Delta_2, \Delta_p \vdash \texttt{valid}_{opm}(o_p)}{\mathcal{D}_\mathcal{H}, \Delta_2, \sigma_{e2} \vdash \texttt{comp}\ (id_p, \texttt{place}, g_p, i_p, o_p) \xrightarrow{ebeh} \Delta_2 \cup (id_p, \Delta_p), \sigma_{e2} \cup (id_p, \sigma_p)}\ \textsc{CompElab}^1$$

(1) $id_p \notin \Delta_2$
$id_p \notin \sigma_{e2}$
$\texttt{place} \in \mathcal{D}_\mathcal{H}$
$\mathcal{M} \subseteq Gens(\Delta_p)$

FIGURE 3.11: The elaboration of the id_p component instance defined in the behavior of the `tl` design.

The elaboration of a component instantiation statement is divided in three parts. First, a dimensioning function is built out of the generic map of the component instance. Figure 3.12 shows a part of the creation of the dimensioning functioning $\mathcal{M}$ from the generic map of the component instance id_p. Basically, the elaboration of a generic map is a transformation from a syntactic construct, i.e. the generic map, into a function, i.e. the dimensioning function

$\mathcal{M}$. For each association of the generic map, the elaboration checks that the actual part of the association is a locally static expression (see Section 3.5.9).

$$\dfrac{\dfrac{\dfrac{}{SE_l(1)}\,\textsc{LSENat} \quad \dfrac{}{1 \xrightarrow{e} 1}\,\textsc{Nat}}{\emptyset \vdash (\texttt{ian},1) \xrightarrow{emapg} \{(\texttt{ian},1)\}}\,\textsc{AssocGElab}^1 \quad \dfrac{\vdots}{\{(\texttt{ian},1)\} \vdash (\texttt{oan},1),(\texttt{mm},1) \xrightarrow{emapg} \{(\texttt{ian},1),(\texttt{oan},1),(\texttt{mm},1)\}}\,\textsc{GMElab}}{\emptyset \vdash (\texttt{ian},1),(\texttt{oan},1),(\texttt{mm},1) \xrightarrow{emapg} \{(\texttt{ian},1),(\texttt{oan},1),(\texttt{mm},1)\}}\,\textsc{GMElab}$$

(1) $\texttt{ian} \notin \emptyset$

FIGURE 3.12: The elaboration of the generic map of the id_p component instance defined in the behavior of the `tl` design.

The second step of the elaboration of a component instance is to retrieve from the design store the design associated with the component instance, and to elaborate this design. Here, the design store is the HILECOP design store $\mathcal{D}_{\mathcal{H}}$, and the design associated with id_p is the `place` design. The dimensioning function $\mathcal{M}$ sets the value of the generic constants declared in the `place` design. The full code of `place` design is available in Appendix A. In Figures 3.13 and 3.14, we give the elaborated design Δ_p and the default design state σ_p resulting of the elaboration of the `place` design given the dimensioning function $\mathcal{M}$.

$$
\begin{aligned}
\Delta_p := \{ \\
&\mathit{Gens} := \quad \{(\texttt{ian}, (\mathit{nat}(0, \texttt{NATMAX}, 1)), \\
&\qquad\qquad\ (\texttt{oan}, (\mathit{nat}(0, \texttt{NATMAX}), 1)) \\
&\qquad\qquad\ (\texttt{mm}, (\mathit{nat}(0, \texttt{NATMAX}), 1))\} \\
&\mathit{Ins} := \quad \{(\texttt{im}, \mathit{nat}(0,1)), \\
&\qquad\qquad\ (\texttt{iaw}, \mathit{array}(\mathit{nat}(0,255), 0, 0)), \\
&\qquad\qquad\ (\texttt{oat}, \mathit{array}(\mathit{nat}(0,2), 0, 0)), \\
&\qquad\qquad\ (\texttt{oaw}, \mathit{array}(\mathit{nat}(0,255), 0, 0)), \\
&\qquad\qquad\ (\texttt{itf}, \mathit{array}(\mathit{bool}, 0, 0)), \\
&\qquad\qquad\ (\texttt{otf}, \mathit{array}(\mathit{bool}, 0, 0))\}, \\
&\mathit{Outs} := \quad \{(\texttt{oav}, \mathit{array}(\mathit{bool}, 0, 0)), \\
&\qquad\qquad\ (\texttt{pauths}, \mathit{array}(\mathit{bool}, 0, 0)), \\
&\qquad\qquad\ (\texttt{rtt}, \mathit{array}(\mathit{bool}, 0, 0))\} \\
&\mathit{Sigs} := \quad \{(\texttt{sits}, \mathit{nat}(0,1)), \\
&\qquad\qquad\ (\texttt{sm}, \mathit{nat}(0,1)), \\
&\qquad\qquad\ (\texttt{sots}, \mathit{nat}(0,1))\}, \\
&\mathit{Ps} := \quad \{(\texttt{input\_tokens\_sum}, \{(\text{"}\mathit{v\_internal\_its}\text{"}, (\mathit{nat}(0,1), 0))\}), \\
&\qquad\qquad\ (\texttt{output\_tokens\_sum}, \{(\text{"}\mathit{v\_internal\_ots}\text{"}, (\mathit{nat}(0,1), 0))\})\} \\
&\qquad\qquad\ (\texttt{priority\_evaluation}, \{(\text{"}\mathit{v\_saved\_ots}\text{"}, (\mathit{nat}(0,1), 0))\})\} \\
&\mathit{Comps} := \emptyset \\
\}
\end{aligned}
$$

FIGURE 3.13: An elaborated version of the place design built with the dimensioning function deduced from the generic map of the component instance id_p.

In Δ_p, all the types associated with ports and internal signals of the `place` design have been *resolved*; i.e. the expressions qualifying the bounds of the range and index constraints in type indications have been evaluated. For example, `array(boolean, 0, input_arcs_number-1)` is the type indication associated with the `input_transitions_fired` input port (i.e. `itf`) defined in the port clause of the `place` design. The dimensioning function $\mathcal{M}$ sets the value of the `input_arcs_number` (i.e. `ian`) generic constant to 1. After the elaboration, the type indication `array(boolean, 0, input_arcs_number-1)` is thus transformed into the semantic type $\mathit{array}(\mathit{bool}, 0, 0)$. Thus, we have $\Delta_p(\texttt{itf}) = \mathit{array}(\mathit{bool}, 0, 0)$ in the resulting Δ_p.

Figure 3.14 shows the default design state σ_p of Δ_p.

$$
\begin{aligned}
\sigma_p := \{ & \\
\quad \mathcal{S} := \quad & \{(\texttt{im}, (0)), (\texttt{iaw}, (0)), (\texttt{oat}, (0)), \\
& (\texttt{oaw}, (0)), (\texttt{itf}, (\bot)), (\texttt{otf}, (\bot)), \\
& (\texttt{oav}, (\bot)), (\texttt{pauths}, (\bot)), (\texttt{rtt}, (\bot)) \\
& (\texttt{sits}, 0), (\texttt{sm}, 0), (\texttt{sots}, 0)\}, \\
\quad \mathcal{C} := & \emptyset \\
\quad \mathcal{E} := & \emptyset \\
\} &
\end{aligned}
$$

FIGURE 3.14: The default design state σ_p of the elaborated design Δ_p.

The component store of design state σ_p is empty as there are no component instantiation statements in the behavior of the `place` design. The same stands for the *Comps* sub-environment of Δ_p. Also, the set of events of a default design state is always empty.

The final step in the elaboration of a component instantiation statement is to check the well-formedness and the well-typedness of the input and output port maps. The $\texttt{valid}_{ipm}$ and $\texttt{valid}_{opm}$ relations, defined in Section 3.5.10, state the validity of the port maps. The rule of Figure 3.15 presents a part of the construction of the $valid_{opm}$ relation applied to the output port map of the place component instance id_p. Note that Δ_p is necessary to check the validity of the output port map of id_p, as it holds the correspondence between port identifiers and port types.

$$\dfrac{\dfrac{\dfrac{}{SE_l(0)}\ \textsc{LSENat} \quad \dfrac{}{0 \xrightarrow{e} 0}\ \textsc{Nat} \quad \dfrac{}{0 \in_c nat(0,0)}\ \textsc{IsCNat}}{\Delta_2, \Delta_p, \emptyset, \emptyset \vdash (\texttt{oav}(0), id_{av}) \xrightarrow{list_{opm}} \{(\texttt{oav}, 0)\}, \{id_{av}\}}\ 1 \quad \vdots\ \textsc{ListOPMCons}_B}{\Delta_2, \Delta_p, \emptyset, \emptyset \vdash \begin{matrix}(\texttt{oav}(0), id_{av}), (\texttt{pauths}, \texttt{open}) \\ (\texttt{rtt}(0), id_{rt}), (marked, id_m)\end{matrix} \xrightarrow{list_{opm}} \begin{matrix}\{(\texttt{oav}, 0), \texttt{pauths}, \\ (\texttt{rtt}, 0), \texttt{marked}\}\end{matrix}, \{id_{av}, id_{rt}, id_m\}}\ \textsc{ListOPMCons}_A$$

$$\dfrac{\vdots}{\Delta_2, \Delta_p \vdash \texttt{valid}_{opm}\left(\begin{matrix}(\texttt{oav}(0), id_{av}), (\texttt{pauths}, \texttt{open}) \\ (\texttt{rtt}(0), id_{rt}), (\texttt{marked}, id_m)\end{matrix}\right)}\ \textsc{ValidOPM}$$

$$\dfrac{\vdots}{\Delta_2, \Delta_p, \{(\texttt{oav}, 0)\}, \{id_{av}\} \vdash \begin{matrix}(\texttt{pauths}, \texttt{open}), \\ (\texttt{rtt}(0), id_{rt}), \\ (marked, id_m)\end{matrix} \xrightarrow{list_{opm}} \begin{matrix}\{(\texttt{oav}, 0), \\ \texttt{pauths}, \\ (\texttt{rtt}, 0), \\ \texttt{marked}\}\end{matrix}, \{id_{av}, id_{rt}, id_m\}}\ \textsc{ListOPMCons}_B$$

(1) $\Delta_p(\texttt{oav}) = array(bool, 0, 0)$
$\Delta_2(id_{av}) = bool$
$\texttt{oav} \notin \emptyset$ and $(\texttt{oav}, 0) \notin \emptyset$
$id_{av} \notin \emptyset$

FIGURE 3.15: An example of validity checking performed on the output port map of the place component instance id_p. The bottom proof tree represents the top-right premise of the top proof tree.

At the end of the elaboration of the `tl` design's behavior, the *Comps* sub-environment of Δ is as follows: $Comps(\Delta) := \{(id_p, \Delta_p), (id_t, \Delta_t)\}$. Here, Δ_t represents the elaborated version of the `transition` design obtained from the elaboration of the transition component instance id_t.

Also, at the end of the elaboration, the component store of σ_e is as follows: $\mathcal{C}(\sigma_e) := \{(id_p, \sigma_p), (id_t, \sigma_t)\}$. Here, σ_t is the default design state of the transition component instance id_t.

3.7.2 Simulation of the tl design

Let us now present the rules pertaining to the simulation of the `tl` design, that is, pertaining to the execution of the `tl` design's behavior with respect to our formal simulation algorithm.

Initialization

The rule of Figure 3.16 presents the initialization phase in the proceeding of the simulation of the `tl` design. The initialization phase builds the initial state of the simulation. The first step

of the initialization, formalized by the *runinit* relation, runs the processes and the internal behavior of component instances exactly once (with the execution of the first part of reset blocks). Then, a stabilization phase follows.

$$\frac{\vdots \qquad \vdots}{\mathcal{D}_{\mathcal{H}}, \Delta, \sigma_e \vdash cs_{tl} \xrightarrow{runinit} \sigma' \qquad \mathcal{D}_{\mathcal{H}}, \Delta, \sigma_e \vdash cs_{tl} \overset{\rightsquigarrow}{\longrightarrow} \sigma_0}{\mathcal{D}_{\mathcal{H}}, \Delta, \sigma_e \vdash cs_{tl} \xrightarrow{init} \sigma_0}\ \textsc{Init}$$

FIGURE 3.16: The initialization phase, first step of the simulation of the `tl` design.

The rule in Figure 3.17 presents the execution of the `tl` design's behavior during the *runinit* phase. The `tl` design's behavior is defined by the composition of concurrent statements. Here, the `marked` process is at the head of the behavior, whereas it is not the case in Listing 3.6. We formally proved, with the Coq proof assistant, that the || composition operator for concurrent statements is commutative and associative with respect to the *runinit* relation. In Figure, the `marked` process is executed and yields the state σ'_e. Then, the rest of the `tl` design's behavior is executed and yields the state σ''_e. Finally, the starting state σ_e and the two states σ'_e and σ''_e are merged into one by the `merge` function.

$$\frac{\vdots \qquad \vdots}{\mathcal{D}_{\mathcal{H}}, \Delta, \sigma_e \vdash \texttt{process}(\texttt{marked}, \dots) \xrightarrow{runinit} \sigma'_e \qquad \mathcal{D}_{\mathcal{H}}, \Delta, \sigma_e \vdash cs'_{tl} \xrightarrow{runinit} \sigma''_e}{\mathcal{D}_{\mathcal{H}}, \Delta, \sigma_e \vdash \texttt{process}(\texttt{marked}, \dots) \mid\mid cs'_{tl} \xrightarrow{runinit} \texttt{merge}(\sigma_e, \sigma'_e, \sigma''_e)}\ \textsc{CompRunInit}^{1}$$

(1) $\mathcal{E}'_e \cap \mathcal{E}''_e$

FIGURE 3.17: The *runinit* phase applied to the concurrent statements composing the behavior of the `tl` design.

In what follows, we detail the execution of a process statement and of a component instantiation statement during the first part of the initialization, i.e. the *runinit* phase.

Execution of a process statement with the *runinit* relation

The rule in Figure 3.18 shows the execution of the `marked` process during the *runinit* phase. The first part of the reset block defining the statement body of the `marked` process is executed. This first part assigns the expression `false` to signal id_{a0}.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \texttt{false} \xrightarrow{e} \bot}\ \textsc{False} \qquad \dfrac{}{\bot \in_c bool}\ \textsc{IsBool}}{\mathcal{D}_{\mathcal{H}}, \Delta, \sigma_e, \sigma_e \vdash id_{a0} \Leftarrow \texttt{false} \xrightarrow{ss_i} \sigma'_e, \emptyset}\ \textsc{SigAssign}^2}{\mathcal{D}_{\mathcal{H}}, \Delta, \sigma_e \vdash \texttt{rst}(id_{a0} \Leftarrow \texttt{false})(\dots) \xrightarrow{ss_i} \sigma'_e, \emptyset}\ \textsc{RstExec}}{\mathcal{D}_{\mathcal{H}}, \Delta, \sigma_e \vdash \texttt{process}(\texttt{marked}, \dots) \xrightarrow{runinit} \sigma'_e}\ \textsc{PsRunInit}^1$$

$$\begin{array}{ll} (1)\ \Delta(\texttt{marked}) = \emptyset & \begin{array}{l} \Delta(id_{a0}) = bool \\ \sigma'_e = <\mathcal{S}'_e, \mathcal{C}'_e, \mathcal{E}'_e> \\ (2)\ \mathcal{S}'_e = \mathcal{S}_e(id_{a0}) \leftarrow \bot \\ \mathcal{E}'_e = \mathcal{E}_e \cup (\mathcal{S}_e \overset{\neq}{\cap} \mathcal{S}'_e) \end{array} \end{array}$$

FIGURE 3.18: The *runinit* phase applied to the concurrent statements composing the behavior of the `tl` design.

In the side conditions of the SIGASSIGN rule, an new event set $\mathcal{E}'_e$ is computed based on the event set $\mathcal{E}_e$ joined to the expression $\mathcal{S}_e \overset{\neq}{\cap} \mathcal{S}'_e$. This expression returns the set of signals with a different value between signal store $\mathcal{S}_e$ and singal store $\mathcal{S}'_e$. The only signal that possibly has a different value from $\mathcal{S}_e$ to $\mathcal{S}'_e$ is the assigned signal id_{a0}. Thus, this expression is a shorthand to test if the value of signal id_{a0} has changed after the execution of the signal assignment statement. If it is the case, then the event set receives the signal identifier id_{a0}; id_{a0} is then an eventful signal. In the present case, the value of signal id_{a0} was $\bot$ at state σ_e and is still $\bot$ after the execution of the signal assignment statement. Therefore, no event is registered on signal id_{a0}. When states σ_e, σ'_e and σ''_e will be merged (cf. Figure 3.17), if id_{a0} is part of the event set of state σ''_e, then, the merged state will return the value associated to id_{a0} in state σ''_e. We would have $\texttt{merge}(\sigma_e, \sigma'_e, \sigma''_e)(id_{a0}) = \sigma''_e(id_{a0})$. However, signal id_{a0} would be a potentially multiply-driven signal because both the `marked` process and the concurrent statement cs'_{tl} (cf. Figure 3.17) both assign the signal value.

Execution of a component instantiation statement with the *runinit* relation

The rule of Figure 3.19 presents the execution of the `place` component instance id_p during the *runinit* phase. The execution of a component instantiation statement is pretty much the same in all the phases of the simulation algorithm. The difference lies in the choice of the relation used to execute of the internal behavior of the component instance. During the *runinit* phase, it is the *runinit* relation that executes the internal behavior of component instances; during the falling edge phase, it is the $\downarrow$ relation that executes the internal behaviors, etc.

$$\frac{\vdots \qquad \vdots \qquad \vdots}{\Delta, \Delta_p, \sigma_e, \sigma_p \vdash i_p \xrightarrow{mapip} \sigma'_p \qquad \mathcal{D}_\mathcal{H}, \Delta_p, \sigma'_p \vdash cs_p \xrightarrow{runinit} \sigma''_p \qquad \Delta, \Delta_p, \sigma_e, \sigma''_p \vdash o_p \xrightarrow{mapop} \sigma'_e}\ \textsc{CompRunInit}^1$$
$$\mathcal{D}_\mathcal{H}, \Delta, \sigma_e \vdash \texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \xrightarrow{runinit} \sigma''_e$$

$$\sigma'_e = <\mathcal{S}'_e, \mathcal{C}'_e, \mathcal{E}'_e>$$
(1) $$\sigma''_e = <\mathcal{S}'_e, \mathcal{C}''_e, \mathcal{E}'_e \cup (\mathcal{C}_e \overset{\neq}{\cap} \mathcal{C}''_e)>$$
$$\mathcal{C}''_e = \mathcal{C}'_e(id_p) \leftarrow \sigma''_p$$

FIGURE 3.19: The execution of the `place` component instance id_p during the *runinit* phase (first part of the initialization).

The execution of a component instantiation statement is decomposed in four parts. First, the input ports of the component instance receive new values through the evaluation of the input port map. Second, the internal behavior of the component instance is executed. Thirdly, the evaluation of the output port map propagates the values coming from the output interface of the component to the signals of the embedding design. Finally, the component instance is assigned to its new internal state in the component store of the embedding design; here, σ''_p is assigned to id_p in the component store $\mathcal{C}''_e$. Moreover, if the new internal state of the component instance is different from its older internal state, then the component instance identifier is added to the event set of the embedding design. Here, the expression $\mathcal{C}_e \overset{\neq}{\cap} \mathcal{C}''_e$ performs the state comparison; we have:

$$\begin{aligned} \mathcal{C}_e \overset{\neq}{\cap} \mathcal{C}''_e &= \mathcal{C}_e \overset{\neq}{\cap} (\mathcal{C}'_e \leftarrow \sigma''_p) \\ &= \mathcal{C}_e \overset{\neq}{\cap} (\mathcal{C}_e \leftarrow \sigma''_p) \\ &= \begin{cases} \{id_p\} \texttt{ if } \sigma_p \neq \sigma''_p \\ \emptyset \textit{ otherwise} \end{cases} \end{aligned}$$

In the second line, we have $\mathcal{C}_e = \mathcal{C}'_e$ because the evaluation of the output port map (performed by the *mapop* relation) does not affect the component store.

The rule of Figure 3.20 gives a part of the evaluation of the input port map of id_p.

$$\dfrac{\dfrac{\dfrac{}{\Delta, \sigma_e \vdash 1 \xrightarrow{e} 1}\ \textsc{Nat} \quad \dfrac{}{1 \in_c nat(0,1)}\ \textsc{IsCNat}}{\Delta, \Delta_p, \sigma_e, \sigma_p \vdash (\texttt{im}, 1) \xrightarrow{mapip} \sigma'_{p0}}\ \textsc{MapipSimple}^1 \quad \dfrac{\vdots}{\Delta, \Delta_p, \sigma_e, \sigma'_{p0} \vdash \begin{matrix}(\texttt{iaw}(0),1)\\(\texttt{oat}(0),0),(\texttt{oaw}(0),1),\\(\texttt{itf}(0),id_{ft}),(\texttt{otf}(0),id_{ft})\end{matrix} \xrightarrow{mapip} \sigma'_p}}{\Delta, \Delta_p, \sigma_e, \sigma_p \vdash \begin{matrix}(\texttt{im},1),(\texttt{iaw}(0),1)\\(\texttt{oat}(0),0),(\texttt{oaw}(0),1),\\(\texttt{itf}(0),id_{ft}),(\texttt{otf}(0),id_{ft})\end{matrix} \xrightarrow{mapip} \sigma'_p}\ \textsc{MapipComp}$$

(1)
$$\begin{matrix}\Delta_p(\texttt{im}) = nat(0,1)\\ \sigma_p = <\mathcal{S},\mathcal{C},\mathcal{E}>\\ \mathcal{S}' = \mathcal{S}(\texttt{im}) \leftarrow 1\\ \sigma'_{p0} = <\mathcal{S}',\mathcal{C},\mathcal{E}>\end{matrix}$$

FIGURE 3.20: The evaluation of the input port map of the `place` component instance id_p.

The evaluation of the input port map of id_p changes the value of the `initial_marking` input port (i.e. `im`). We have $\sigma_p(\texttt{im}) = 0$ and $\sigma'_p(\texttt{im}) = 1$. As the value of one of its input port has changed, the `place` component instance id_p will be registered as an eventful component instance.

The rule of Figure 3.21 gives a part of the evaluation of the output port map of id_p.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash 0 \xrightarrow{e} 0} \quad \dfrac{}{0 \in_c nat(0,0)}}{\Delta_p, \sigma''_p \vdash \texttt{oav}(0) \xrightarrow{e_o} \top}\ \textsc{IdxSig}^2 \quad \dfrac{}{\top \in_c bool}\ \textsc{IsBool}}{\Delta, \Delta_p, \sigma_e, \sigma''_p \vdash (\texttt{oav}(0), id_{av}) \xrightarrow{mapop} \sigma'_{e0}}\ 1 \quad \dfrac{\vdots}{\Delta, \Delta_p, \sigma'_{e0}, \sigma''_p \vdash \begin{matrix}(\texttt{pauths},\texttt{open})\\(\texttt{rtt}(0), id_{rt}),(\texttt{marked}, id_m)\end{matrix} \xrightarrow{mapop} \sigma'_e}}{\Delta, \Delta_p, \sigma_e, \sigma''_p \vdash \begin{matrix}(\texttt{oav}(0), id_{av}),(\texttt{pauths},\texttt{open})\\(\texttt{rtt}(0), id_{rt}),(\texttt{marked}, id_m)\end{matrix} \xrightarrow{mapop} \sigma'_e}\ \textsc{MapopComp}$$

(1)
$$\begin{matrix}\Delta(id_{av}) = bool\\ \sigma_e = <\mathcal{S},\mathcal{C},\mathcal{E}>\\ \mathcal{S}' = \mathcal{S}(id_{av}) \leftarrow \top\\ \mathcal{E}' = \mathcal{E} \cup (\mathcal{S} \overset{\neq}{\cap} \mathcal{S}')\\ \sigma'_{e0} = <\mathcal{S}',\mathcal{C},\mathcal{E}'>\end{matrix}$$
(2)
$$\begin{matrix}\Delta_p(\texttt{oav}) = array(bool,0,0)\\ \sigma''_p(\texttt{oav}) = (\top)\\ \texttt{get\_at}(0,(\top)) = \top\end{matrix}$$

FIGURE 3.21: The evaluation of the output port map of the `place` component instance id_p.

Stabilization

A stabilization phase happens after the *runinit* phase during the initialization phase, but also after the rising edge phase and the falling edge phase in the course of a simulation cycle. The stabilization phase executes the combinational parts of the design's behavior. The `tl` design holds no combinational processes in its behavior. The `marked` and `fired` processes are both synchronous. To illustrate the execution of a combinational process during a stabilization phase, let us consider the `fired_evaluation` process defined in the behavior of the `transition` design. The `fired_evaluation` process will be executed with the internal behavior of the `transition` component instance id_t during the stabilization phase. The rule of Figure 3.22 presents the execution of the internal behavior of the `transition` component instance id_t. As shown, the internal behavior cs_t is executed three times before reaching a stable state. Here, the number of execution before stabilization is arbitrary. In Figure 3.22, σ_{t0} corresponds to the state of id_t after the *runinit* phase and after the evaluation of its input port map. Remember that the evaluation of the input port map of a component instance always precedes the execution of the internal behavior of the component. Since σ_{t0} and σ_{t1} are not stable states, it means that their event set is not empty. Thus, we have $\mathcal{E}(\sigma_{t0}) \neq \emptyset$ and $\mathcal{E}(\sigma_{t1}) \neq \emptyset$. On the contrary, σ_{t2} is a stable state, and thus, $\mathcal{E}(\sigma_{t2}) = \emptyset$.

$$\dfrac{\dfrac{\vdots}{\mathcal{D}_{\mathcal{H}}, \Delta_t, \sigma_{t0} \vdash cs_t \xrightarrow{comb} \sigma_{t1}} \qquad \dfrac{\dfrac{\vdots}{\mathcal{D}_{\mathcal{H}}, \Delta_t, \sigma_{t1} \vdash cs_t \xrightarrow{comb} \sigma_{t2}} \qquad \dfrac{\dfrac{\vdots}{\mathcal{D}_{\mathcal{H}}, \Delta_t, \sigma_{t2} \vdash cs_t \xrightarrow{comb} \sigma_{t2}}}{\mathcal{D}_{\mathcal{H}}, \Delta_t, \sigma_{t2} \vdash cs_t \rightsquigarrow \sigma_{t2}}\ \textsc{StabilizeEnd}}{\mathcal{D}_{\mathcal{H}}, \Delta_t, \sigma_{t0} \vdash cs_t \rightsquigarrow \sigma_{t2}}\ \textsc{StabilizeLoop}}{\mathcal{D}_{\mathcal{H}}, \Delta_t, \sigma_{t0} \vdash cs_t \rightsquigarrow \sigma_{t2}}\ \textsc{StabilizeLoop}$$

FIGURE 3.22: Three rounds of execution of the combinational parts of the `transition` component instance id_t during a stabilization phase.

Now, let us illustrate the execution of the `fired_evaluation` process, which is a part of cs_t, happening during the stabilization phase. The rule of Figure 3.23 details the execution of the body of process `fired_evaluation` performed by the *comb* relation.

$$\dfrac{\dfrac{\dfrac{}{\Delta_t, \sigma_{t0} \vdash \texttt{sfa} \xrightarrow{e} \bot}\,\textsc{Sig} \quad \dfrac{}{\Delta_t, \sigma_{t0} \vdash \texttt{spc} \xrightarrow{e} \top}\,\textsc{Sig}}{\Delta_t, \sigma_{t0} \vdash \texttt{sfa and spc} \xrightarrow{e} \bot}\,\textsc{BoolBinOp} \quad \dfrac{}{\bot \in_c bool}\,\textsc{IsBool}}{\mathcal{D}_{\mathcal{H}}, \Delta_t, \sigma_{t0}, NoEv(\sigma_{t0}), \emptyset \vdash \texttt{fired} \Leftarrow \texttt{sfa and spc} \xrightarrow{ss_c} \sigma'_{t0}, \emptyset}\,\textsc{SigAssign}^2}{\mathcal{D}_{\mathcal{H}}, \Delta_t, \sigma_{t0} \vdash \begin{array}{l}\texttt{process}(\texttt{fired\_evaluation}, \{\texttt{sfa}, \texttt{spc}\}, \emptyset, \xrightarrow{\rightsquigarrow} \sigma'_{t0} \\ \texttt{fired} \Leftarrow \texttt{sfa and spc})\end{array}}\,\textsc{PsComb}^1$$

(1) $\Delta_t(\texttt{fired\_evaluation}) = \emptyset$

(2) $\Delta_t(\texttt{fired}) = bool$
$NoEv(\sigma_{t0}) = <\mathcal{S}, \mathcal{C}, \emptyset>$
$\mathcal{S}' = \mathcal{S}(\texttt{fired}) \leftarrow \bot$
$\sigma'_{t0} = <\mathcal{S}', \mathcal{C}, \mathcal{S} \overset{\neq}{\cap} \mathcal{S}'>$

FIGURE 3.23: The execution of the `fired_evaluation` process during a stabilization phase. The `fired_evaluation` process is defined in the `transition` design's behavior.

Let us consider that the value of the `fired` signal was $\top$ at state σ_{t0}, i.e. $\sigma_{t0}(\texttt{fired}) = \top$. Then, since $\sigma'_{t0}(\texttt{fired}) = \bot$, we have $\mathcal{S} \overset{\neq}{\cap} \mathcal{S}' = \{\texttt{fired}\}$. When state σ'_{t0} will be merged with the other states generated by the concurrent execution of processes defining the `transition` design's behavior, the resulting merged state will have a non-empty set of events. Thus, another round of execution will be triggered. A stable state has been reached when the execution of the combinational parts of the behavior does not generate any event anymore.

Simulation cycle and clock phases

We describe here the execution of the `tl` design over one clock cycle. After the initialization phase, the design under simulation will execute τ simulation cycles, where τ is a natural numbers passed as a parameter. In the rule of Figure 3.24, τ equals 1. Thus, the behavior of the `tl` design is executed during one clock cycle and then the simulation ends. In Figure 3.24, σ_0 is the initial simulation state, i.e. the one at the end of the initialization phase. One simulation cycle yields two states σ_1 and σ_2, where σ_1 is the state in the middle of the first cycle, and σ_2 is the state at the end of the first cycle. The resulting simulation trace is only composed of states σ_1 and σ_2.

$$\dfrac{\dfrac{\vdots}{\mathcal{D}_{\mathcal{H}}, E_p, \Delta, 1, \sigma_0 \vdash cs_{tl} \xrightarrow{\uparrow,\downarrow} \sigma_1, \sigma_2}\ \textsc{SimCyc} \qquad \dfrac{}{\mathcal{D}_{\mathcal{H}}, E_p, \Delta, 0, \sigma_2 \vdash cs_{tl} \rightarrow []}\ \textsc{SimEnd}}{\mathcal{D}_{\mathcal{H}}, E_p, \Delta, 1, \sigma_0 \vdash cs_{tl} \rightarrow (\sigma_1 :: \sigma_2 :: [])}\ \textsc{SimLoop}^1$$

(1) $1 > 0$

FIGURE 3.24: The execution of the `tl` design's behavior during one clock cycle.

The rule of Figure 3.25 zooms in the first simulation cycle. The state σ_1 is produced by a rising edge phase followed by a stabilization phase. The state σ_2 is produced by a falling edge phase followed by a stabilization phase. The value of the primary input ports of the `tl` design are updated before each clock event phase. States σ_{0i} and σ_{1i} are the new states obtained after the update of the primary input port values. The update corresponds to the capture of values yielded by the given simulation environment E_p. The `tl` design has only one primary input port, i.e. the input port id_{c0}. The value of id_{c0} at state σ_{0i} is equal to the value yielded by the environment E_p at the rising edge of clock during the first clock cycle. Thus, we have $\sigma_{0i}(id_{c0}) = E_p(1, \uparrow)(id_{c0})$. To perform the proof that the HILECOP transformation is semantic preserving, we need the hypothesis that the primary input ports of the top-level design stay stable during a clock cycle. For instance, the value of id_{c0} must be the same during the first clock cycle independently of the considered clock event. Thus, we have $E_p(1, \uparrow)(id_{c0}) = E_p(1, \downarrow)(id_{c0})$. In this setting, we only need to capture the value of primary input ports at the beginning a clock cycle. However, to keep the definition of the $\mathcal{H}$-VHDL semantics as general as possible, we preserve the update of primary input ports at each clock event.

$$\dfrac{\dfrac{\vdots}{\mathcal{D}_{\mathcal{H}}, \Delta, \sigma_{0i} \vdash cs_{tl} \xrightarrow{\uparrow} \sigma_{\uparrow}} \quad \dfrac{\vdots}{\mathcal{D}_{\mathcal{H}}, \Delta, \sigma_{\uparrow} \vdash cs_{tl} \xrightarrow{\rightsquigarrow} \sigma_1} \quad \dfrac{\vdots}{\mathcal{D}_{\mathcal{H}}, \Delta, \sigma_{1i} \vdash cs_{tl} \xrightarrow{\downarrow} \sigma_{\downarrow}} \quad \dfrac{\vdots}{\mathcal{D}_{\mathcal{H}}, \Delta, \sigma_{\downarrow} \vdash \xrightarrow{\rightsquigarrow} \sigma_2}}{\mathcal{D}_{\mathcal{H}}, E_p, \Delta, 1, \sigma_0 \vdash \xrightarrow{\uparrow,\downarrow} \sigma_1, \sigma_2}\ \textsc{SimCyc}^1$$

(1) $\begin{matrix}\texttt{Inject}_{\uparrow}(\sigma_0, E_p, 1, \sigma_{0i}) \\ \texttt{Inject}_{\downarrow}(\sigma_1, E_p, 1, \sigma_{1i})\end{matrix}$

FIGURE 3.25: Details of the execution of the `tl` design's behavior during the first clock cycle.

The rule of Figure 3.26 describes the execution of the `fired` process, defined in the `tl` design's behavior, during the rising edge phase of the first simulation cycle. During the rising edge phase, rising blocks are executed. Thus, the $\uparrow$ relation triggers the execution of the rising block defined in the body of the `fired` process.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Delta, \sigma_{0i} \vdash id_{ft} \xrightarrow{e} \top}\,\textsc{Sig}^3 \quad \dfrac{}{\texttt{false} \xrightarrow{e} \bot}\,\textsc{False}}{\Delta, \sigma_{0i} \vdash id_{ft}\ \texttt{or}\ \texttt{false} \xrightarrow{e} \top}\,\textsc{BoolBinOp} \quad \dfrac{}{\top \in_c bool}\,\textsc{IsBool}}{\Delta, \sigma_{0i}, \sigma_{0i}, \emptyset \vdash id_{f0} \Leftarrow id_{ft}\ \texttt{or}\ \texttt{false} \xrightarrow{ss_{\uparrow}} \sigma'_{0i}, \emptyset}\,\textsc{SigAssign}^2}{\Delta, \sigma_{0i}, \sigma_{0i}, \emptyset \vdash \texttt{rising}\ (id_{f0} \Leftarrow id_{ft}\ \texttt{or}\ \texttt{false}) \xrightarrow{ss_{\uparrow}} \sigma'_{0i}, \emptyset}\,\textsc{RisingEdgeExec}}{\Delta, \sigma_{0i}, \sigma_{0i}, \emptyset \vdash (\texttt{rst}\ (id_{f0} \Leftarrow \texttt{false})(\texttt{rising}\ (id_{f0} \Leftarrow\ id_{ft}\ \texttt{or}\ \texttt{false}))) \xrightarrow{ss_{\uparrow}} \sigma'_{0i}, \emptyset}\,\textsc{RstDefault}}{\mathcal{D}_{\mathcal{H}}, \Delta, \sigma_{0i} \vdash \begin{array}{l}\texttt{process}\ (\texttt{fired}, \{\texttt{clk}\}, \emptyset, \\ (\texttt{rst}\ (id_{f0} \Leftarrow \texttt{false})(\texttt{rising}\ (id_{f0} \Leftarrow\ id_{ft}\ \texttt{or}\ \texttt{false})))\end{array} \xrightarrow{\uparrow} \sigma'_e}\,\textsc{PsRecClk}^1$$

(1) $\begin{array}{l}\texttt{clk} \in \{\texttt{clk}\} \\ \Delta(\texttt{fired}) = \emptyset\end{array}$ (2) $\begin{array}{l}\Delta(id_{f0}) = bool \\ \sigma_{0i} = <\mathcal{S}, \mathcal{C}, \mathcal{E}> \\ \mathcal{S}' = \mathcal{S}(id_{f0}) \leftarrow \top \\ \mathcal{E}' = \mathcal{E} \cup (\mathcal{S} \overset{\neq}{\cap} \mathcal{S}') \\ \sigma'_{0i} = <\mathcal{S}', \mathcal{C}, \mathcal{E}'>\end{array}$ (3) $\sigma_{0i}(id_{ft}) = \top$

FIGURE 3.26: The execution of the `fired` process during a rising edge phase. The `fired` process is a part of the `tl` design's behavior.

3.8 Implementation of the $\mathcal{H}$-VHDL syntax and semantics

This section presents the implementation of the $\mathcal{H}$-VHDL abstract syntax, and also of the elaboration and the simulation semantics of $\mathcal{H}$-VHDL designs with the Coq proof assistant. The full code is available under the `hvhdl` folder of the following repository: https://github.com/viampietro/ver-hilecop.

3.8.1 Implementation of the $\mathcal{H}$-VHDL abstract syntax, elaborated design and design state

$\mathcal{H}$-VHDL abstract syntax

The implementation of the $\mathcal{H}$-VHDL abstract syntax is naturally done leveraging the `Inductive` construct of the Coq proof assistant. The result is strictly similar to the formal definition of the abstract syntax given in Section 3.3. The reader can refer to the `AbstractSyntax.v` under the `hvhdl` folder for the details of the implementation.

Elaborated design

Listing 3.7 presents the implementation of the elaborated design structure (cf. Definition 14). Two definitions are involved in the implementation of the elaborated design structure. The first one defines the `SemanticObject` inductive type. Each constructor of this type corresponds

to a sub-environment of the elaborated design. For instance, the `Generic` constructor correspond to the couple $(type \times value)$ associated with a generic constant identifier in the *Gens* sub-environment of Definition 14. The `Process` constructor corresponds to the local variable environment associated with the process identifiers in the *Ps* sub-environment. A local variable environment is implemented by the `LEnv` type. The `LEnv` type is a map between identifiers and couples $(type \times value)$. Identifiers are implemented by the `ident` type, an alias of the `nat` type. The `type` and `value` types are the implementation of the semantic *type* and *value* presented in Table 3.2. The `ElDesign` type implements the elaborated design structure. It is an alias to the `IdMap SemanticObject` type. The `IdMap` is the type of maps from identifiers (i.e. belonging to the `ident` type) to instances of the type passed as a parameter. Here, the parameter is the `SemanticObject` type. Thus, an elaborated design is implemented as a map between identifiers and terms of the `SemanticObject` type. We leverage the `FMaps` module defined in the Coq standard library to implement the `IdMap` type. The `IdMap` type ensures that an identifier is only mapped once. Thus, the implementation of the elaborated design structure verifies that there are no intersection between the domains of sub-environments. For instance, a generic constant identifier can not be a input port identifier, and, as it is implemented, an identifier `id` can not be mapped to a `Generic` object and to an `Input` object in the same instance of `ElDesign`.

```
1  Inductive SemanticObject : Type :=
2  | Generic (t : type) (v : value)
3  | Input (t : type)
4  | Output (t : type)
5  | Declared (t : type)
6  | Process (lenv : LEnv)
7  | Component (Δc : IdMap SemanticObject).
8
9  Definition ElDesign := IdMap SemanticObject.
```

LISTING 3.7: The implementation of the elaborated design structure with the Coq proof assistant.

Design state

Listing 3.8 gives the implementation of the design state structure through the definition of the `DState` inductive type. The constructor of the `DState` type defines three fields: `sigstore`, implementing the signal store $\mathcal{S}$ of the design state, `compstore`, implementing the component store C, and `events`, implementing the set of events $\mathcal{E}$ of the design state. The `sigstore` field is a map from idenfifiers to values. The `compstore` field is a map from idenfifiers to design states, justifying the inductive definition of the `DState` type. The `events` field is an instance of the `IdSet` type. The `IdSet` is the type of sets of identifiers (i.e. sets of natural numbers). The `IdSet` type is defined leveraging the `MSets` module of the Coq standard library.

```
1  Inductive DState : Type := MkDState {
2    sigstore : IdMap value;
3    compstore : IdMap DState;
4    events   : IdSet;
```

```
5 }.
```

LISTING 3.8: The implementation of the design state structure with the Coq proof assistant.

3.8.2 Implementation of the elaboration phase

The design elaboration relation, as presented in Section 3.5.1, is implemented in Coq by the `edesign` relation. Listing 3.9 presents the definition of the `edesign` relation as an inductive type. As usual, a n-ary relation is implemented in Coq by a type defined with n parameters and projecting to the `Prop` type. The `edesign` relation as five parameters. The first parameter is the design store $\mathcal{D}$ of type `IdMap design`, i.e. a map from identifiers to $\mathcal{H}$-VHDL designs as defined by the abstract syntax. The second parameter is the dimensioning function $\mathcal{M}$ of type `IdMap value`, i.e. a map from identifiers to values. The third parameter is the design being elaborated, of type `design`. The fifth and sixth parameters are the elaborated design (of type `ElDesign` and the default design state (of type `DState`) resulting from the elaboration. In Listing 3.9, the `EDesign` constructor implements the DESIGNELAB rule presented in Section 3.5.1. From Line 7 to Line 10, the constructor defines the premises of Rule DESIGNELAB. The empty elaborated design structure, denoted $\Delta_\emptyset$, is implemented by the `EmptyElDesign` definition, and the empty design state structure, denoted by $\sigma_\emptyset$, is implemented by the `EmptyDState` definition. Line 13 implements the conclusion of Rule DESIGNELAB.

```
1  Inductive edesign (𝒟 : IdMap design) : IdMap value → design → ElDesign → DState → Prop :=
2  | EDesign :
3      forall ℳ id_e id_a gens ports sigs behavior
4             Δ Δ' Δ'' Δ''' σ σ' σ'',
5
6        (* Premises *)
7        egens EmptyElDesign ℳ gens Δ →
8        eports Δ EmptyDState ports Δ' σ →
9        edecls Δ' σ sigs Δ'' σ' →
10       ebeh 𝒟 Δ'' σ' behavior Δ''' σ'' →
11
12       (* Conclusion *)
13       edesign 𝒟 ℳ (design_ id_e id_a gens ports sigs behavior) Δ''' σ''
14
15 with ebeh (𝒟 : IdMap design) : ElDesign → DState → cs → ElDesign → DState → Prop :=
16 ...
```

LISTING 3.9: The implementation of the design elaboration relation with the Coq proof assistant.

The `edesign` relation necesitates a mutually recursive definition with the `ebeh` relation. The mutually recursive definition is performed leveraging the `with` clause at the end of Listing 3.9. The `ebeh` relation needs the `edesign` relation to elaborate the component instances found in the behavior of a design. Listing 3.10 gives the details of the `with` clause defining the `ebeh` relation.

At Line 2, the EBehPs constructor implements the Rule PsElab defining the elaboration of a process statement (cf. Section 3.5.6). Lines 6 and 7 implement the premises of the rule; the evars relation implements the elaboration of the local variable declaration list of the process; the validss relation implements the relation that type-checks the statement body of the process. Lines 10 to 14 implement the side conditions of the rule. The term ~NatMap.In id_p Δ implements the side condition $id_p \notin \Delta$. The NatMap.In id m relation states that a given identifier id is a key of the m map. At Line 13, the NatSet.In id_s sl term states that id_s belongs to the identifier set sl. At Line 14, the term MapsTo id_s (Input t) Δ states that the identifier id_s is mapped to Input t in the elaborated design Δ, i.e. $Ins(\Delta)(id_s) = \texttt{t}$. More generally, MapsTo is a ternary relation stating that a given key k of type nat, is mapped to a value v of a type A, in a given map m, i.e. Mapsto k v m. Line 17 implements the conclusion of Rule PsElab. The NatMap.add function binds the process identifier id_p to the term Process Λ in the elaborated design Δ, i.e. $\Delta \cup (id_p, \Lambda)$.

At Line 19, the EBehComp constructor implements the Rule CompElab (cf. Section 3.5.6). This rule describes the elaboration of a component instantiation statement. Lines 24 to 27 implement the premises of the rule. Line 25 appeals to the edesign relation to elaborate the cdesign design associated with the component instance id_c; thence, the mutually recursive definition with the ebeh relation. As it is stated at Line 32, the cdesign design is associated to identifier id_e, i.e. the entity identifier of component instance id_c, in the design store $\mathcal{D}$. Lines 30 to 33 implement the side conditions of the rule. Line 30 checks that the identifier id_c is not already bound to a semantic object in the elaborated design Δ. Line 31 checks that the identifier id_c is not already bound in the component store of σ. Line 33 checks that all identifiers defined in the domain of map $\mathcal{M}$, i.e. the dimensioning function, are bound to generic constants in the elaborated design Δ_c (i.e. $\mathcal{M} \subseteq Gens(\Delta_c)$). Lines 36 to 38 implement the conclusion of Rule CompElab. At Line 39, the cstore_add function binds id_c to design state σ_c in the component store of state σ and returns the resulting state.

At Line 41, the EBehNull constructor implements Rule CsNullElab. At Line 43, the EBehPar constructor implements Rule CsParElab.

```
1  with ebeh (𝒟 : IdMap design) : ElDesign → DState → cs → ElDesign → DState → Prop :=
2  | EBehPs :
3      forall id_p sl vars stmt Λ Δ σ,
4
5        (* Premises *)
6        evars Δ EmptyLEnv vars Λ →
7        validss Δ σ Λ stmt →
8
9        (* Side conditions *)
10       ~NatMap.In id_p Δ →
11
12       (forall id_s,
13           NatSet.In id_s sl →
14           exists t, MapsTo id_s (Declared t) Δ ∨ MapsTo id_s (Input t) Δ) →
15
16       (* Conclusion *)
17       ebeh 𝒟 Δ σ (cs_ps id_p sl vars stmt) (NatMap.add id_p (Process Λ) Δ) σ
```

```
18
19  | EBehComp :
20      forall Δ σ id_c id_e gmap ipmap opmap
21             M Δ_c σ_c formals actuals cdesign,
22
23        (* Premises *)
24        emapg (NatMap.empty value) gmap M →
25        edesign D M cdesign Δ_c σ_c →
26        validipm Δ Δ_c σ ipmap formals →
27        validopm Δ Δ_c opmap formals actuals →
28
29        (* Side conditions *)
30        ~NatMap.In id_c Δ →
31        ~NatMap.In id_c (compstore σ) →
32        MapsTo id_e cdesign D →
33        (forall g, NatMap.In g M → exists t v, MapsTo g (Generic t v) Δ_c) →
34
35        (* Conclusion *)
36        ebeh D Δ σ (cs_comp id_c id_e gmap ipmap opmap)
37                   (NatMap.add id_c (Component Δ_c) Δ)
38                   (cstore_add id_c σ_c σ)
39
40  | EBehNull: forall Δ σ, ebeh D Δ σ cs_null Δ σ
41
42  | EBehPar:
43      forall Δ Δ' Δ'' σ σ' σ'' cstmt cstmt',
44        ebeh D Δ σ cstmt Δ' σ' →
45        ebeh D Δ' σ' cstmt' Δ'' σ'' →
46        ebeh D Δ σ (cs_par cstmt cstmt') Δ'' σ''.
```

LISTING 3.10: The implementation of the ebeh behavior elaboration relation with the Coq proof assistant.

3.8.3 Implementation of the simulation algorithm

The full simulation relation (cf. Section 3.6.1) formalizes the $\mathcal{H}$-VHDL simulation algorithm. The Coq implementation of the full simulation relation, presented in Listing 3.11, is a strict translation of Rule FULLSIM. At Lines 14 and 15, the term (behavior d) represents the concurrent statements defining the behavior of the $\mathcal{H}$-VHDL design d (i.e. d.cs in the formal rule). Line 13 corresponds to the elaboration phase, Line 14 to the initialization phase, and Line 15 to the main simulation loop.

```
1  Inductive fullsim
2            (D : IdMap design)
3            (M : IdMap value)
4            (E_p : nat → Clk → IdMap value)
5            (τ : nat)
```

```
6           (Δ : ElDesign)
7           (d : design) : list DState → Prop :=
8
9   | FullSim :
10    forall σ_e σ_0 θ,
11
12      (* * Premises * *)
13      edesign D M d Δ σ_e →
14      init D Δ σ_e (behavior d) σ_0 →
15      simloop D E_p Δ σ_0 (behavior d) τ θ →
16
17      (* * Conclusion * *)
18      fullsim D M E_p τ Δ d (σ_0 :: θ).
```

LISTING 3.11: The implementation of the full simulation relation with the Coq proof assistant.

The `simloop` relation appeals to the `simcycle` that implements the simulation cycle relation defined in Section 3.6.3. Listing 3.12 presents the implementation of the `simcycle` relation. The `simcycle` relation is a strict transcription of the SIMCYC rule. At Line 13, the `vrising` relation implements the $\uparrow$ relation, i.e. the rising edge phase of the cycle. At Line 15, the `vfalling` relation implements the $\downarrow$ relation, i.e. the falling edge phase of the cycle. At Lines 14 and 16, the `stabilize` relation implements the $\rightsquigarrow$ relation, i.e. the stabilization phases of the simulation cycle. At Lines 18 and 19, the `IsInjectedDState` relation implements the $\texttt{Inject}_{\downarrow}$ and $\texttt{Inject}_{\uparrow}$ relations. Line 18 states that the σ_i state is the result of the injection of the map $(E_p\ \tau\ \uparrow)$ in the signal store of state σ.

```
1   Inductive simcycle (D : IdMap design) (E_p : nat →
2   Clk → IdMap value) (Δ : ElDesign) (τ : nat) (σ :
3   DState) (behavior : cs) (σ' σ'' : DState) : Prop := |
4   SimCycle : forall σ_i σ_↑ σ'_i
5   σ_↓,
6
7       (* * Premises * *)
8       vrising D Δ σ_i behavior σ_↑ →
9       stabilize D Δ σ_↑ behavior σ' →
10      vfalling D Δ σ'_i behavior σ_↓ →
11      stabilize D Δ σ_↓ behavior σ'' →
12
13      (* * Side conditions * *)
14      IsInjectedDState σ (E_p τ ↑) σ_i →
15      IsInjectedDState σ' (E_p τ ↓) σ'_i →
16
17      (* * Conclusion * *)
18      simcycle D E_p Δ τ σ behavior σ''.
```

LISTING 3.12: The implementation of the simulation cycle relation with the Coq proof assistant.

3.9 Conclusion

In this chapter, we gave an overview of the VHDL language and its informal simulation semantics. Then, considering our needs, that is considering the content of the VHDL programs generated by the HILECOP model-to-text transformation, we defined a synthesizable and synchronous subset of the VHDL language called $\mathcal{H}$-VHDL. We gave a small-step semantics to $\mathcal{H}$-VHDL by formalizing a simplified simulation algorithm. The simulation algorithm yields a simulation trace, i.e. time-ordered list of states, corresponding to the execution of the behavior of a $\mathcal{H}$-VHDL design over multiple clock cycles. The formalization of the $\mathcal{H}$-VHDL semantics also includes the formalization of the design elaboration. The elaboration, prior to the simulation, ensures the well-formedness and the well-typedness of a $\mathcal{H}$-VHDL design. Moreover, we have implemented the $\mathcal{H}$-VHDL syntax and semantics with the Coq proof assistant.

Ever since the mechanization of the proof of behavior preservation has begun, the semantics of $\mathcal{H}$-VHDL has been evolving. Section 3.3, 3.4, 3.5 and 3.6 present the most recent version of the semantics. However, it will probably be evolving again. With regards to our proof task, we realized that an operational semantics close to the simulation algorithm carried a lot of elements that were of no use. These elements could sometimes complexify the proof. For instance, in the VHDL simulation algorithm, the body of a process is executed during the stabilization phase only if one signal of its sensitivity list is part of the current state's event set. However, it is through the execution of the body of a process with the rules of the $\mathcal{H}$-VHDL semantics that we can determine the *combinational* equation associated with the value of a signal. In the proceeding of the proof of semantic preservation, we must often describe the value of an output signal with regards to the value of its input, or *source*, signals (cf. Section 4.4). Due to the event-based system of resuming a process activity, a combinational process could sometimes never be executed during a stabilization phase. Then, we are not able to determine the value of signals. We had to carry extra hypotheses in the definition of our lemmas to deal with this problem. Finally, our current semantics always executes the body combinational processes during a stabilization phase, and this greatly simplifies the proof. By doing this kind of simplification, we realized that we were heading toward a semantics that was closer to the "synthesis" semantics we talked about at the beginning of the chapter. This semantics tends to get closer to the combinational logic and the synchronous logic rules. These rules that a hardware system designer has in mind when devising a model with a hardware description language.

In the future, we contrive to improve the implementation of the $\mathcal{H}$-VHDL semantics with more dependent types. Especially, the elaborated design and the design state structure are formally defined with intentional subsets. These subsets could be easily implemented with the `sig` type of the Coq proof assistant. Also, we plan to improve the formalization of the elaboration phase with a global lookup of multiply-driven signals.

Chapter 4

Proving semantic preservation in HILECOP

In this chapter, we present our semantic preservation theorem (or behavior preservation theorem, both denomination are equivalent) along with its informal "paper" proof. The written proof is about a hundred-page long after compilation of the LaTeX files. 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 C. The theorems and lemmas presented in this chapter will be referring to the lemmas of Appendix C. The structure of this chapter is as follows: in Section 4.1, we present our review of the literature pertaining to the proof of semantic preservation theorems for transformation functions; in Section 4.2, we detail our state similarity relation, i.e. the semantic bond between an SITPN and its $\mathcal{H}$-VHDL translation; in Section 4.3, we draw out our behavior preservation theorem; in Section 4.4, we detail a particular point of the proof related to the SITPN firing process, and leverage the opportunity to demonstrate our proof strategy; 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 4.5, we present some points of the mechanization of the proof with the Coq proof assistant.

4.1 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 possess 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, i.e, what does one mean by semantic preservation, and in seeking usual proof strategies.

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 [36], X.Leroy presents the two points of major importance to express semantic preservation theorems for GPL 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 to 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 4.1, excerpt from [36], shows the different kinds of simulation diagrams possibly relating two programs together.

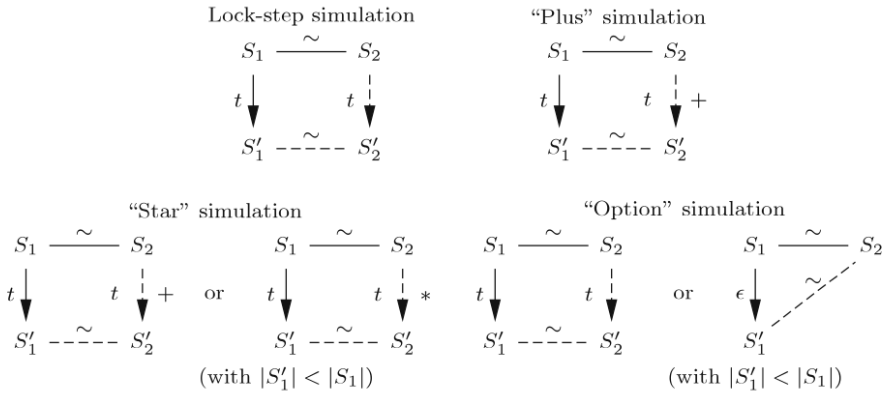


FIGURE 4.1: Simulation diagrams relating the execution of a source program to the execution of a target program; S_1 and S_2 are the initial states of the source and the target program, and S_1' and S_2' are the final states of the source and target program, i.e. the states resulting of the execution 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 can exhibit a given program. 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 18.

Definition 18 (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. The other kind of task that can be applied to certify a compiler is to perform compiler validation. Compiler validation is interested in generating a proof of behavior preservation (or a counter-example showing that behaviors diverge) for a given input program alongside the compilation process. Thus, for a given input program, the compiler yields a target program and the proof that the input and target have the same behavior. Exhibiting a theorem of semantic preservation is stronger than building a proof of semantic preservation for each input program. Therefore, compiler verification is stronger than compiler validation. The thesis aims to perform compiler *verification* over the HILECOP methodology.

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?

4.1.1 Compilers for generic programming languages

Taking the CompCert compiler as an example, the compilation pass from Clight programs to Cminor programs is described in [6, 36]. 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 (of course, we are dealing with programming languages). 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). About 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 that 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. 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 to perform case analysis on the translation function.

The pattern to compiler verification for GPLs is more or less the same as presented above. May it be compilers for imperative languages [36, 49], or compilers for functional languages [16, 50], 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 programs in [36]). As a consequence, the complexity of the memory model comparison relation increases.

4.1.2 Compilers for hardware description languages

In the case of HDL 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; indeed, 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 [8, 10], 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 [8], 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 18, 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 circuit after one clock cycle leads to equal register stores.

In [9], the source language is a subset of Lustre and the target language is an imperative language called Obc. A Lustre program is composed of nodes; each node treats a set of input streams and publishes output streams after the computation of its statement body. In its statement body, a Lustre node possibly refers to instances of other nodes. In the compilation process, each Lustre node is translated into an Obc class. An Obc class holds a vector of variables composing its internal memory and a vector of other Obc class instances. The authors define a data flow semantics for the Lustre language; judgments of the semantics describe how output streams are computed based on input streams. Furthermore, as we are dealing with hardware circuits, the semantics rules cover synchronous statements and combinational ones. 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 [38], 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 semantics of netlists is set as a big-step operational semantics through an interpreter that runs a netlist list 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.

In [56], 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 [30], 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.

4.1.3 Model transformations

Regarding model transformations, a lot of works consider semantic preservation as the preservation of structural properties in the transformed model [3, 14, 39].

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 4.1).

In [18] and [55], the authors are interested in giving a translational semantics to a given model having itself a reference execution semantics. In [18], 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 4.2.

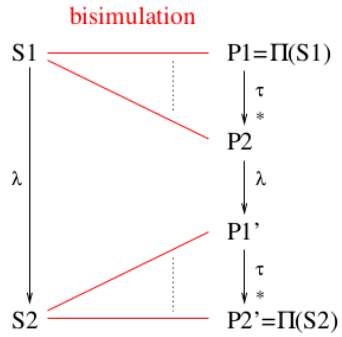


FIGURE 4.2: Bisimulation diagram relating an xSpem model execution and a Petri net execution

In Figure 4.2, 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 [55], 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 [26], 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.

4.1.4 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 proceedings are 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 18. 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 around 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 permits it, 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.

4.2 The state similarity relation

Before presenting our behavior preservation theorem, we must clarify the meaning of semantic preservation between an SITPN and a $\mathcal{H}$-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 a $\mathcal{H}$-VHDL design, does this semantic similarity must 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). A $\mathcal{H}$-VHDL design state is composed 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 generated alongside the transformation from an SITPN to a $\mathcal{H}$-VHDL design, we are able to relate the

elements of the SITPN structure to the component instance states and signal values of the $\mathcal{H}$-VHDL design state. Thus, the state similarity relation, depending on a γ binder and expressing a semantic match between an SITPN state and a $\mathcal{H}$-VHDL design, is defined as follows:

Definition 19 (General state 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* $\gamma \in WM(sitpn, d)$*, an SITPN state* $s \in S(sitpn)$ *and a design state* $\sigma \in \Sigma(\Delta)$ *are similar, written* $\gamma \vdash s \sim \sigma$ *iff*

1. $\forall p \in P, id_p \in Comps(\Delta)\ s.t.\ \gamma(p) = id_p,\ s.M(p) = \sigma(id_p)(\text{"s\_marking"})$.

2. $\forall t \in T_i, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t,$
   $\big(upper(I_s(t)) = \infty \wedge s.I(t) \leq lower(I_s(t)) \Rightarrow s.I(t) = \sigma(id_t)(\text{"s\_time\_counter"})\big)$
   $\wedge\big(upper(I_s(t)) = \infty \wedge s.I(t) > lower(I_s(t)) \Rightarrow \sigma(id_t)(\text{"s\_time\_counter"}) = lower(I_s(t))\big)$
   $\wedge\big(upper(I_s(t)) \neq \infty \wedge s.I(t) > upper(I_s(t)) \Rightarrow \sigma(id_t)(\text{"s\_time\_counter"}) = upper(I_s(t))\big)$
   $\wedge\big(upper(I_s(t)) \neq \infty \wedge s.I(t) \leq upper(I_s(t)) \Rightarrow s.I(t) = \sigma(id_t)(\text{"s\_time\_counter"})\big)$.

3. $\forall t \in T_i, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t, s.reset_t(t) = \sigma(id_t)(\text{"s\_reinit\_time\_counter"})$.

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 Item 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. Items 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 Item 2, the relation between the value of a time counter and the value of the `s_time_counter` signal is a 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 points of Item 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 overreached 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 does the state similarity relation must 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 4.4), the state similarity relation of Definition 4.2 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 (Item 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 Item 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 Item 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 20 (Post rising edge state 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 $\gamma \in WM(sitpn, d)$, an SITPN state $s \in S(sitpn)$ and a design state $\sigma \in \Sigma(\Delta)$ are similar after a rising edge happening, written $\gamma \vdash s \overset{\uparrow}{\sim} \sigma$ iff*

1. $\forall p \in P, id_p \in Comps(\Delta)\ s.t.\ \gamma(p) = id_p,\ s.M(p) = \sigma(id_p)(\text{"s\_marking"})$.

2. $\forall t \in T_i, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t,$
$\big(upper(I_s(t)) = \infty \wedge s.I(t) \leq lower(I_s(t)) \Rightarrow s.I(t) = \sigma(id_t)(\text{"s\_time\_counter"})\big)$
$\wedge\big(upper(I_s(t)) = \infty \wedge s.I(t) > lower(I_s(t)) \Rightarrow \sigma(id_t)(\text{"s\_time\_counter"}) = lower(I_s(t))\big)$
$\wedge\big(upper(I_s(t)) \neq \infty \wedge s.I(t) > upper(I_s(t)) \Rightarrow \sigma(id_t)(\text{"s\_time\_counter"}) = upper(I_s(t))\big)$
$\wedge\big(upper(I_s(t)) \neq \infty \wedge s.I(t) \leq upper(I_s(t)) \Rightarrow s.I(t) = \sigma(id_t)(\text{"s\_time\_counter"})\big)$.

3. $\forall t \in T_i, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t, s.reset_t(t) = \sigma(id_t)(\text{"s\_reinit\_time\_counter"})$.

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 20 is similar to Definition 19 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 Item 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 a $\mathcal{H}$-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 21 (Full post rising edge state 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 $\gamma \in WM(sitpn, d)$, 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 $\sigma \in \Sigma(\Delta)$ are fully similar after a rising edge happening at clock cycle count τ, written $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$ iff $\gamma \vdash s \overset{\uparrow}{\sim} \sigma$ (Definition 20) and*

1. $\forall t \in T, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t, t \in Sens(s.M) \Leftrightarrow \sigma(id_t)(\text{"s\_enabled"}) = \texttt{true}$.

2. $\forall t \in T, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t, t \notin Sens(s.M) \Leftrightarrow \sigma(id_t)(\text{"s\_enabled"}) = \texttt{false}$.

3. $\forall t \in T, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t,$
$$\sigma(id_t)(\text{"s\_condition\_combination"}) = \prod_{c \in conds(t)} \begin{cases} E_c(\tau, c) & if\ \mathbb{C}(t,c) = 1 \\ \texttt{not}(E_c(\tau, c)) & if\ \mathbb{C}(t,c) = -1 \end{cases}$$
where $conds(t) = \{c \in \mathcal{C} \mid \mathbb{C}(t,c) = 1 \vee \mathbb{C}(t,c) = -1\}$.

Definition 21 extends Definition 20 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*.

Now, let us define the state similarity relation describing how things must be compared between an SITPN state and a $\mathcal{H}$-VHDL design state after the falling edge of a clock signal:

Definition 22 (Post falling edge state 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 $\gamma \in WM(sitpn, d)$, an SITPN state $s \in S(sitpn)$ and a design state $\sigma \in \Sigma(\Delta)$ are similar after a falling edge, written $\gamma \vdash s \overset{\downarrow}{\sim} \sigma$ iff*

1. $\forall p \in P, id_p \in Comps(\Delta)\ s.t.\ \gamma(p) = id_p,\ s.M(p) = \sigma(id_p)(\text{"s\_marking"})$.
2. $\forall t \in T_i, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t,$
   $\big(upper(I_s(t)) = \infty \wedge s.I(t) \leq lower(I_s(t)) \Rightarrow s.I(t) = \sigma(id_t)(\text{"s\_time\_counter"})\big)$
   $\wedge\big(upper(I_s(t)) = \infty \wedge s.I(t) > lower(I_s(t)) \Rightarrow \sigma(id_t)(\text{"s\_time\_counter"}) = lower(I_s(t))\big)$
   $\wedge\big(upper(I_s(t)) \neq \infty \wedge s.I(t) > upper(I_s(t)) \Rightarrow \sigma(id_t)(\text{"s\_time\_counter"}) = upper(I_s(t))\big)$
   $\wedge\big(upper(I_s(t)) \neq \infty \wedge s.I(t) \leq upper(I_s(t)) \Rightarrow s.I(t) = \sigma(id_t)(\text{"s\_time\_counter"})\big)$.
3. $\forall c \in \mathcal{C}, id_c \in Ins(\Delta)\ s.t.\ \gamma(c) = id_c,\ s.cond(c) = \sigma(id_c)$.
4. $\forall a \in \mathcal{A}, id_a \in Outs(\Delta)\ s.t.\ \gamma(a) = id_a,\ s.ex(a) = \sigma(id_a)$.
5. $\forall f \in \mathcal{F}, id_f \in Outs(\Delta)\ s.t.\ \gamma(f) = id_f,\ s.ex(f) = \sigma(id_f)$.

As explained above, Definition 22 is similar to Definition 19 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 23 (Full post falling edge state 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 $\gamma \in WM(sitpn, d)$, an SITPN state $s \in S(sitpn)$ and a design state $\sigma \in \Sigma(\Delta)$ are fully similar after a falling edge, written $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$ iff $\gamma \vdash s \overset{\downarrow}{\sim} \sigma$ (Definition 22) and*

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

Definition 23 extends Definition 22 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`

4.3 Behavior preservation theorem

In this section, we lay out 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.

4.3.1 Proof notations

To add some readability to our proofs, we use the following notations:

- The most recent framed box above the point of reading denotes the current pending goal (what we are currently trying to prove): $\boxed{\forall n \in \mathbb{N},\ n > 0 \vee n = 0}$
- A red framed box denotes a completed goal (i.e. equivalent to QED): `true = true`
- A green framed box denotes the current induction hypothesis:

  $\forall n \in \mathbb{N},\ n + 1 > 0$

- The mention **CASE** directly follows an item bullet to denote a case during a proof by case analysis.

During a proof, we constantly refer to the names of the constants and signals declared in the $\mathcal{H}$-VHDL place and transition designs. Some constants and signals have very long names, and therefore we use aliases to refer to them in the following proofs. Table C.1 gives the full correspondence between constants and signals, 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) identifier associated to a given condition c (resp. action a and function f) through $\gamma(c) = id_c$.

4.3.2 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 24 states that the execution environment for a given SITPN and the simulation environment for a given $\mathcal{H}$-VHDL design are similar.

Definition 24 (Similar environments)**.** *For a given $sitpn \in SITPN$, a $\mathcal{H}$-VHDL design $d \in design$, a design store $\mathcal{D} \in entity\text{-}id \rightharpoonup design$, an elaborated version $\Delta \in ElDesign(d, \mathcal{D})$ of design d, and a binder $\gamma \in WM(sitpn, d)$, the environment $E_p \in (\mathbb{N} \times \{\uparrow, \downarrow\}) \to Ins(\Delta) \to value$, that yields the value of the primary input ports of Δ at a given simulation cycle and a given clock event, and the environment E_c, that yields the value of conditions of $sitpn$ at a given execution cycle, are similar, noted $\gamma \vdash E_p \overset{env}{=} E_c$, iff for all $\tau \in \mathbb{N}$, $clk \in \{\uparrow, \downarrow\}$, $c \in \mathcal{C}$, $id_c \in Ins(\Delta)$ s.t. $\gamma(c) = id_c$, $E_p(\tau, clk)(id_c) = E_c(\tau)(c)$.*

Definition 24 also states that every input port of the top-level design related to a SITPN condition by the γ binder has a stable boolean value during a whole clock cycle. That is to say, in the context of Definition 24, there exists no id_c such that $E_p(\tau, \uparrow)(id_c) \neq E_p(\tau, \downarrow)(id_c)$.

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 25 permits to compare such traces.

Definition 25 (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 $\theta_\sigma \in \texttt{list}(\Sigma(\Delta))$ are similar, written $\gamma \vdash \theta_s \overset{clk}{\sim} \theta_\sigma$, where $clk \in \{\uparrow, \downarrow\}$, according to the following rules:*

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

In Definition 25, 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 falling edge, we give a slightly different definition of the execution trace similarity relation in Definition 26.

Definition 26 (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 $\theta_\sigma \in \texttt{list}(\Sigma(\Delta))$ are fully similar, written $\gamma \vdash \theta_s \sim \theta_\sigma$, according to the following rules:*

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

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.

4.3.3 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 SITPN (see Definition 13). 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 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 now, we choose to consider Theorems 2, 3 and 4 as axioms.

Theorem 1 (Behavior preservation). *For all well-defined $sitpn \in SITPN$, an $\mathcal{H}$-VHDL design $d \in design$, a binder $\gamma \in WM(sitpn, d)$, a clock cycle count $\tau \in \mathbb{N}$, a execution environment $E_c \in \mathbb{N} \to \mathcal{C} \to \mathbb{B}$ and an execution trace $\theta_s \in \texttt{list}(S(sitpn))$ s.t.*

- *SITPN $sitpn$ translates into $\mathcal{H}$-VHDL design d and yields a binder γ: $\lfloor sitpn \rfloor_{\mathcal{H}} = (d, \gamma)$*
- *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 exists an elaborated design $\Delta \in ElDesign(d, \mathcal{D}_{\mathcal{H}})$ s.t. for all simulation environment $E_p \in (\mathbb{N} \times \{\uparrow, \downarrow\}) \to Ins(\Delta) \to value$, verifying

- *Simulation/Execution environments are similar: $\gamma \vdash E_p \overset{env}{=} E_c$*

then there exists a simulation trace $\theta_\sigma \in \texttt{list}(\Sigma(\Delta))$ *s.t.*

- *Under the HILECOP design store* $\mathcal{D}_\mathcal{H}$ *and with an empty generic constant dimensioning function* $(\emptyset)$*, design d 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 $E_c \in \mathbb{N} \to \mathcal{C} \to \mathbb{B}$ and a $\theta_s \in \texttt{list}(S(sitpn))$, let us show that

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

Appealing to Theorems 2, 3 and 4, let us take an elaborated design $\Delta \in ElDesign(d, \mathcal{D}_\mathcal{H})$, two design states $\sigma_e, \sigma_0 \in \Sigma(\Delta)$, and a simulation trace $\theta_\sigma \in$ 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: $\mathcal{D}_\mathcal{H}, E_p, \Delta, \tau, \sigma_0 \vdash \mathrm{d.cs} \to \theta_\sigma$

By definition of the $\mathcal{H}$-VHDL full simulation relation, we have:

$$\begin{aligned} \mathcal{D}_\mathcal{H}, \Delta, \emptyset, E_p, \tau \vdash \mathrm{d} \xrightarrow{full} \theta_\sigma \equiv \exists \sigma_e, \sigma_0 \in \Sigma(\Delta),\ \mathcal{D}_\mathcal{H}, \emptyset \vdash d \xrightarrow{elab} (\Delta, \sigma_e) \\ \wedge \mathcal{D}_\mathcal{H}, \Delta, \sigma_e \vdash d.cs \xrightarrow{init} \sigma_0 \\ \wedge \mathcal{D}_\mathcal{H}, E_p, \Delta, \tau, \sigma_0 \vdash \mathrm{d.cs} \to \theta_\sigma \end{aligned} \tag{4.1}$$

Rewriting the goal with (4.1):

$$\boxed{\exists \Delta,\ \forall E_p,\ \gamma \vdash E_p \overset{env}{=} E_c,\ \exists \theta_\sigma, \sigma_e, \sigma_0 \text{ s.t. } \mathcal{D}_\mathcal{H}, \emptyset \vdash d \xrightarrow{elab} (\Delta, \sigma_e) \wedge \mathcal{D}_\mathcal{H}, \Delta, \sigma_e \vdash d.cs \xrightarrow{init} \sigma_0 \wedge \mathcal{D}_\mathcal{H}, E_p, \Delta, \tau, \sigma_0 \vdash \mathrm{d.cs} \to \theta_\sigma \wedge \theta_s \sim \theta_\sigma}$$

Let us use $\Delta, \sigma_e, \sigma_0 \in \Sigma(\Delta)$ and θ_σ to prove the goal:

$$\boxed{\mathcal{D}_\mathcal{H}, \emptyset \vdash d \xrightarrow{elab} (\Delta, \sigma_e) \wedge \mathcal{D}_\mathcal{H}, \Delta, \sigma_e \vdash d.cs \xrightarrow{init} \sigma_0 \wedge \mathcal{D}_\mathcal{H}, E_p, \Delta, \tau, \sigma_0 \vdash \mathrm{d.cs} \to \theta_\sigma \wedge \theta_s \sim \theta_\sigma}$$

We assumed the three first points of the goal, and the last point, i.e $\theta_s \sim \theta_\sigma$, is proved by appealing to Theorem 5.

□

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, i.e. code that could be the input to a simulator program.

Theorem 2 (Elaboration). *For all well-defined* $sitpn \in SITPN$, $d \in design$, $\gamma \in WM(sitpn, d)$ *s.t.*

- $\lfloor sitpn \rfloor_{\mathcal{H}} = (d, \gamma)$

then there exists an elaborated design $\Delta \in ElDesign(d, \mathcal{D}_{\mathcal{H}})$ *and a design state* $\sigma_e \in \Sigma(\Delta)$ *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.

Theorem 3 (Initialization). *For all well-defined* $sitpn \in SITPN$, $d \in design$, $\gamma \in WM(sitpn, d)$, $\Delta \in ElDesign(d, \mathcal{D}_{\mathcal{H}})$, $\sigma_e \in \Sigma(\Delta)$ *s.t.*

- $\lfloor sitpn \rfloor_{\mathcal{H}} = (d, \gamma)$ *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* $sitpn \in SITPN$, $d \in design$, $\gamma \in WM(sitpn, d)$, $\Delta \in ElDesign(d, \mathcal{D}_{\mathcal{H}})$, $\sigma_e, \sigma_0 \in \Sigma(\Delta)$ *s.t.*

- $\lfloor sitpn \rfloor_{\mathcal{H}} = (d, \gamma)$ *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 for all simulation environment $E_p \in (\mathbb{N} \times \{\uparrow, \downarrow\}) \rightarrow Ins(\Delta) \rightarrow value$, *and simulation cycle count* $\tau \in \mathbb{N}$, *there exists a simulation trace* $\theta_\sigma \in \texttt{list}(\Sigma(\Delta))$ *s.t.*

- *Design* d *yields the simulation trace* θ_σ *after* τ *simulation cycles, starting from initial state* σ_0: $\mathcal{D}_{\mathcal{H}}, E_p, \Delta, \tau, \sigma_0 \vdash \text{d.cs} \rightarrow \theta_\sigma$

4.3.4 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 26. In this thesis, we proved this particular theorem, and as said before, we left the proofs of Theorems 2, 3 and 4 for later. We choose 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 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 19); this is done by appealing to Lemma 5. The first execution cycle is particular because, by definition of the SITPN full execution relation, no transitions are fired during the first rising edge. Therefore, after the first rising edge, the SITPN state is still equal to its initial state s_0. We prove that the post rising edge similarity relation is verified after the first rising edge by appealing to Lemma 15. The detailled proofs for Lemmas 5 and 15 are given in Sections C.1 and C.2.

Theorem 5 (Full bisimulation). *For all* $sitpn \in SITPN$, $d \in design$, $\gamma \in WM(sitpn, d)$, $\tau \in \mathbb{N}$, $E_c \in \mathbb{N} \to \mathcal{C} \to \mathbb{B}$, $\theta_s \in \texttt{list}(S(sitpn))$, $\Delta \in ElDesign(d, \mathcal{D}_\mathcal{H})$, $E_p \in (\mathbb{N} \times \{\uparrow, \downarrow\}) \to Ins(\Delta) \to value$, $\theta_\sigma \in \texttt{list}(\Sigma(\Delta))$ *s.t.*

- $\lfloor sitpn \rfloor_\mathcal{H} = (d, \gamma)$
- $\gamma \vdash E_p \overset{env}{=} E_c$
- $E_c, \tau \vdash sitpn \xrightarrow{full} \theta_s$
- $\mathcal{D}_\mathcal{H}, \Delta, \emptyset, E_p, \tau \vdash \mathrm{d} \xrightarrow{full} \theta_\sigma$

then $\gamma \vdash \theta_s \sim \theta_\sigma$

Proof. Assuming the above hypotheses, let us show $\boxed{\gamma \vdash \theta_s \sim \theta_\sigma}$.
Let us perform case analysis on 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 $[\sigma_0]$, and by definition of the full execution trace similarity relation, what is left to prove is: $\boxed{\gamma \vdash s_0 \sim \sigma_0}$

Appealing to Lemma 5, 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 and θ_σ, the new goal is: $\boxed{\gamma \vdash (s_0 :: s_0 :: s :: \theta) \sim (\sigma_0 :: \theta')}$

  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:

  $E_p, \Delta, \tau, \sigma_0 \vdash d.cs \xrightarrow{\uparrow_\mathcal{N}\downarrow} \sigma, \sigma'$ and $E_p, \Delta, \tau - 1, \sigma' \vdash \mathrm{d.cs} \to \theta''$ and $\theta' = \sigma :: \sigma' :: \theta''$

  Rewriting θ', the new goal is: $\boxed{\gamma \vdash (s_0 :: s_0 :: s :: \theta) \sim (\sigma_0 :: \sigma :: \sigma' :: \theta'')}$

  By definition of the full execution trace similarity relation, there are four points to prove:

  1. $\boxed{\gamma \vdash s_0 \sim \sigma_0.}$ Appealing to Lemma 5, we can show $\gamma \vdash s_0 \sim \sigma_0$.
  2. $\boxed{\gamma, E_c, \tau \vdash s_0 \overset{\uparrow}{\sim} \sigma.}$ Appealing to Lemma 15, 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 15 and 3, 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. $\boxed{\gamma \vdash \theta \overset{\updownarrow}{\sim} \theta''.}$

     Appealing to Lemma 15 and 3, we have $\gamma \vdash s \overset{\downarrow}{\approx} \sigma'$.

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

□

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, $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$). 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 $sitpn$, d, γ, E_p, E_c, τ, s, θ_s, σ, θ_σ, Δ, σ_e, assume that:*

- $\lfloor sitpn \rfloor_{\mathcal{H}} = (d, \gamma)$ *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. Assuming the above hypotheses, let us show $\boxed{\gamma \vdash \theta_s \overset{\uparrow}{\sim} \theta_\sigma.}$ Let us reason by induction on 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**: $\tau > 0$.

  $\forall s, \sigma, \theta_s, \theta_\sigma$ 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 $\gamma \vdash \theta_s \overset{\uparrow}{\approx} \theta_\sigma$.

  By definition of the SITPN execution and the $\mathcal{H}$-VHDL simulation relations for $\tau > 0$, we have:

  - $E_c, \tau \vdash s \xrightarrow{\uparrow} s'$ and $E_c, \tau \vdash s' \xrightarrow{\downarrow} s''$ and $E_c, \tau - 1 \vdash sitpn, s'' \to \theta$.
  - $\texttt{Inject}_{\uparrow}(\sigma, E_p, \tau, \sigma_i)$ and $\Delta, \sigma_i \vdash$ d.cs $\xrightarrow{\uparrow} \sigma_{\uparrow}$ and $\Delta, \sigma_{\uparrow} \vdash$ d.cs $\xrightarrow{\rightsquigarrow} \sigma'$
  - $\texttt{Inject}_{\downarrow}(\sigma', E_p, \tau, \sigma'_i)$ and $\Delta, \sigma'_i \vdash$ d.cs $\xrightarrow{\downarrow} \sigma_{\downarrow}$ and $\Delta, \sigma_{\downarrow} \vdash$ d.cs $\xrightarrow{\rightsquigarrow} \sigma''$
  - $E_p, \Delta, \tau - 1, \sigma'' \vdash d.cs \to \theta'$.

  and $\theta_s = s' :: s'' :: \theta$ 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, 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. $\boxed{\gamma \vdash \theta \overset{\updownarrow}{\sim} \theta'.}$

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

   Appealing to Lemmas 3 and 2, we can show $\gamma \vdash s'' \overset{\downarrow}{\approx} \sigma''$.

□

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 4.3.

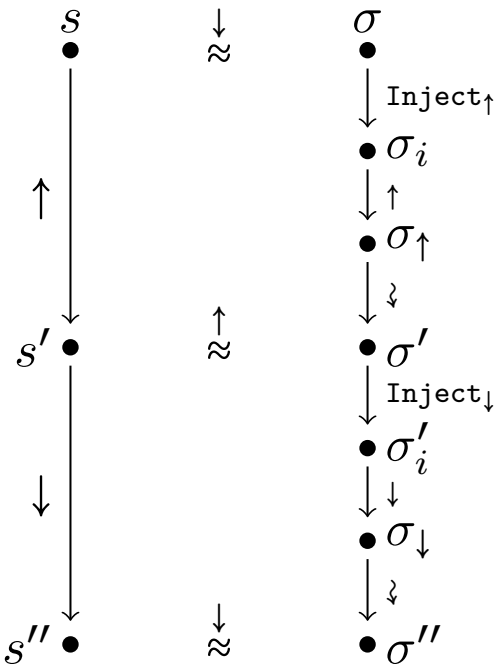


FIGURE 4.3: Bisimulation diagram over one clock cycle for a source SITPN and a target $\mathcal{H}$-VHDL design; the left part of the diagram presents the execution of an SITPN over one clock cycle, and the right part of the diagram presents the simulation of an $\mathcal{H}$-VHDL design over one clock 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 is proved by Lemma 2. 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 4.3). 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 is proved by Lemma 3. 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, $s' \overset{\downarrow}{\approx} \sigma'$ in Figure 4.3). Then, Lemma 2 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.

Here, we present Lemma 2 and Lemma 3, along with their proofs. In the two lemmas, we added an extra hypothesis 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 mandatory to determine the expression associated to combinational signals, i.e. the *combinational equations*, at the beginning of the clock phase (see Section 4.4 for more details about combinational equations).

To prove Lemmas 2 and 3, one must show 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 C. The proof strategy to show the equalities or equivalences laid out 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 $\texttt{Inject}_{clk}$, $\uparrow$, $\downarrow$ 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 4.4.

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

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

then $\gamma, E_c, \tau \vdash s' \overset{\uparrow}{\approx} \sigma'$.

Proof. By definition of the Full post rising edge state similarity relation, there are 8 points to prove:

1. $\forall p \in P, id_p \in Comps(\Delta)\ s'.t.\ \gamma(p) = id_p,\ s'.M(p) = \sigma'(id_p)(\text{"s\_marking"})$.
2. $\forall t \in T_i, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t,$
   $(upper(I_s(t)) = \infty \wedge s'.I(t) \leq lower(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(\text{"s\_time\_counter"}))$
   $\wedge (upper(I_s(t)) = \infty \wedge s'.I(t) > lower(I_s(t)) \Rightarrow \sigma'(id_t)(\text{"s\_time\_counter"}) = lower(I_s(t)))$
   $\wedge (upper(I_s(t)) \neq \infty \wedge s'.I(t) > upper(I_s(t)) \Rightarrow \sigma'(id_t)(\text{"s\_time\_counter"}) = upper(I_s(t)))$
   $\wedge (upper(I_s(t)) \neq \infty \wedge s'.I(t) \leq upper(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(\text{"s\_time\_counter"}))$.
3. $\forall t \in T_i, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t, s'.reset_t(t) = \sigma'(id_t)(\text{"s\_reinit\_time\_counter"})$.
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. $\forall t \in T, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t, t \in Sens(s'.M) \Leftrightarrow \sigma'(id_t)(\text{"s\_enabled"}) = \texttt{true}$.
7. $\forall t \in T, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t, t \notin Sens(s'.M) \Leftrightarrow \sigma'(id_t)(\text{"s\_enabled"}) = \texttt{false}$.
8. $\forall t \in T, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t,$
   $$\sigma'(id_t)(\text{"s\_condition\_combination"}) = \prod_{c \in conds(t)} \begin{cases} E_c(\tau, c) & if\ \mathbb{C}(t,c) = 1 \\ \texttt{not}(E_c(\tau, c)) & if\ \mathbb{C}(t,c) = -1 \end{cases}$$
   where $conds(t) = \{c \in \mathcal{C} \mid \mathbb{C}(t,c) = 1 \vee \mathbb{C}(t,c) = -1\}$.

Each point is proved by a separate lemma:

- Apply the Rising edge equal marking lemma to solve 1.
- Apply the Rising edge equal time counters lemma to solve 2.
- Apply the Rising edge equal reset orders lemma to solve 3.
- Apply the Rising edge equal action executions lemma to solve 4.
- Apply the Rising edge equal function executions lemma to solve 5.
- Apply the Rising edge equal sensitized lemma to solve 6.
- Apply the Rising edge equal not sensitized lemma to solve 7.
- Apply the Rising edge equal condition combination lemma to solve 8.

□

Lemma 3 (Falling edge). *For all* $sitpn \in SITPN$, $d \in design$, $\gamma \in WM(sitpn, d)$, $E_c \in \mathbb{N} \to \mathcal{C} \to \mathbb{B}$, $\Delta \in ElDesign(d, \mathcal{D}_{\mathcal{H}})$, $E_p \in (\mathbb{N} \times \{\uparrow, \downarrow\}) \to Ins(\Delta) \to value$, $\tau \in \mathbb{N}$, $s, s' \in S(sitpn)$, $\sigma_e, \sigma, \sigma_i, \sigma_\downarrow, \sigma' \in \Sigma(\Delta)$, *assume that:*

- $\lfloor sitpn \rfloor_{\mathcal{H}} = (d, \gamma)$ *and* $\gamma \vdash E_p \overset{env}{=} E_c$ *and* $\mathcal{D}_{\mathcal{H}}, \emptyset \vdash \mathrm{d} \xrightarrow{elab} \Delta, \sigma_e$
- $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$
- $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$
- $\texttt{Inject}_{\downarrow}(\sigma, E_p, \tau, \sigma_i)$ *and* $\Delta, \sigma_i \vdash \mathrm{d.cs} \xrightarrow{\downarrow} \sigma_{\downarrow}$ *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. $\forall p \in P, id_p \in Comps(\Delta)\ s.t.\ \gamma(p) = id_p,\ s'.M(p) = \sigma'(id_p)(\text{"s\_marking"})$.
2. $\forall t \in T_i, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t,$
   $\big(upper(I_s(t)) = \infty \wedge s'.I(t) \leq lower(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(\text{"s\_time\_counter"})\big)$
   $\wedge \big(upper(I_s(t)) = \infty \wedge s'.I(t) > lower(I_s(t)) \Rightarrow \sigma'(id_t)(\text{"s\_time\_counter"}) = lower(I_s(t))\big)$
   $\wedge \big(upper(I_s(t)) \neq \infty \wedge s'.I(t) > upper(I_s(t)) \Rightarrow \sigma'(id_t)(\text{"s\_time\_counter"}) = upper(I_s(t))\big)$
   $\wedge \big(upper(I_s(t)) \neq \infty \wedge s'.I(t) \leq upper(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(\text{"s\_time\_counter"})\big)$.
3. $\forall c \in \mathcal{C}, id_c \in Ins(\Delta)\ s.t.\ \gamma(c) = id_c,\ s'.cond(c) = \sigma'(id_c)$.
4. $\forall a \in \mathcal{A}, id_a \in Outs(\Delta)\ s.t.\ \gamma(a) = id_a,\ s'.ex(a) = \sigma'(id_a)$.
5. $\forall f \in \mathcal{F}, id_f \in Outs(\Delta)\ s.t.\ \gamma(f) = id_f,\ s'.ex(f) = \sigma'(id_f)$.
6. $\forall t \in T, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t, t \in Firable(s') \Leftrightarrow \sigma'(id_t)(\text{"s\_firable"}) = \texttt{true}$.
7. $\forall t \in T, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t, t \notin Firable(s') \Leftrightarrow \sigma'(id_t)(\text{"s\_firable"}) = \texttt{false}$.
8. $\forall t \in T, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t, t \in Fired(s') \Leftrightarrow \sigma'(id_t)(\text{"fired"}) = \texttt{true}$.
9. $\forall t \in T, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t, t \notin Fired(s') \Leftrightarrow \sigma'(id_t)(\text{"fired"}) = \texttt{false}$.
10. $\forall p \in P, id_p \in Comps(\Delta)\ s.t.\ \gamma(p) = id_p, \sum_{t \in Fired(s')} pre(p,t) = \sigma'(id_p)(\text{"s\_output\_token\_sum"})$.
11. $\forall p \in P, id_p \in Comps(\Delta)\ s.t.\ \gamma(p) = id_p, \sum_{t \in Fired(s')} post(t,p) = \sigma'(id_p)(\text{"s\_input\_token\_sum"})$.

Each point is proved by a separate lemma:

– Apply the Falling edge equal marking lemma to solve 1.

– Apply the Falling edge equal time counters lemma to solve 2.

– Apply the Falling edge equal condition values lemma to solve 3.

– Apply the Falling edge equal action executions lemma to solve 4.

– Apply the Falling edge equal function executions lemma to solve 5.

– Apply the Falling edge equal firable lemma to solve 6.

– Apply the Falling edge equal not firable lemma to solve 7.

– Apply the Falling edge equal fired lemma to solve 8.

– Apply the Falling Edge Equal Not Fired lemma to solve 9.

– Apply the Falling Edge Equal Output Token Sum lemma to solve 10.

– Apply the Falling Edge Equal Input Token Sum lemma to solve 11.

□

4.4 A detailled proof: equivalence of fired transitions

The goal of this section is to present the overall proof strategy to establish the semantic preservation property. We use the proof of the 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.

4.4.1 An accompanied journey along the proof

The proof of Lemma 4 pertains 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. Therefore, 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. 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 27 (Falling edge hypotheses)**.** *Given a* $sitpn \in SITPN, d \in design, \gamma \in WM(sitpn, d)$, $E_c \in \mathbb{N} \to \mathcal{C} \to \mathbb{B}$, $\Delta \in ElDesign(d, \mathcal{D}_{\mathcal{H}})$, $E_p \in (\mathbb{N} \times \{\uparrow, \downarrow\}) \to Ins(\Delta) \to value$, $\tau \in \mathbb{N}$, $s, s' \in S(sitpn)$, $\sigma_e, \sigma, \sigma_i, \sigma_\downarrow, \sigma' \in \Sigma(\Delta)$, *assume that:*

- *SITPN sitpn translates into $\mathcal{H}$-VHDL design d and yields a binder γ:* $\lfloor sitpn \rfloor_{\mathcal{H}} = (d, \gamma)$
- *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':* $\texttt{Inject}_{\downarrow}(\sigma, E_p, \tau, \sigma_i)$ *and* $\Delta, \sigma_i \vdash \mathrm{d.cs} \xrightarrow{\downarrow} \sigma_{\downarrow}$ *and* $\Delta, \sigma_{\downarrow} \vdash \mathrm{d.cs} \overset{\rightsquigarrow}{\longrightarrow} \sigma'$
- *State σ is a stable design state:* $\mathcal{D}_{\mathcal{H}}, \Delta, \sigma \vdash \mathrm{d.cs} \xrightarrow{comb} \sigma$

The hypotheses of Definition 27 are used in all the lemmas expressing some properties about the falling edge phase. Therefore, Definition 27 enables the conciser expression of these lemmas. Then, we can express Lemma Falling edge equal fired:

Lemma 4 (Falling edge equal fired)**.** *For all sitpn, d, γ, Δ, σ_e, E_c, E_p, τ, s, s′, σ, σ_i, $\sigma_{\downarrow}$, σ' that verify the hypotheses of Definition 27, then* $\forall t \in T, id_t \in Comps(\Delta)$ *s.t.* $\gamma(t) = id_t$, $t \in Fired(s') \Leftrightarrow \sigma'(id_t)(\text{"}fired\text{"}) = \texttt{true}$.

Then, 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: $\boxed{t \in Fired(s') \Leftrightarrow \sigma'(id_t)(\text{"}fired\text{"}) = \texttt{true}.}$

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 receives an intentional and recursive definition (see Definition 5) 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 (which will also be the state at the beginning of the next rising edge phase). A transition belongs to the set of fired transitions if it is *firable* (see Definition 4) and sensitized by the *residual* marking at the considered SITPN state. Figure 4.4 gives the set of fired transitions, i.e. $Fired(s)$, for an example 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.

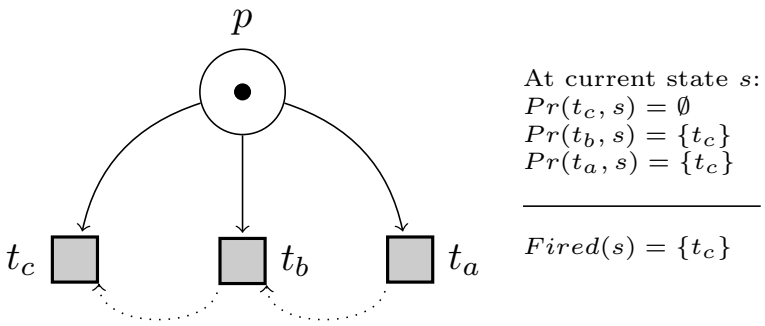


FIGURE 4.4: The set of fired transitions for an example SITPN at a given SITPN state s; on the right side, the dotted arrows indicates the priority relation between the three transitions (t_c is the top-priority transition); on the left side, each transition is associated to its Pr set which are necessary 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 4.4). 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 13). By definition, top-priority transitions have an empty Pr set. Indeed, there exist 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 can not 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 4.4, it is impossible to know the content of $Pr(t_a)$ 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 $Pr(t_b)$. 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 hints at 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 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 the 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 4.5 shows a part of the transition design architecture describing how the `fired` port is connected to the other internal signals.

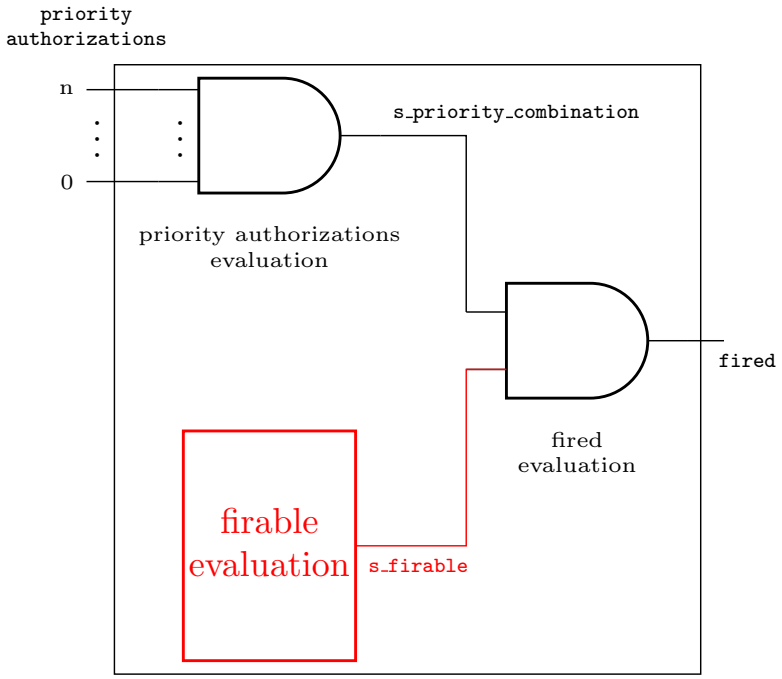


FIGURE 4.5: Wiring of the `fired` output port in the transition design architecture; on the left side is the input interface of the transition design; on the right side is the output interface of the transition design, with the `fired` port; in red are the parts of the architecture that depend on synchronous logic and in black are the parts that are purely combinational.

In Figure 4.5, 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 4.5 is a 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 combinational equation hold when the signal values are stable), the `fired` port equation at σ is:

$$\sigma(id_t)(''fired'') = \sigma(id_t)(''s\_firable'') \,.\, \sigma(id_t)(''s\_priority\_combination'') \tag{4.2}$$

Equation (4.2) states that the value of the `fired` port is a simple "and" expression[1] between the value of the internal signal `s_firable` and `s_priority_combination`.

Remark 7 (Signals and combinational equations)**.** *In the proceeding of the proof, a lot of combinational equations are established (e.g, Equation (4.2)). These equations relate the value of a given signal to the value of other signals or expressions. All these equations are deduced by running the $\mathcal{H}$-VHDL semantics rules on the internal behavior (i.e., the processes) of the transition and the place designs. A*

[1]To differentiate the formulas of the intuitionistic 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 intuitionistic formulas.

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 4.1, assigns the* `fired` *output port. Reasoning on the* `fired_evaluation` *process statement body and on the H-VHDL semantics rules permits us to deduce Equation (4.2).*

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

LISTING 4.1: The `fired_evaluation` process in the transition design architecture; its body statement assigns the `fired` output port; symbol ⇐ is the signal assignment operator.

Listing 4.2 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 4.2: The `priority_authorizations_evaluation` process in the transition design's architecture. The local variable `v_priority_combination` accumulates the product of the `priority_authorizations` input ports in the `for` loop; then the last statement assigns the value of `v_priority_combination` to the `s_priority_combination` internal signal.

Equation (4.3) 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.*

$$\sigma(id_t)(\text{"spc"}) = \prod_{i=0}^{\Delta(id_t)(\text{"input\_arcs\_number"})-1} \sigma(id_t)(\text{"priority\_authorizations"})[i] \tag{4.3}$$

In Equation (4.3), "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 C 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 Appendix).

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 $\sigma'(id_t)(''fired'') = \sigma'(id_t)(''s\_firable'') \,.\, \sigma'(id_t)(''s\_priority\_combination'')$. Using these definitions to rewrite the terms of the current goal, the new goal to prove is:

$$t \in Firable(s') \wedge t \in Sens(s'.M - \sum_{t_i \in Pr(t,s')} pre(t_i)) \Leftrightarrow$$
$$\sigma'(id_t)(''s\_firable'') \,.\, \sigma'(id_t)(''s\_priority\_combination'') = \texttt{true}$$

Thanks to Lemma 39, we know that $t \in Firable(s')$ iff $\sigma'(id_t)(''s\_firable'') = \texttt{true}$. Then, we can get rid of these two terms in the current goal; what is left to prove is:

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

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

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

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

To prove the current goal, we can equivalently show that:
$\forall i \in [0, \Delta(id_t)(\text{"}ian\text{"}) - 1],\ \sigma'(id_t)(\text{"}pauths\text{"})[i] = \texttt{true}.$
For a given $i \in [0, \Delta(id_t)(\text{"}ian\text{"}) - 1]$, let us show that $\sigma'(id_t)(\text{"}pauths\text{"})[i] = \texttt{true}.$ As shown in Figure 4.5, the signal `priority_authorizations` is an input port of the transition design. Therefore, to know what is the value of the element i-th element of port `priority_authorizations` 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, 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 of the `priority_authorizations` input port of the transition component instance id_t is connected to the j-th element of the `priority_authorizations` output port of the PCI id_p. Figure 4.6 shows the connection of the `priority_authorizations` port between the component instances id_p and id_t.

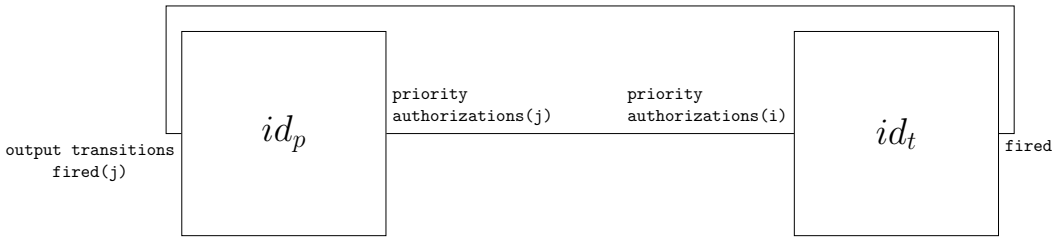


FIGURE 4.6: Connection of the j-th element of the `priority_authorizations` output port of id_p to the i-th element of the `priority_authorizations` input port of id_t; also the `fired` output port of id_t is connected to the j-th element of the `output_transitions_fired` input port of id_p.

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. Thus, to show that $\sigma'(id_t)(\text{"}pauths\text{"})[i] = \texttt{true}$, we must now show that $\sigma'(id_p)(\text{"}pauths\text{"})[j] = \texttt{true}.$ 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 4.7 illustrates the wiring of the `priority_authorizations` output port in a place design.

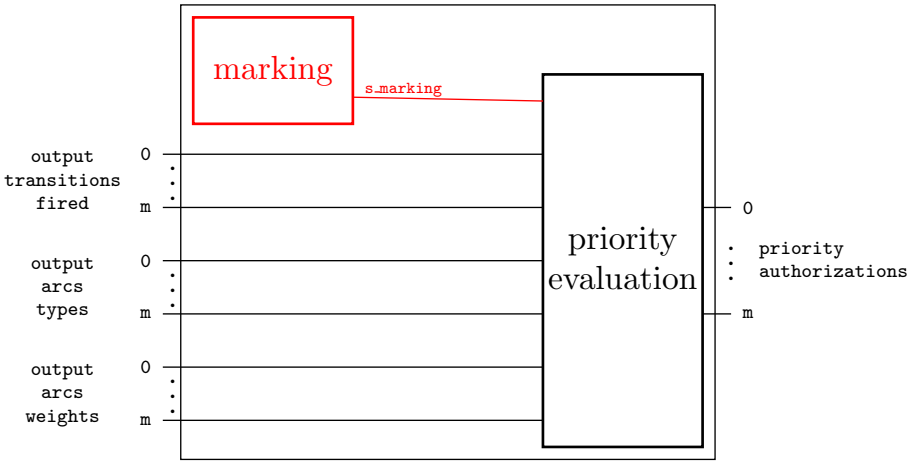


FIGURE 4.7: Wiring of the `priority_authorizations` output port in the architecture of the place design; the input port interface is on the left side and the output port interface is on the right side; the synchronous parts are in red and the combinational ones are in black.

Figure 4.7 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 input ports `output_transitions_fired`, `output_arcs_types` and `output_arcs_weights`. In Figure 4.7, 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 place component instance id_p, and is equal to the number of output transitions of place p. Listing 4.3 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 4.3: The `priority_evaluation` process in the place design's architecture.

In the statement body of the `priority_evaluation` process, each element 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 the j-th element of the `priority_authorizations` port at state σ':

$$\sigma'(id_p)(\text{"pauths"})[j] = (\sigma'(id_p)(\text{"s\_marking"}) - \texttt{vsots} \geq \sigma'(id)(\text{"output\_arcs\_weights"})[j]) \tag{4.4}$$

Then, rewriting the goal with Equation (4.4), the new goal is:

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

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:

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

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 4.3 (Lines 10 to 12), we can deduce the value of the `vsots`:

$$\texttt{vsots} = \sum_{l=0}^{j-1} \begin{cases} \sigma'(id_p)(\text{"oaw"})[l] \texttt{ if } \sigma'(id_p)(\text{"otf"})[l]. \\ \qquad\qquad\qquad\qquad \sigma'(id_p)(\text{"oat"})[l] = \texttt{basic} \\ 0 \textit{ otherwise} \end{cases} \tag{4.5}$$

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 $j-1$. The 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 by a `basic` input arc (i.e. the `output_arcs_types` input port of id_p equals `basic` at index l). 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 4.8 reuses the SITPN of Figure 4.4 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 transformation).

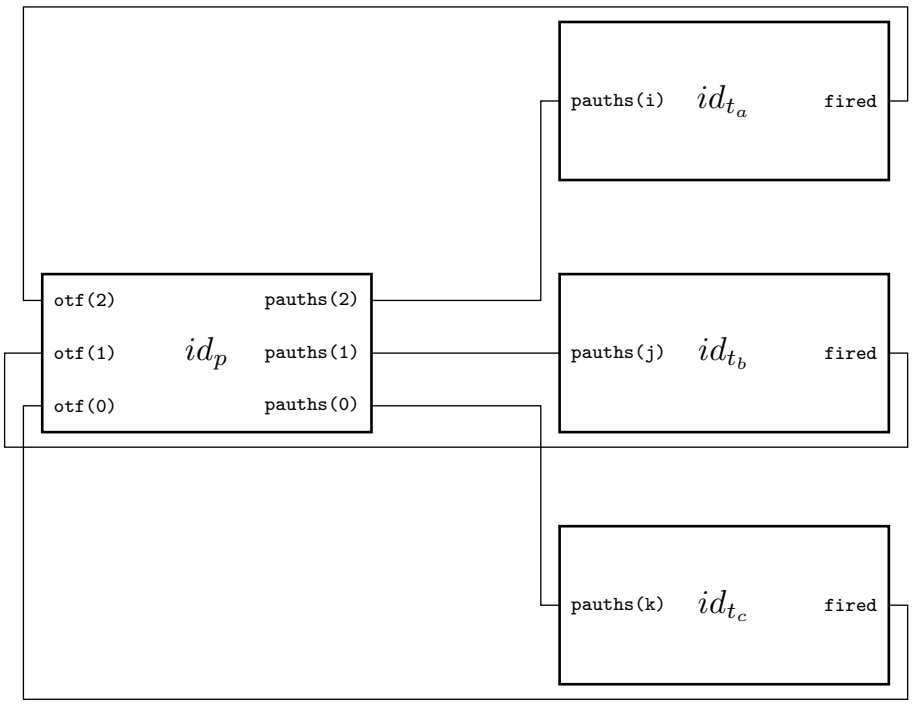


FIGURE 4.8: Connection between the `priority_authorizations` output port of PCI id_p and the `priority_authorizations` input port of TCIs id_{t_a}, id_{t_b} and id_{t_c}, and between the `output_transitions_fired` input port of id_p and the `fired` ports of id_{t_a}, id_{t_b} and id_{t_c}. `pauths` stands for `priority_authorizations` and `otf` stands for `output_transitions_fired`.

In Figure 4.8, 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.

As a reminder, the current goal to prove is:

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

The current goal is the $\mathcal{H}$-VHDL implementation of the test that the residual marking in place p enables transition t. We made the hypothesis 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 *Sens*, 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 preceding formula to the terms of the goal. We can easily show, appealing to Lemma 32, that $s'.M(p)$ equals $\sigma'(id_p)(\text{"}s\_marking\text{"})$. 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. ω. The last thing to show is the equality between the two sum terms:

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

On the left side of the equality, we have unfolded term $\sum_{t_i \in Pr(t,s')} pre(t_i)$ to its full definition (see Remark in Section). On the right side is the full definition of term `vsots`. We can reason by induction over the sum terms of the goal to complete the proof. At some point, we will have to show that there is a relation between an index $l \in [0, j-1]$ and a transition $t_i \in Pr(t,s')$. Thanks to the ordering of the indexes based on the priority order of output transitions (see Figure 4.8), we know that there is a bijection between the output transitions of p with a higher priority than t and the indexes of interval $[0, j-1]$. However, to complete the proof, we must assume that for a given transition t_i with a higher priority than t and its corresponding TCI id_{t_i} the "fired" equivalence holds, i.e.: $t_i \in Fired(s') \Leftrightarrow \sigma'(id_{t_i})("fired") = \texttt{true}$. Unfortunately, this is exactly the property 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. 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. 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 ← ∅
2 T_s ← T
3 while T_s ≠ ∅ do
4     tp ← GetTopPriorityTransitions(T_s, ≻)
5     f ← ElectFired(s, tp, F)
6     F ← F ∪ f
7     T_s ← T_s \ tp
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 for firing or have been discarded. At Line 4, function `GetTopPriorityTransitions` returns the tp set of top-priority transitions inside T_s. Then, we can proceed to the election of the fired transitions inside tp. We know that the following loop invariant holds: all fired transitions with a higher firing priority than the transitions of the tp set are inside set F. Therefore, set F contains all the transitions necessary to compute the residual marking that will be used to elect the fired transitions inside tp. This is done by

the `ElectFired` function at Line 5. Then, the set f of fired transitions inside tp is merged with the overall set of fired transitions F. The statement of Line 7 withdraws the transitions of tp from 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 top-priority transitions in T_s. Thus, tp is never empty and T_s is always decrementing after the assignment of Line 7. Thus, the algorithm always terminates and returns the set of fired transitions at state s.

We make a relational definition of Algorithm 2 through the definition of the *IsFiredSet* relation given in Definition 28. Definition 29 states that a given transition is fired in relation to the *IsFiredSet* relation.

Definition 28 (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:*
$IsFiredSet(s, fset) \equiv IsFiredSetAux(s, \emptyset, T, fset)$

Definition 29 (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 $t \in fset$.*

We are now satisfied with the definition of the set of fired transitions provided through the *IsFiredSet* 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 43. The full formal proof of Lemma 43 is given in Section C.4 of Appendix C.

The definition of the *IsFiredSet* relation depends on the definition of the *IsFiredSetAux* relation given in Definition 30. 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 44.

Definition 30 (IsFiredSetAux). *The IsFiredSetAux relation is defined by the following rules:*

$$\frac{}{IsFiredSetAux(s, fired, \emptyset, fired)}\ \textsc{IsFiredSetAuxNil}$$

$$\frac{\begin{array}{c} IsTopPrioritySet(T_s, tp) \\ ElectFired(s, fired, tp, fired') \\ IsFiredSetAux(s, fired', T_s \setminus tp, fset) \end{array}}{IsFiredSetAux(s, fired, T_s, fset)}\ \textsc{IsFiredSetAuxCons}$$

The *IsFiredSetAux* relation depends on the definitions of the *IsTopPrioritySet* and the *ElectFired* relations given in Definition 31 and 31. The *IsTopPrioritySet* is a relational implementation of the `GetTopPriorityTransitions` function appearing at Line 4 of Algorithm 2. The *ElectFired* relation is a relational implementation of the `ElectFired` function appearing at Line 5 of Algorithm 2.

Definition 31 (IsTopPrioritySet). *Given an $sitpn \in SITPN$, a subset $T_s \subseteq T$, and a subset $tp \subseteq T$, the IsTopPrioritySet relation is defined as follows:*
$IsTopPrioritySet(T_s, tp) \equiv IsTopPrioritySetAux(T_s, \emptyset, \emptyset, tp)$

Definition 32 (IsTopPrioritySetAux). *The IsTopPrioritySetAux relation is defined by the following rules:*

IsTPSetAuxEmpty
$$\frac{}{IsTopPrioritySetAux(\emptyset, T_b, tp, tp)}$$

IsTPSetAuxTP
$$\frac{\forall t' \in T_a \cup T_b, t' \not\succ t \quad IsTopPrioritySetAux(T_a, \{t\} \cup T_b, \{t\} \cup tp, tp')}{IsTopPrioritySetAux(\{t\} \cup T_a, T_b, tp, tp')}$$

IsTPSetAuxNTP
$$\frac{\exists t' \in T_a \cup T_b \; s.t. \; t' \succ t \quad IsTopPrioritySetAux(T_a, \{t\} \cup T_b, tp, tp')}{IsTopPrioritySetAux(\{t\} \cup T_a, T_b, tp, tp')}$$

Definition 33 (ElectFired). *The ElectFired relation is defined by the following rules:*

ElectFiredEmpty
$$\frac{}{ElectFired(s, fired, \emptyset, fired)}$$

ElectFired$\bot$
$$\frac{\neg\big(t \in Firable(s) \wedge t \in Sens(s.M - \sum_{t_i \in Pr(t,fired)} pre(t_i))\big) \quad ElectFired(s, fired, tp, fired')}{ElectFired(s, fired, \{t\} \cup tp, fired')} \; Pr(t, fired) = \{t' \mid t' \succ t \wedge t' \in fired\}$$

ElectFired$\top$
$$\frac{t \in Firable(s) \quad t \in Sens(s.M - \sum_{t_i \in Pr(t,fired)} pre(t_i)) \quad ElectFired(s, \{t\} \cup fired, tp, fired')}{ElectFired(s, fired, \{t\} \cup tp, fired')} \; Pr(t, fired) = \{t' \mid t' \succ t \wedge t' \in fired\}$$

4.4.2 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 4.9 shows a case of divergence between time counters and `s_time_counter` signals values in the course of an execution.

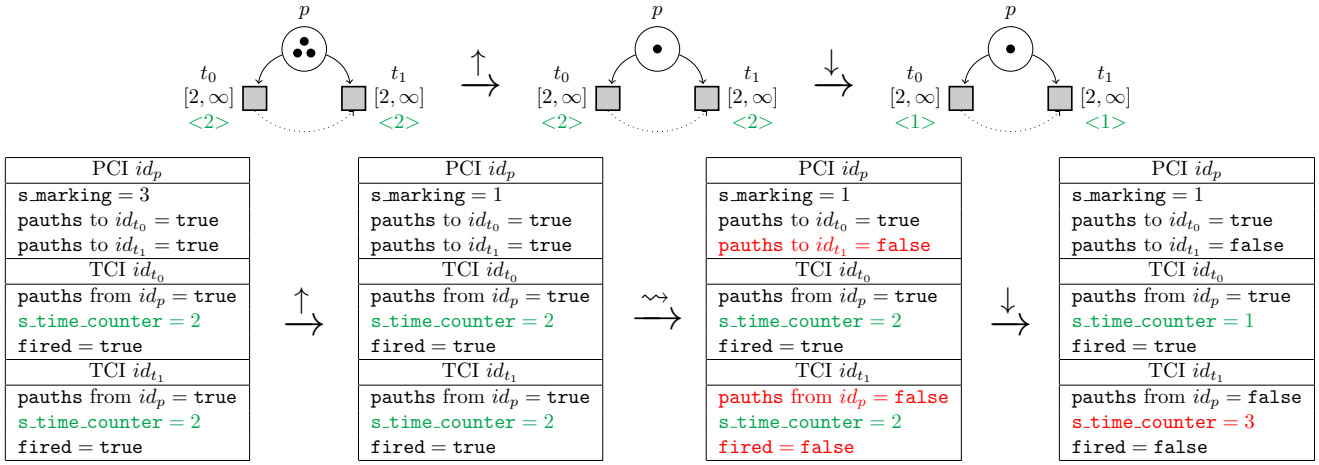


FIGURE 4.9: Bug detection: divergence between the value of time counters and the value of the `s_time_counter` signals due to the loss of the firing status information during the stabilization phase; the value of time counters and of the `s_time_counter` signals are in green; the value of diverging signals are in red.

In Figure 4.9, 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_countersignal` 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 any the SITPN semantics rules happening during 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 5). 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 26 and 35 about the equivalence of time counters).

4.5 Mechanized verification of the proof

The work of mechanizing the proof of the Full bisimulation theorem is an ongoing task. At the time of the writing, we have only verified thirty per cent of the proof concerning the Similar initial states 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 C) 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 .

add r
repo

Listing 4.4 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 mm 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 mm = (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 4.4: Coq implementation of the Full bisimulation theorem and the mechanized version of its proof.

The proof laid out in Listing 4.4 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 (i.e.). 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 4.4 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 [16]. 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 4.5 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 4.5: 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'[2]. 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 [36] to automate proof for such properties. The result is the tactic `minv` massively used in the proofs pertaining to state invariants[3].

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 C 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 Lemma 6 by taking a place p and a PCI identifier id_p linked through the γ binder returned by the transformation function. Then, we state the existence of a PCI statement, identified by id_p and with an associated generic map, input port map and output port map, in the behavior of the top-level design returned by the transformation function. To do so, we use the following the sentence:

"Let us take a $p \in P$ and an $id_p \in Comps(\Delta)$ such that $\gamma(p) = id_p$. By construction, there exist gm_p, ipm_p, opm_p s.t. $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$."

The expression "by construction" is a shorthand for "knowing how the target $\mathcal{H}$-VHDL design is constructed by the transformation function", or "based on the definition 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

[2]Remember that the γ binder is part of the compile-time state record type.

[3]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.

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 id 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.

4.6 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 4.2.

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 4.3 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 succesful 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 C. 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 C is long; about a hundred pages. However, as we explained in Section 4.4, 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 4.5, 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 [53], 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.

Appendix A

The place design in concrete and abstract VHDL syntax

```
entity place is
  generic(
    input_arcs_number : natural := 1;
    output_arcs_number : natural := 1;
    maximal_marking : natural := 1
  );
  port(
    clock : in std_logic;
    reset_n : in std_logic;
    initial_marking : in natural range 0 to maximal_marking;
    input_arcs_weights : in weight_vector_t(input_arcs_number1 downto 0);
    output_arcs_types : in arc_vector_t(output_arcs_number1 downto 0);
    output_arcs_weights : in weight_vector_t(output_arcs_number1 downto 0);
    input_transitions_fired : in std_logic_vector(input_arcs_number1 downto 0);
    output_transitions_fired : in std_logic_vector(output_arcs_number1 downto 0);
    output_arcs_valid : out std_logic_vector(output_arcs_number1 downto 0);
    priority_authorizations : out std_logic_vector(output_arcs_number1 downto 0);
    reinit_transitions_time : out std_logic_vector(output_arcs_number1 downto 0);
    marked : out std_logic
  );
end place;

architecture place_architecture of place is

  subtype local_weight_t is natural range 0 to maximal_marking;

  signal s_input_token_sum : local_weight_t;
  signal s_marking : local_weight_t;
  signal s_output_token_sum : local_weight_t;

begin

```

```
33  input_tokens_sum : process(input_arcs_weights, input_transitions_fired)
34    variable v_internal_input_token_sum : local_weight_t;
35  begin
36    v_internal_input_token_sum := 0;
37
38    for i in 0 to input_arcs_number - 1 loop
39      if (input_transitions_fired(i) = '1') then
40        v_internal_input_token_sum := v_internal_input_token_sum + input_arcs_weights(i
              );
41      end if;
42    end loop;
43
44    s_input_token_sum <= v_internal_input_token_sum;
45  end process input_tokens_sum;
46
47  output_tokens_sum : process(output_arcs_types, output_arcs_weights,
        output_transitions_fired)
48    variable v_internal_output_token_sum : local_weight_t;
49  begin
50    v_internal_output_token_sum := 0;
51
52    for i in 0 to output_arcs_number - 1 loop
53      if (output_transitions_fired(i) = '1' and output_arcs_types(i) = arc_t(BASIC)) then
54        v_internal_output_token_sum := v_internal_output_token_sum +
              output_arcs_weights(i);
55      end if;
56    end loop;
57
58    s_output_token_sum <= v_internal_output_token_sum;
59  end process output_tokens_sum;
60
61  marking : process(clock, reset_n, initial_marking)
62  begin
63    if (reset_n = '0') then
64      s_marking <= initial_marking;
65    elsif rising_edge(clock) then
66      s_marking <= s_marking + (s_input_token_sum - s_output_token_sum);
67    end if;
68  end process marking;
69
70  determine_marked : process(s_marking)
71  begin
72    if (s_marking = 0) then
73      marked <= '0';
74    else
75      marked <= '1';
76    end if;
```

```
77  end process determine_marked;
78
79  marking_validation_evaluation : process(output_arcs_types, output_arcs_weights,
        s_marking)
80  begin
81    for i in 0 to output_arcs_number − 1 loop
82      if ( ( ((output_arcs_types(i) = arc_t(BASIC)) or (output_arcs_types(i) = arc_t(TEST)))
            and (s_marking >= output_arcs_weights(i)) )
83           or ( (output_arcs_types(i) = arc_t(INHIBITOR)) and (s_marking <
                output_arcs_weights(i)) ) )
84      then
85        output_arcs_valid(i) ⇐ '1';
86      else
87        output_arcs_valid(i) ⇐ '0';
88      end if;
89    end loop;
90  end process marking_validation_evaluation;
91
92  priority_evaluation : process(output_arcs_types, output_arcs_weights,
        output_transitions_fired, s_marking)
93    variable v_saved_output_token_sum : local_weight_t;
94  begin
95    v_saved_output_token_sum := 0;
96
97    for i in 0 to output_arcs_number − 1 loop
98      if (s_marking >= v_saved_output_token_sum + output_arcs_weights(i)) then
99        priority_authorizations(i) ⇐ '1';
100     else
101       priority_authorizations(i) ⇐ '0';
102     end if;
103
104     if ((output_transitions_fired(i) = '1') and (output_arcs_types(i) = arc_t(BASIC)))
            then
105       v_saved_output_token_sum := v_saved_output_token_sum + output_arcs_weights(i);
106     end if;
107
108   end loop;
109 end process priority_evaluation;
110
111 reinit_transitions_time_evaluation : process(clock, reset_n)
112 begin
113   if (reset_n = '0') then
114     reinit_transitions_time ⇐ (others ⇒ '0');
115   elsif rising_edge(clock) then
116     for i in 0 to output_arcs_number − 1 loop
117       if ( (((output_arcs_types(i) = arc_t(BASIC)) or (output_arcs_types(i) = arc_t(TEST)))
                and (s_marking − s_output_token_sum < output_arcs_weights(i))
```

```
119                 and (s_output_token_sum > 0))
120               or output_transitions_fired(i) = '1' )
121         then
122           reinit_transitions_time(i) <= '1';
123         else
124           reinit_transitions_time(i) <= '0';
125         end if;
126       end loop;
127     end if;
128   end process reinit_transitions_time_evaluation;
129
130 end place_architecture;
```

LISTING A.1: The `place` design in concrete VHDL syntax.

```
1  design place place_architecture
2
3  -- Generic clause
4  ((input_arcs_number, natural(0,NATMAX), 1),
5   (output_arcs_number, natural(0,NATMAX), 1),
6   (maximal_marking, natural(0,NATMAX), 1))
7
8  -- Port clause
9  ((in, initial_marking, natural(0, maximal_marking)),
10  (in, input_arcs_weights, array(natural(0,255), 0, input_arcs_number−1)),
11  (in, output_arcs_types, array(natural(0,2), 0, output_arcs_number−1)),
12  (in, output_arcs_weights, array(natural(0,2), 0, output_arcs_number−1)),
13  (in, input_transitions_fired, array(boolean, 0, input_arcs_number−1)),
14  (in, output_transitions_fired, array(boolean, 0, output_arcs_number−1)),
15  (out, output_arcs_valid, array(boolean, 0, output_arcs_number−1)),
16  (out, priority_authorizations, array(boolean, 0, output_arcs_number−1)),
17  (out, reinit_transitions_time, array(boolean, 0, output_arcs_number−1)),
18  (out, marked, boolean))
19
20 -- Architecture declarative part
21 ((s_input_token_sum, natural(0, maximal_marking)),
22  (s_marking, natural(0, maximal_marking)),
23  (s_output_token_sum, natural(0, maximal_marking)))
24
25 -- Behavior
26 process(input_tokens_sum,
27         (input_arcs_weights, input_transitions_fired),
28         ((v_internal_input_token_sum, natural(0, maximal_marking))),
29
30     (v_internal_input_token_sum := 0;
31      (for(i, 0, input_arcs_number − 1)
32        (if (input_transitions_fired(i) = '1')
```

```
33            (v_internal_input_token_sum := v_internal_input_token_sum +
                  input_arcs_weights(i))));
34      s_input_token_sum ⇐ v_internal_input_token_sum)
35
36  process(output_tokens_sum,
37          (output_arcs_types, output_arcs_weights, output_transitions_fired),
38          ((v_internal_output_token_sum,natural(0, maximal_marking))),
39      (v_internal_output_token_sum := 0;
40      (for i in 0 to output_arcs_number − 1 loop
41        if (output_transitions_fired(i) = true and output_arcs_types(i) = 0)
42          (v_internal_output_token_sum := v_internal_output_token_sum +
                output_arcs_weights(i)));
43      s_output_token_sum ⇐ v_internal_output_token_sum)
44
45  process(marking, (clk, initial_marking), ∅)
46      rst (s_marking ⇐ initial_marking)
47          (rising (s_marking ⇐ s_marking + (s_input_token_sum − s_output_token_sum)))
48
49  process(determine_marked, (s_marking), ∅,
50          (marked ⇐ s_marking > 0))
51
52  process(marking_validation_evaluation,
53          (output_arcs_types, output_arcs_weights, s_marking), ∅,
54      (for(i, 0, output_arcs_number − 1)
55           (output_arcs_valid(i) ⇐ ((((output_arcs_types(i) = 0) or (output_arcs_types(i) = 1))
56                                        and (s_marking >= output_arcs_weights(i)))
57                                    or ((output_arcs_types(i) = 2)
58                                        and (s_marking < output_arcs_weights(i)))))))
59
60  process(priority_evaluation,
61          (output_arcs_types, output_arcs_weights, output_transitions_fired, s_marking),
62          ((v_saved_output_token_sum, natural(0, maximal_marking))),
63      (v_saved_output_token_sum := 0;
64       (for(i, 0, output_arcs_number − 1)
65          (priority_authorizations(i) ⇐ (s_marking >= v_saved_output_token_sum +
                output_arcs_weights(i))));
66        (if ((output_transitions_fired(i) = '1') and (output_arcs_types(i) = 0))
67          (v_saved_output_token_sum := v_saved_output_token_sum + output_arcs_weights(i))))
68
69  procees(reinit_transitions_time_evaluation, (clk), ∅,
70      rst (for (i, 0, output_arcs_number−1) (reinit_transitions_time(i) ⇐ false))
71          (rising
72           (for (i, 0, output_arcs_number−1)
73                (reinit_transitions_time(i) ⇐ (((output_arcs_types(i) = 0) or (
                      output_arcs_types(i) = 1))
74                                                and (s_marking − s_output_token_sum <
                                                      output_arcs_weights(i))
```

```
75                                    and (s_output_token_sum > 0))
76                                  or output_transitions_fired(i) = true))))
```

LISTING A.2: The `place` design in $\mathcal{H}$-VHDL abstract syntax.

Appendix B

The transition design in concrete and abstract VHDL syntax

```
1  entity transition is
2    generic(
3      transition_type : transition_t := NOT_TEMPORAL;
4      input_arcs_number : natural := 1;
5      conditions_number : natural := 1;
6      maximal_time_counter : natural := 1
7      );
8    port(
9      clock : in std_logic;
10     reset_n : in std_logic;
11     input_conditions : in std_logic_vector(conditions_number1 downto 0);
12     time_A_value : in natural range 0 to maximal_time_counter;
13     time_B_value : in natural range 0 to maximal_time_counter;
14     input_arcs_valid : in std_logic_vector(input_arcs_number1 downto 0);
15     reinit_time : in std_logic_vector(input_arcs_number1 downto 0);
16     priority_authorizations : in std_logic_vector(input_arcs_number1 downto 0);
17     fired : out std_logic
18     );
19 end transition;
20
21 architecture transition_architecture of transition is
22
23   signal s_condition_combination : std_logic;
24   signal s_enabled : std_logic;
25   signal s_firable : std_logic;
26   signal s_firing_condition : std_logic;
27   signal s_priority_combination : std_logic;
28   signal s_reinit_time_counter : std_logic;
29   signal s_time_counter : natural range 0 to maximal_time_counter;
30
31 begin
32
```

```
33  condition_evaluation : process(input_conditions)
34    variable v_internal_condition : std_logic;
35  begin
36    v_internal_condition := '1';
37
38    for i in 0 to conditions_number - 1 loop
39      v_internal_condition := v_internal_condition and input_conditions(i);
40    end loop;
41
42    s_condition_combination <= v_internal_condition;
43  end process condition_evaluation;
44
45  enable_evaluation : process(input_arcs_valid)
46    variable v_internal_enabled : std_logic;
47  begin
48    v_internal_enabled := '1';
49
50    for i in 0 to input_arcs_number - 1 loop
51      v_internal_enabled := v_internal_enabled and input_arcs_valid(i);
52    end loop;
53
54    s_enabled <= v_internal_enabled;
55  end process enable_evaluation;
56
57  reinit_time_counter_evaluation : process(reinit_time, s_enabled)
58    variable v_internal_reinit_time_counter : std_logic;
59  begin
60    v_internal_reinit_time_counter := '0';
61
62    for i in 0 to input_arcs_number - 1 loop
63      v_internal_reinit_time_counter := v_internal_reinit_time_counter or reinit_time(i
          );
64    end loop;
65
66    s_reinit_time_counter <= v_internal_reinit_time_counter;
67  end process reinit_time_counter_evaluation;
68
69  time_counter : process(reset_n, clock)
70  begin
71    if (reset_n = '0') then
72      s_time_counter <= 0;
73    elsif falling_edge(clock) then
74      if ((s_enabled = '1') and (transition_type /= transition_t(NOT_TEMPORAL))) then
75        if (s_reinit_time_counter = '0') then
76          if (s_time_counter < maximal_time_counter) then
77            s_time_counter <= s_time_counter + 1;
78          end if;
```

```
79          else
80            s_time_counter <= 1;
81          end if;
82        else
83          s_time_counter <= 0;
84        end if;
85      end if;
86    end process time_counter;
87
88    firing_condition_evaluation : process (s_enabled, s_condition_combination,
          s_reinit_time_counter, s_time_counter)
89    begin
90      if ( (s_condition_combination = '1')
91           and (s_enabled = '1')
92           and ( (transition_type = transition_t(NOT_TEMPORAL))
93
94                 or ( (transition_type = transition_t(TEMPORAL_A_B))
95                      and (s_reinit_time_counter = '0')
96                      and (s_time_counter >= (time_A_value1))
97                      and (s_time_counter < time_B_value)
98                      and (time_A_value /= 0)
99                      and (time_B_value /= 0) )
100
101                or ( (s_reinit_time_counter = '0')
102                     and (time_A_value /= 0)
103                     and ( ((transition_type = transition_t(TEMPORAL_A_A))
104                            and (s_time_counter = (time_A_value1)))
105                     or ( (transition_type = transition_t(TEMPORAL_A_INFINITE))
106                          and (s_time_counter >= (time_A_value1)) )
107                )
108
109                or ( (transition_type /= transition_t(NOT_TEMPORAL))
110                     and (s_reinit_time_counter = '1')
111                     and (time_A_value = 1) )
112                )
113     ) then
114       s_firing_condition <= '1';
115     else
116       s_firing_condition <= '0';
117     end if;
118   end process firing_condition_evaluation;
119
120   priority_authorization_evaluation : process(priority_authorizations)
121     variable v_priority_combination : std_logic;
122   begin
123     v_priority_combination := '1';
124
```

```
125     for i in 0 to input_arcs_number − 1 loop
126       v_priority_combination := v_priority_combination and priority_authorizations(i);
127     end loop;
128
129     s_priority_combination ⇐ v_priority_combination;
130   end process priority_authorization_evaluation;
131
132   firable : process(reset_n, clock)
133   begin
134     if (reset_n = '0') then
135       s_firable ⇐ '0';
136     elsif falling_edge(clock) then
137       s_firable ⇐ s_firing_condition;
138     end if;
139   end process firable;
140
141   fired_evaluation : process (s_firable, s_priority_combination)
142   begin
143     fired ⇐ s_firable and s_priority_combination;
144   end process fired_evaluation;
145
146 end transition_architecture;
```

LISTING B.1: The transition design in concrete VHDL syntax.

Appendix C

Semantic preservation proof

Constants and signals reference			
Full name	*Alias*	*Category*	*Type*
"input_conditions"	*"ic"*	input port (T)	$\mathbb{B}$
"reinit_time"	*"rt"*	input port (T)	$\mathbb{B}$
"input_arcs_valid"	*"iav"*	input port (T)	$\mathbb{B}$
"fired"	*"f"*	output port (T)	$\mathbb{B}$
"s_condition_combination"	*"scc"*	internal signal (T)	$\mathbb{B}$
"s_reinit_time_counter"	*"srtc"*	internal signal (T)	$\mathbb{B}$
"s_priority_combination"	*"spc"*	internal signal (T)	$\mathbb{B}$
"s_fired"	*"sf"*	internal signal (T)	$\mathbb{B}$
"s_firable"	*"sfa"*	internal signal (T)	$\mathbb{B}$
"s_enabled"	*"se"*	internal signal (T)	$\mathbb{B}$
"input_arcs_number"	*"ian"*	generic constant (T)	$\mathbb{N}$
"transition_type"	*"tt"*	generic constant (T)	{`not_temp`, `temp_a_b`, `temp_a_a`, `temp_a_inf`}
"conditions_number"	*"cn"*	generic constant (T)	$\mathbb{N}$
"maximal_time_counter"	*"mtc"*	generic constant (T)	$\mathbb{N}$
"s_marking"	*"sm"*	internal signal (P)	$\mathbb{N}$
"s_output_token_sum"	*"sots"*	internal signal (P)	$\mathbb{N}$
"s_input_token_sum"	*"sits"*	internal signal (P)	$\mathbb{N}$
"reinit_transition_time"	*"rtt"*	output port (P)	$\mathbb{B}$
"output_arcs_types"	*"oat"*	input port (P)	{`basic`, `test`, `inhib`}
"output_arcs_weights"	*"oaw"*	input port (P)	$\mathbb{N}$
"output_transition_fired"	*"otf"*	input port (P)	$\mathbb{B}$
"input_arcs_weights"	*"iaw"*	input port (P)	$\mathbb{N}$
"input_transition_fired"	*"itf"*	input port (P)	$\mathbb{B}$

TABLE C.1: Constants and signals reference for the $\mathcal{H}$-VHDL transition and place designs

C.1 Initial States

Definition 34 (Initial state hypotheses). *Given an* $sitpn \in SITPN$, $d \in design$, $\gamma \in WM(sitpn, d)$, $\Delta \in ElDesign(d, \mathcal{D}_{\mathcal{H}})$, $\sigma_e, \sigma_0 \in \Sigma(\Delta)$, *assume that:*

- *SITPN* $sitpn$ *translates into design* d: $\lfloor sitpn \rfloor_{\mathcal{H}} = (d, \gamma)$
- Δ *is the elaborated version of* d, σ_e *is the default state of* Δ, *i.e, state of* Δ *where all signals have their default value:*

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

- σ_0 *is the initial state of* Δ: $\Delta, \sigma_e \vdash d.cs \xrightarrow{init} \sigma_0$

Lemma 5 (Similar initial states). *For all* $sitpn \in SITPN$, $d \in design$, $\gamma \in WM(sitpn, d)$, $\Delta \in ElDesign(d, \mathcal{D}_{\mathcal{H}})$, $\sigma_e, \sigma_0 \in \Sigma(\Delta)$ *that verify the hypotheses of Definition 34, then* $\gamma \vdash s_0 \sim \sigma_0$.

Proof. By definition of the General state similarity relation, there are 6 points to prove.

1. $\forall p \in P, id_p \in Comps(\Delta)\ s.t.\ \gamma(p) = id_p,\ s_0.M(p) = \sigma_0(id_p)(\text{"s\_marking"})$.
2. $\forall t \in T_i, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t,$
   $\big(upper(I_s(t)) = \infty \wedge s_0.I(t) \leq lower(I_s(t)) \Rightarrow s_0.I(t) = \sigma_0(id_t)(\text{"s\_time\_counter"})\big)$
   $\wedge\big(upper(I_s(t)) = \infty \wedge s_0.I(t) > lower(I_s(t)) \Rightarrow \sigma_0(id_t)(\text{"s\_time\_counter"}) = lower(I_s(t))\big)$
   $\wedge\big(upper(I_s(t)) \neq \infty \wedge s_0.I(t) > upper(I_s(t)) \Rightarrow \sigma_0(id_t)(\text{"s\_time\_counter"}) = upper(I_s(t))\big)$
   $\wedge\big(upper(I_s(t)) \neq \infty \wedge s_0.I(t) \leq upper(I_s(t)) \Rightarrow s_0.I(t) = \sigma_0(id_t)(\text{"s\_time\_counter"})\big)$.
3. $\forall t \in T_i, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t, s_0.reset_t(t) = \sigma_0(id_t)(\text{"s\_reinit\_time\_counter"})$.
4. $\forall c \in \mathcal{C}, id_c \in Ins(\Delta)\ s.t.\ \gamma(c) = id_c,\ s_0.cond(c) = \sigma_0(id_c)$.
5. $\forall a \in \mathcal{A}, id_a \in Outs(\Delta)\ s.t.\ \gamma(a) = id_a,\ s_0.ex(a) = \sigma_0(id_a)$.
6. $\forall f \in \mathcal{F}, id_f \in Outs(\Delta)\ s.t.\ \gamma(f) = id_f,\ s_0.ex(f) = \sigma_0(id_f)$.

- Apply the Initial states equal marking lemma to solve 1.
- Apply the Initial states equal time counters lemma to solve 2.
- Apply the Initial states equal reset orders lemma to solve 3.
- Apply the Initial states equal condition values lemma to solve 4.
- Apply the Initial states equal action executions lemma to solve 5.
- Apply the Initial states equal function executions lemma to solve 6.

□

C.1.1 Initial states and marking

Lemma 6 (Initial states equal marking). *For all* $sitpn \in SITPN$, $d \in design$, $\gamma \in WM(sitpn, d)$, $\Delta \in ElDesign(d, \mathcal{D}_{\mathcal{H}}), \sigma_e, \sigma_0 \in \Sigma(\Delta)$ *that verify the hypotheses of Definition 34, then* $\forall p \in P, id_p \in Comps(\Delta)$ *s.t.* $\gamma(p) = id_p$, $s_0.M(p) = \sigma_0(id_p)(\text{"}s\_marking\text{"})$.

Proof. Given a $p \in P$ and an $id_p \in Comps(\Delta)$ s.t. $\gamma(p) = id_p$, let us show that
$\boxed{s_0.M(p) = \sigma_0(id_p)(\text{"}s\_marking\text{"}).}$
By construction and by definition of id_p, there exist gm_p, ipm_p, opm_p s.t. $\texttt{comp}(id_p, \text{"}place\text{"}, gm_p, ipm_p, opm_p) \in d.cs$.
By property of the $\mathcal{H}$-VHDL initialization relation, $\texttt{comp}(id_p, \text{"}place\text{"}, gm_p, ipm_p, opm_p) \in d.cs$, and through the examination of the `marking` process defined in the place design architecture, we can deduce $\sigma_0(id_p)(\text{"}s\_marking\text{"}) = \sigma_0(id_p)(\text{"}initial\_marking\text{"})$.
Rewriting $\sigma_0(id_p)(\text{"}sm\text{"})$ as $\sigma_0(id_p)(\text{"}initial\_marking\text{"})$, $\boxed{\sigma_0(id_p)(\text{"}initial\_marking\text{"}) = s_0.M(p).}$
By construction, $<\texttt{initial\_marking} \Rightarrow M_0(p)> \in ipm_p$.
By property of the $\mathcal{H}$-VHDL initialization relation, and $\texttt{comp}(id_p, \text{"}place\text{"}, gm_p, ipm_p, opm_p) \in d.cs$, then $\sigma_0(id_p)(\text{"}initial\_marking\text{"}) = M_0(p)$. Rewriting $\sigma_0(id_p)(\text{"}initial\_marking\text{"})$ as $M_0(p)$ in the current goal: $\boxed{M_0(p) = s_0.M(p).}$
By definition of s_0, we can rewrite $s_0.M(p)$ as $M_0(p)$ in the current goal, tautology.
□

Lemma 7 (Null input token sum at initial state). *For all* $sitpn \in SITPN$, $d \in design$, $\gamma \in WM(sitpn, d)$, $\Delta \in ElDesign(d, \mathcal{D}_{\mathcal{H}}), \sigma_e, \sigma_0 \in \Sigma(\Delta)$ *that verify the hypotheses of Definition 34, then* $\forall p \in P, id_p \in Comps(\Delta)$ *s.t.* $\gamma(p) = id_p$, $\sigma_0(id_p)(\text{"}s\_input\_token\_sum\text{"}) = 0$.

Proof. Given a p and an id_p s.t. $\gamma(p) = id_p$, let us show that $\boxed{\sigma_0(id_p)(\text{"}s\_input\_token\_sum\text{"}) = 0.}$
By construction and by definition of id_p, there exist gm_p, ipm_p, opm_p s.t. $\texttt{comp}(id_p, \text{"}place\text{"}, gm_p, ipm_p, opm_p) \in d.cs$.
By property of the initialization relation, $\texttt{comp}(id_p, \text{"}place\text{"}, gm_p, ipm_p, opm_p) \in d.cs$, and through the examination of the `input_tokens_sum` process defined in the place design architecture, we can deduce:

$$\sigma_0(id_p)(\text{"}sits\text{"}) = \sum_{i=0}^{\Delta(id_p)(\text{"}ian\text{"})-1} \begin{cases} \sigma_0(id_p)(\text{"}iaw\text{"})[i] \texttt{ if } \sigma_0(id_p)(\text{"}itf\text{"})[i] \\ 0 \textit{ otherwise} \end{cases} \tag{C.1}$$

Rewriting the goal with Equation (C.1):
$$\boxed{\sum_{i=0}^{\Delta(id_p)(\text{"}ian\text{"})-1} \begin{cases} \sigma_0(id_p)(\text{"}iaw\text{"})[i] \texttt{ if } \sigma_0(id_p)(\text{"}itf\text{"})[i] \\ 0 \textit{ otherwise} \end{cases} = 0.}$$
Let us perform case analysis on $input(p)$; there are two cases:

1. $input(p) = \emptyset$:

   By construction, $<\texttt{input\_arcs\_number} \Rightarrow 1> \in gm_p$, $<\texttt{input\_transitions\_fired(0)} \Rightarrow \texttt{true}> \in ipm_p$, and $<\texttt{input\_arcs\_weights(0)} \Rightarrow 0> \in ipm_p$.

By property of the elaboration relation, $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$, and $<\texttt{input\_arcs\_number} \Rightarrow 1> \in gm_p$, we can deduce $\Delta(id_p)("ian") = 1$.

By property of the initialization relation, $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$, $<\texttt{input\_transitions\_fired(0)} \Rightarrow \texttt{true}> \in ipm_p$ and $<\texttt{input\_arcs\_weights(0)} \Rightarrow 0> \in ipm_p$, we can deduce $\sigma_0(id_p)("itf")[0] = \texttt{true}$ and $\sigma_0(id_p)("iaw")[0] = 0$.

Rewriting the goal with $\Delta(id_p)("ian") = 1$, $\sigma_0(id_p)("itf")[0] = \texttt{true}$, $\sigma_0(id_p)("iaw")[0] = 0$ and simplifying the goal, tautology.

2. $input(p) \neq \emptyset$:

By construction, $<\texttt{input\_arcs\_number} \Rightarrow |input(p)|> \in gm_p$, and by property of the elaboration relation, and $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$, we can deduce $\Delta(id_p)("ian") = |input(p)|$.

Let us reason by induction on the sum term of the goal.

- **BASE CASE**: The sum term equals 0, then tautology.
- **INDUCTION CASE**:

$$\sum_{i=1}^{\Delta(id_p)("ian")-1} \begin{cases} \sigma_0(id_p)("iaw")[i] \texttt{ if } \sigma_0(id_p)("itf")[i] \\ 0 \textit{ otherwise} \end{cases} = 0$$

$$\begin{cases} \sigma_0(id_p)("iaw")[0] \texttt{ if } \sigma_0(id_p)("itf")[0] \\ 0 \textit{ otherwise} \end{cases} + \sum_{i=1}^{\Delta(id_p)("ian")-1} \begin{cases} \sigma_0(id_p)("iaw")[i] \texttt{ if } \sigma_0(id_p)("itf")[i] \\ 0 \textit{ otherwise} \end{cases}$$

Using the induction hypothesis to rewrite the goal:

$$\begin{cases} \sigma_0(id_p)("iaw")[0] \texttt{ if } \sigma_0(id_p)("itf")[0] \\ 0 \textit{ otherwise} \end{cases} = 0$$

Since $input(p) \neq \emptyset$, by construction, there exist an $id_t \in Comps(\Delta), gm_t, ipm_t, opm_t$ s.t. $\texttt{comp}(id_t, "transition", gm_t, ipm_t, opm_t) \in d.cs$, $id_{ft} \in Sigs(\Delta)$ s.t. $<\texttt{fired} \Rightarrow id_{ft}> \in opm_t$ and $<\texttt{input\_transitions\_fired(0)} \Rightarrow \texttt{id}_{\texttt{ft}}> \in ipm_p$.

By property of the initialization relation, $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$, $\texttt{comp}(id_t, "transition", gm_t, ipm_t, opm_t) \in d.cs$, $<\texttt{fired} \Rightarrow id_{ft}> \in opm_t$ and $<\texttt{input\_transitions\_fired(0)} \Rightarrow \texttt{id}_{\texttt{ft}}> \in ipm_p$, we can deduce $\sigma_0(id_p)("itf")[0] = \sigma_0(id_t)("fired")$.

Rewriting the goal with $\sigma_0(id_p)("itf")[0] = \sigma_0(id_t)("fired")$:

$$\begin{cases} \sigma_0(id_p)("iaw")[0] \texttt{ if } \sigma_0(id_t)("fired") \\ 0 \textit{ otherwise} \end{cases} = 0$$

Appealing to Lemma 14, we can deduce $\sigma_0(id_t)("fired") = \texttt{false}$.

Rewriting the goal with $\sigma_0(id_t)("fired") = \texttt{false}$, and simplifying the goal, tautology.

□

Lemma 8 (Null output token sum at initial state). *For all* $sitpn \in SITPN$, $d \in design$, $\gamma \in WM(sitpn, d)$, $\Delta \in ElDesign(d, \mathcal{D}_{\mathcal{H}}), \sigma_e, \sigma_0 \in \Sigma(\Delta)$ *that verify the hypotheses of Definition 34, then* $\forall p \in P, id_p \in Comps(\Delta)$ *s.t.* $\gamma(p) = id_p, \sigma_0(id_p)(\text{"s\_output\_token\_sum"}) = 0$.

Proof. The proof is similar to the proof of Lemma 7. □

C.1.2 Initial states and time counters

Lemma 9 (Initial states equal time counters). *For all* $sitpn \in SITPN, d \in design, \gamma \in WM(sitpn, d)$, $\Delta \in ElDesign(d, \mathcal{D}_{\mathcal{H}}), \sigma_e, \sigma_0 \in \Sigma(\Delta)$ *that verify the hypotheses of Definition 34, then* $\forall t \in T_i, id_t \in Comps(\Delta)$ *s.t.* $\gamma(t) = id_t$,
$upper(I_s(t)) = \infty \wedge s_0.I(t) \leq lower(I_s(t)) \Rightarrow s_0.I(t) = \sigma_0(id_t)(\text{"s\_time\_counter"}) \wedge$
$upper(I_s(t)) = \infty \wedge s_0.I(t) > lower(I_s(t)) \Rightarrow \sigma_0(id_t)(\text{"s\_time\_counter"}) = lower(I_s(t)) \wedge$
$upper(I_s(t)) \neq \infty \wedge s_0.I(t) > upper(I_s(t)) \Rightarrow \sigma_0(id_t)(\text{"s\_time\_counter"}) = upper(I_s(t)) \wedge$
$upper(I_s(t)) \neq \infty \wedge s_0.I(t) \leq upper(I_s(t)) \Rightarrow s_0.I(t) = \sigma_0(id_t)(\text{"s\_time\_counter"})$.

Proof. Given a $t \in T_i$ and an $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, let us show that:

1. $\boxed{upper(I_s(t)) = \infty \wedge s_0.I(t) \leq lower(I_s(t)) \Rightarrow s_0.I(t) = \sigma_0(id_t)(\text{"s\_time\_counter"})}$
2. $\boxed{upper(I_s(t)) = \infty \wedge s_0.I(t) > lower(I_s(t)) \Rightarrow \sigma_0(id_t)(\text{"s\_time\_counter"}) = lower(I_s(t))}$
3. $\boxed{upper(I_s(t)) \neq \infty \wedge s_0.I(t) > upper(I_s(t)) \Rightarrow \sigma_0(id_t)(\text{"s\_time\_counter"}) = upper(I_s(t))}$
4. $\boxed{upper(I_s(t)) \neq \infty \wedge s_0.I(t) \leq upper(I_s(t)) \Rightarrow s_0.I(t) = \sigma_0(id_t)(\text{"s\_time\_counter"})}$

By construction and by definition of id_p, there exist gm_p, ipm_p, opm_p s.t. $\texttt{comp}(id_p, \text{"place"}, gm_p, ipm_p, opm_p) \in d.cs$.
Then, let us show the 4 previous points.

1. Assuming that $upper(I_s(t)) = \infty \wedge s_0.I(t) \leq lower(I_s(t))$, then let us show $\boxed{s_0.I(t) = \sigma_0(id_t)(\text{"s\_time\_counter"}).}$

   Rewriting $s_0.I(t)$ as 0, by definition of s_0, $\boxed{\sigma_0(id_t)(\text{"s\_time\_counter"}) = 0.}$

   By property of the $\mathcal{H}$-VHDL initialization relation, $\texttt{comp}(id_t, \text{"transition"}, gm_t, ipm_t, opm_t) \in d.cs$, and through the examination of the `time_counter` process defined in the transition design architecture, we can deduce $\sigma_0(id_t)(\text{"s\_time\_counter"}) = 0$.

2. Assuming that $upper(I_s(t)) = \infty$ and $s_0.I(t) > lower(I_s(t))$, let us show $\boxed{\sigma_0(id_t)(\text{"s\_time\_counter"}) = lower(I_s(t))}$.

   By definition, $lower(I_s(t)) \in \mathbb{N}^*$ and $s_0.I(t) = 0$. Then, $lower(I_s(t)) < 0$ is a contradiction.

3. Assuming that $upper(I_s(t)) \neq \infty$ and $s_0.I(t) > upper(I_s(t))$, let us show $\boxed{\sigma_0(id_t)(\text{"s\_time\_counter"}) = upper(I_s(t))}$.

   By definition, $upper(I_s(t)) \in \mathbb{N}^*$ and $s_0.I(t) = 0$. Then, $upper(I_s(t)) < 0$ is a contradiction.

4. Assuming that $upper(I_s(t)) \neq \infty$ and $s_0.I(t) \leq upper(I_s(t))$, let us show $\boxed{s_0.I(t) = \sigma_0(id_t)(\text{"s\_time\_counter"})}$.

   Rewriting $s_0.I(t)$ as 0, by definition of s_0, $\boxed{\sigma_0(id_t)(\text{"s\_time\_counter"}) = 0.}$

   By property of the $\mathcal{H}$-VHDL initialization relation, $\texttt{comp}(id_t, \text{"transition"}, gm_t, ipm_t, opm_t) \in d.cs$, and through the examination of the `time_counter` process defined in the transition design architecture, we can deduce $\sigma_0(id_t)(\text{"s\_time\_counter"}) = 0$.

□

C.1.3 Initial states and reset orders

Lemma 10 (Initial states equal reset orders). *For all $sitpn \in SITPN, d \in design, \gamma \in WM(sitpn, d)$, $\Delta \in ElDesign(d, \mathcal{D}_{\mathcal{H}}), \sigma_e, \sigma_0 \in \Sigma(\Delta)$ that verify the hypotheses of Definition 34, then $\forall t \in T_i, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, $s_0.reset_t(t) = \sigma_0(id_t)(\text{"s\_reinit\_time\_counter"})$.*

Proof. Given a $t \in T_i$ and an $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, let us show that $\boxed{s_0.reset_t(t) = \sigma_0(id_t)(\text{"s\_reinit\_time\_counter"}).}$
Rewriting $s_0.reset_t(t)$ as `false`, by definition of s_0, $\boxed{\sigma_0(id_t)(\text{"s\_reinit\_time\_counter"}) = \texttt{false}.}$
By construction and by definition of id_t, there exist gm_t, ipm_t, opm_t s.t. $\texttt{comp}(id_t, \text{"transition"}, gm_t, ipm_t, opm_t) \in d.cs$.
By property of the $\mathcal{H}$-VHDL initialization relation, $\texttt{comp}(id_t, \text{"transition"}, gm_t, ipm_t, opm_t) \in d.cs$, and through the examination of the `reinit_time_counter_evaluation` process defined in the transition design architecture
we can deduce $\sigma_0(id_t)(\text{"s\_reinit\_time\_counter"}) = \prod_{i=0}^{\Delta(id_t)(\text{"ian"})-1} \sigma_0(id_t)(\text{"rt"})[i]$.
Rewriting $\sigma_0(id_t)(\text{"s\_reinit\_time\_counter"})$ as $\prod_{i=0}^{\Delta(id_t)(\text{"ian"})-1} \sigma_0(id_t)(\text{"rt"})[i]$,

$\boxed{\prod_{i=0}^{\Delta(id_t)(\text{"ian"})-1} \sigma_0(id_t)(\text{"rt"})[i] = \texttt{false}.}$
For all $t \in T$ (resp. $p \in P$), let $input(t)$ (resp. $input(p)$) be the set of input places of t (resp. input transitions of p), and let $output(t)$ (resp. $output(p)$) be the set of output places of t (resp. output transitions of p).
Let us perform case analysis on $input(t)$; there are 2 cases:

- **CASE** $input(t) = \emptyset$.

By construction, $<\texttt{input\_arcs\_number} \Rightarrow 1> \in gm_t$, and by property of the elaboration relation, and $\texttt{comp}(id_t, "transition", gm_t, ipm_t, opm_t) \in d.cs$, we can deduce $\Delta(id_t)("ian") = 1$.

By construction, $< \texttt{reinit\_time(0)} \Rightarrow \texttt{false} > \in ipm_t$, and by property of the initialization relation and $\texttt{comp}(id_t, "transition", gm_t, ipm_t, opm_t) \in d.cs$, we can deduce $\sigma_0(id_t)("rt")[0] = \texttt{false}$.

Rewriting $\Delta(id_t)("ian")$ as 1 and $\sigma_0(id_t)("rt")[0]$ as false, tautology.

- **CASE** $input(t) \neq \emptyset$.

  To prove the current goal, we can equivalently prove that
  $\boxed{\exists i \in [0, \Delta(id_t)("ian") - 1] \; s.t. \; \sigma_0(id_t)("rt")[i] = \texttt{false}.}$

  Since $input(t) \neq \emptyset$, $\exists p \; s.t. \; p \in input(t)$. Let us take such a $p \in input(t)$.

  By construction, for all $p \in P$, there exist $id_p \; s.t. \; \gamma(p) = id_p$.

  By construction and by definition of id_p, there exist gm_p, ipm_p, opm_p s.t. $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$.

  By construction, there exist $i \in [0, |input(t)| - 1]$, $j \in [0, |output(p)| - 1]$, $id_{ji} \in Sigs(\Delta)$ s.t. $<\texttt{reinit\_transitions\_time(j)} \Rightarrow id_{ji}> \in opm_p$ and $<\texttt{reinit\_time(i)} \Rightarrow id_{ji}> \in ipm_t$. Let us take such a i, j and id_{ji}.

  By construction and $input(t) \neq \emptyset$, $<\texttt{input\_arcs\_number} \Rightarrow |input(t)|> \in gm_t$.

  By property of the $\mathcal{H}$-VHDL elaboration relation and $<\texttt{input\_arcs\_number} \Rightarrow |input(t)|> \in gm_t$, we can deduce $\Delta(id_t)("ian") = |input(t)|$.

  Since $\Delta(id_t)("ian") = |input(t)|$ and we have an $i \in [0, |input(t)| - 1]$, then, we have an $i \in [0, \Delta(id_t)("ian") - 1]$. Let us take that i to prove the goal.

  Then, we must show $\boxed{\sigma_0(id_t)("rt")[i] = \texttt{false}.}$

  By property of the $\mathcal{H}$-VHDL initialization relation and $<\texttt{reinit\_time(i)} \Rightarrow id_{ji}> \in ipm_t$, we can deduce $\sigma_0(id_t)("rt")[i] = \sigma_0("id_{ji}")$.

  Rewriting $\sigma_0(id_t)("rt")[i]$ as $\sigma_0("id_{ji}")$, $\boxed{\sigma_0("id_{ji}") = \texttt{false}.}$

  By property of the $\mathcal{H}$-VHDL initialization relation and $< \texttt{reinit\_transitions\_time(j)} \Rightarrow id_{ji} > \in opm_p$, we can deduce $\sigma_0("id_{ji}") = \sigma_0(id_p)("rtt")[j]$.

  Rewriting $\sigma_0("id_{ji}")$ as $\sigma_0(id_p)("rtt")[j]$, $\boxed{\sigma_p^0("rtt")[j] = \texttt{false}.}$

  Since $t \in output(p)$, then we know that $output(p) \neq \emptyset$.

  Then, by construction, $<\texttt{output\_arcs\_number} \Rightarrow |output(p)|> \in gm_p$.

  By property of the elaboration relation and $<\texttt{output\_arcs\_number} \Rightarrow |output(p)|> \in gm_p$, we can deduce that $\Delta(id_p)("oan") = |output(p)|$.

  Since $\Delta(id_p)("oan") = |output(p)|$ and $j \in [0, |output(p)| - 1]$, then $j \in [0, \Delta(id_p)("oan") - 1]$.

By property of the $\mathcal{H}$-VHDL initialization relation, $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$, through the examination of the `reinit_transitions_time_evaluation` process defined in the place design architecture, and since $j \in [0, \Delta(id_p)("oan") - 1]$, $\sigma_0(id_p)("rtt")[j] = \texttt{false}$.

□

C.1.4 Initial states and condition values

Lemma 11 (Initial states equal condition values). *For all* $sitpn \in SITPN$, $d \in design$, $\gamma \in WM(sitpn, d)$, $\Delta \in ElDesign(d, \mathcal{D}_\mathcal{H}), \sigma_e, \sigma_0 \in \Sigma(\Delta)$ *that verify the hypotheses of Definition 34, then* $\forall c \in \mathcal{C}, id_c \in Ins(\Delta)$ *s.t.* $\gamma(c) = id_c$, $s_0.cond(c) = \sigma_0(id_c)$.

Proof. Given a $c \in \mathcal{C}$ and an $id_c \in Ins(\Delta)$ *s.t.* $\gamma(c) = id_c$, let us show that $\boxed{s_0.cond(c) = \sigma_0(id_c).}$

Rewriting $s_0.cond(c)$ as `false`, by definition of s_0, $\boxed{\sigma_0(id_c) = \texttt{false}.}$
By construction, id_c is an input port identifier of Boolean type in the $\mathcal{H}$-VHDL design d, and thus, by property of the $\mathcal{H}$-VHDL elaboration relation, we can deduce $\sigma_e(id_c) = \texttt{false}$.
By property of the $\mathcal{H}$-VHDL initialization relation and $id_c \in Ins(\Delta)$, we can deduce $\sigma_e(id_c) = \sigma_0(id_c)$.
Rewriting $\sigma_0(id_c)$ as $\sigma_e(id_c)$ and $\sigma_e(id_c)$ as `false`, tautology.

□

C.1.5 Initial states and action executions

Lemma 12 (Initial states equal action executions). *For all* $sitpn \in SITPN$, $d \in design$, $\gamma \in WM(sitpn, d)$, $\Delta \in ElDesign(d, \mathcal{D}_\mathcal{H}), \sigma_e, \sigma_0 \in \Sigma(\Delta)$ *that verify the hypotheses of Definition 34, then* $\forall a \in \mathcal{A}, id_a \in Outs(\Delta)$ *s.t.* $\gamma(a) = id_a$, $s_0.ex(a) = \sigma_0(id_a)$.

Proof. Given a $a \in \mathcal{A}$ and an $id_a \in Outs(\Delta)$ *s.t.* $\gamma(a) = id_a$, let us show that $\boxed{s_0.ex(a) = \sigma_0(id_a).}$

Rewriting $s_0.ex(a)$ as `false`, by definition of s_0, $\boxed{\sigma_0(id_a) = \texttt{false}.}$
By construction, id_a is an output port identifier of Boolean type in the $\mathcal{H}$-VHDL design d. Moreover, we know that the output port identifier id_a is assigned to `false` in the generated `action` process during the initialization phase (i.e. the assignment is a part of a *reset* block). Thus, we can deduce that $\sigma_0(id_a) = \texttt{false}$.
Rewriting $\sigma_0(id_a)$ as `false`, tautology.

□

C.1.6 Initial states and function executions

Lemma 13 (Initial states equal function executions). *For all* $sitpn \in SITPN$, $d \in design$, $\gamma \in WM(sitpn, d)$, $\Delta \in ElDesign(d, \mathcal{D}_\mathcal{H}), \sigma_e, \sigma_0 \in \Sigma(\Delta)$ *that verify the hypotheses of Definition 34, then* $\forall f \in \mathcal{F}, id_f \in Outs(\Delta)$ *s.t.* $\gamma(f) = id_f$, $s_0.ex(f) = \sigma_0(id_f)$.

Proof. Given a $f \in \mathcal{F}$ and an $id_f \in Outs(\Delta)$ *s.t.* $\gamma(f) = id_f$, let us show that $\boxed{s_0.ex(f) = \sigma_0(id_f).}$

Rewriting $s_0.ex(f)$ as `false`, by definition of s_0, $\boxed{\sigma_0(id_f) = \texttt{false}.}$
By construction, id_f is an output port identifier of Boolean type in the $\mathcal{H}$-VHDL design d, and thus, by property of the $\mathcal{H}$-VHDL elaboration relation, we can deduce $\sigma_e(id_f) = \texttt{false}$.
By construction, and by property of the initialization relation, we know that the output port identifier id_f is assigned to `false` in the generated `function` process during the initialization phase (i.e. the assignment is a part of a *reset* block). Thus, we can deduce $\sigma_0(id_f) = \texttt{false}$.
Rewriting $\sigma_0(id_f)$ as `false`, tautology.

□

C.1.7 Initial states and fired transitions

Lemma 14 (No fired at initial state). $\forall d \in design, \Delta \in ElDesign(d, \mathcal{D}_\mathcal{H}), \sigma_e, \sigma_0 \in \Sigma(\Delta), id_t \in Comps(\Delta), gm_t, ipm_t, opm_t$ *s.t. :*

- $\mathcal{D}_\mathcal{H}, \emptyset \vdash d.cs \xrightarrow{elab} \sigma_0$
- $\Delta, \sigma_e \vdash d.cs \xrightarrow{init} \sigma_0$
- $\texttt{comp}(id_t, "transition", gm_t, ipm_t, opm_t) \in d.cs$

then $\sigma_0(id_t)("fired") = \texttt{false}$.

Proof. Assuming all the above hypotheses, let us show $\boxed{\sigma_0(id_t)("fired") = \texttt{false}.}$
By property of the initialization relation, $\texttt{comp}(id_t, "transition", gm_t, ipm_t, opm_t) \in d.cs$, and through the examination of the `fired_evaluation` process defined in the transition design architecture, we can deduce:

$$\sigma_0(id_t)("fired") = \sigma_0(id_t)("s\_firable") \,.\, \sigma_0(id_t)("s\_priority\_combination") \tag{C.2}$$

Rewriting the goal with Equation (C.2): $\boxed{\sigma_0(id_t)("sfa") \,.\, \sigma_0(id_t)("spc") = \texttt{false}.}$
By property of the initialization relation, $\texttt{comp}(id_t, "transition", gm_t, ipm_t, opm_t) \in d.cs$, and through the examination of the `firable` process defined in the transition design architecture, we can deduce $\sigma_0(id_t)("sfa") = \texttt{false}$.
Rewriting the goal with $\sigma_0(id_t)("sfa") = \texttt{false}$ and simplifying the goal, tautology. □

C.2 First Rising Edge

Definition 35 (First rising edge hypotheses). *Given an* $sitpn \in SITPN, d \in design, \gamma \in WM(sitpn, d), \Delta \in ElDesign(d, \mathcal{D}_\mathcal{H}), \sigma_e, \sigma_0, \sigma_i, \sigma_\uparrow, \sigma \in \Sigma(\Delta), E_c \in \mathbb{N} \to \mathcal{C} \to \mathbb{B}, E_p \in (\mathbb{N} \times \{\uparrow, \downarrow\}) \to Ins(\Delta) \to value, \tau \in \mathbb{N}$, *assume that:*

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

- σ_0 *is the initial state of* Δ: $\Delta, \sigma_e \vdash d.cs \xrightarrow{init} \sigma_0$
- $E_c, \tau \vdash s_0 \xrightarrow{\uparrow_0} s_0$
- $\texttt{Inject}_{\uparrow}(\sigma_0, E_p, \tau, \sigma_i)$ *and* $\Delta, \sigma_i \vdash \text{d.cs} \xrightarrow{\uparrow} \sigma_{\uparrow}$ *and* $\Delta, \sigma_{\uparrow} \vdash \text{d.cs} \xrightarrow{\theta} \sigma$

Lemma 15 (First rising edge). *For all* $sitpn, d, \gamma, \Delta, \sigma_e, \sigma_0, \sigma_i, \sigma_{\uparrow}, \sigma, E_c, E_p, \tau$ *that verify the hypotheses of Definition 35, then* $\gamma, E_c, \tau \vdash s_0 \overset{\uparrow}{\approx} \sigma$.

Proof. By definition of the Full post rising edge state similarity relation, there are 8 points to prove.

1. $\forall p \in P, id_p \in Comps(\Delta)\ s.t.\ \gamma(p) = id_p,\ s_0.M(p) = \sigma(id_p)(\text{"s\_marking"})$.
2. $\forall t \in T_i, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t,$
   $(upper(I_s(t)) = \infty \wedge s_0.I(t) \leq lower(I_s(t)) \Rightarrow s_0.I(t) = \sigma(id_t)(\text{"s\_time\_counter"}))$
   $\wedge (upper(I_s(t)) = \infty \wedge s_0.I(t) > lower(I_s(t)) \Rightarrow \sigma(id_t)(\text{"s\_time\_counter"}) = lower(I_s(t)))$
   $\wedge (upper(I_s(t)) \neq \infty \wedge s_0.I(t) > upper(I_s(t)) \Rightarrow \sigma(id_t)(\text{"s\_time\_counter"}) = upper(I_s(t)))$
   $\wedge (upper(I_s(t)) \neq \infty \wedge s_0.I(t) \leq upper(I_s(t)) \Rightarrow s_0.I(t) = \sigma(id_t)(\text{"s\_time\_counter"}))$.
3. $\forall t \in T_i, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t, s_0.reset_t(t) = \sigma(id_t)(\text{"s\_reinit\_time\_counter"})$.
4. $\forall a \in \mathcal{A}, id_a \in Outs(\Delta)\ s.t.\ \gamma(a) = id_a,\ s_0.ex(a) = \sigma(id_a)$.
5. $\forall f \in \mathcal{F}, id_f \in Outs(\Delta)\ s.t.\ \gamma(f) = id_f,\ s_0.ex(f) = \sigma(id_f)$.
6. $\forall t \in T, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t, t \in Sens(s_0.M) \Leftrightarrow \sigma(id_t)(\text{"s\_enabled"}) = \texttt{true}$.
7. $\forall t \in T, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t, t \notin Sens(s_0.M) \Leftrightarrow \sigma(id_t)(\text{"s\_enabled"}) = \texttt{false}$.
8. $\forall t \in T, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t,$
   $$\sigma(id_t)(\text{"s\_condition\_combination"}) = \prod_{c \in conds(t)} \begin{cases} E_c(\tau, c) & if\ \mathbb{C}(t,c) = 1 \\ \texttt{not}(E_c(\tau, c)) & if\ \mathbb{C}(t,c) = -1 \end{cases}$$
   where $conds(t) = \{c \in \mathcal{C} \mid \mathbb{C}(t,c) = 1 \vee \mathbb{C}(t,c) = -1\}$.

- Apply the First rising edge equal marking lemma to solve 1.
- Apply the First rising edge equal time counters lemma to solve 2.
- Apply the First rising edge equal reset orders lemma to solve 3.
- Apply the First rising edge equal action executions lemma to solve 4.
- Apply the First rising edge equal function executions lemma to solve 5.

- Apply the First rising edge equal sensitized lemma to solve 6.
- Apply the First rising edge not equal sensitized lemma to solve 7.
- Apply the First rising edge equal condition combination lemma to solve 8.

□

C.2.1 First rising edge and marking

Lemma 16 (First rising edge equal marking). *For all* $sitpn, d, \gamma, \Delta, \sigma_e, \sigma_0, \sigma_i, \sigma_\uparrow, \sigma, E_c, E_p, \tau$ *that verify the hypotheses of Definition 35, then* $\forall p \in P, id_p \in Comps(\Delta)$ *s.t.* $\gamma(p) = id_p$, $s_0.M(p) = \sigma(id_p)(\text{"}s\_marking\text{"})$.

Proof. Given a p and an id_p s.t. $\gamma(p) = id_p$, let us show that $\boxed{s_0.M(p) = \sigma(id_p)(\text{"}s\_marking\text{"}).}$
By construction and by definition of id_p, there exist gm_p, ipm_p, opm_p s.t. $\texttt{comp}(id_p, \text{"}place\text{"}, gm_p, ipm_p, opm_p) \in d.cs$.
By property of the $\texttt{Inject}_\uparrow$ relation, the $\mathcal{H}$-VHDL rising edge relation, the stabilize relation, $\texttt{comp}(id_p, \text{"}place\text{"}, gm_p, ipm_p, opm_p) \in d.cs$, and through the examination of the `marking` process defined in the place design architecture, we can deduce:

$$\sigma(id_p)(\text{"}sm\text{"}) = \sigma_0(id_p)(\text{"}sm\text{"}) + \sigma_0(id_p)(\text{"}sits\text{"}) - \sigma_0(id_p)(\text{"}sots\text{"}) \tag{C.3}$$

Rewriting the goal with Equation (C.3):
$\boxed{s_0.M(p) = \sigma_0(id_p)(\text{"}sm\text{"}) + \sigma_0(id_p)(\text{"}sits\text{"}) - \sigma_0(id_p)(\text{"}sots\text{"}).}$
Appealing to Lemmas 7 and 8, we can deduce $\sigma_0(id_p)(\text{"}sits\text{"}) = 0$ and $\sigma_0(id_p)(\text{"}sots\text{"}) = 0$.
Rewriting the goal with $\sigma_0(id_p)(\text{"}sits\text{"}) = 0$ and $\sigma_0(id_p)(\text{"}sots\text{"}) = 0$, $\boxed{s_0.M(p) = \sigma_0(id_p)(\text{"}sm\text{"}).}$

Appealing to Lemma 6, $s_0.M(p) = \sigma_0(id_p)(\text{"}sm\text{"})$. □

C.2.2 First rising edge and time counters

Lemma 17 (First rising edge equal time counters). *For all* $sitpn, d, \gamma, \Delta, \sigma_e, \sigma_0, \sigma_i, \sigma_\uparrow, \sigma, E_c, E_p, \tau$ *that verify the hypotheses of Definition 35, then*
$\forall t \in T_i, id_t \in Comps(\Delta)$ *s.t.* $\gamma(t) = id_t$,
$upper(I_s(t)) = \infty \wedge s_0.I(t) \leq lower(I_s(t)) \Rightarrow s_0.I(t) = \sigma(id_t)(\text{"}s\_time\_counter\text{"}) \wedge$
$upper(I_s(t)) = \infty \wedge s_0.I(t) > lower(I_s(t)) \Rightarrow \sigma(id_t)(\text{"}s\_time\_counter\text{"}) = lower(I_s(t)) \wedge$
$upper(I_s(t)) \neq \infty \wedge s_0.I(t) > upper(I_s(t)) \Rightarrow \sigma(id_t)(\text{"}s\_time\_counter\text{"}) = upper(I_s(t)) \wedge$
$upper(I_s(t)) \neq \infty \wedge s_0.I(t) \leq upper(I_s(t)) \Rightarrow s_0.I(t) = \sigma(id_t)(\text{"}s\_time\_counter\text{"})$.

Proof. Given a $t \in T_i$ and an $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, let us show that:

1. $\boxed{upper(I_s(t)) = \infty \wedge s_0.I(t) \leq lower(I_s(t)) \Rightarrow s_0.I(t) = \sigma(id_t)(\text{"}s\_time\_counter\text{"})}$
2. $\boxed{upper(I_s(t)) = \infty \wedge s_0.I(t) > lower(I_s(t)) \Rightarrow \sigma(id_t)(\text{"}s\_time\_counter\text{"}) = lower(I_s(t))}$

3. $\boxed{upper(I_s(t)) \neq \infty \wedge s_0.I(t) > upper(I_s(t)) \Rightarrow \sigma(id_t)(\text{"s\_time\_counter"}) = upper(I_s(t))}$

4. $\boxed{upper(I_s(t)) \neq \infty \wedge s_0.I(t) \leq upper(I_s(t)) \Rightarrow s_0.I(t) = \sigma(id_t)(\text{"s\_time\_counter"})}$

By construction and by definition of id_t, there exist gm_t, ipm_t, opm_t s.t. $\texttt{comp}(id_t, \text{"transition"}, gm_t, ipm_t, opm_t) \in d.cs$.
Then, let us show the 4 previous points:

1. Assuming that $upper(I_s(t)) = \infty$ and $s_0.I(t) \leq lower(I_s(t))$, let us show $\boxed{s_0.I(t) = \sigma(id_t)(\text{"stc"}).}$

   By property of the $\texttt{Inject}_{\uparrow}$ relation, the $\mathcal{H}$-VHDL rising edge and stabilize relations, and $\texttt{comp}(id_t, \text{"transition"}, gm_t, ipm_t, opm_t) \in d.cs$, we can deduce $\sigma(id_t)(\text{"stc"}) = \sigma_0(id_t)(\text{"stc"})$.

   Rewriting $\sigma(id_t)(\text{"stc"})$ as $\sigma_0(id_t)(\text{"stc"})$, $\boxed{s_0.I(t) = \sigma_0(id_t)(\text{"stc"}).}$

   Appealing to Lemma 9, $s_0.I(t) = \sigma_0(id_t)(\text{"stc"})$.

2. Assuming that $upper(I_s(t)) = \infty$ and $s_0.I(t) > lower(I_s(t))$, let us show $\boxed{\sigma(id_t)(\text{"stc"}) = lower(I_s(t)).}$

   By definition, $lower(I_s(t)) \in \mathbb{N}^*$ and $s_0.I(t) = 0$. Then, $lower(I_s(t)) < 0$ is a contradiction.

3. Assuming that $upper(I_s(t)) \neq \infty$ and $s_0.I(t) > upper(I_s(t))$, let us show $\boxed{\sigma(id_t)(\text{"stc"}) = upper(I_s(t))}$.

   By definition, $upper(I_s(t)) \in \mathbb{N}^*$ and $s_0.I(t) = 0$. Then, $upper(I_s(t)) < 0$ is a contradiction.

4. Assuming that $upper(I_s(t)) \neq \infty$ and $s_0.I(t) \leq upper(I_s(t))$, let us show $\boxed{s_0.I(t) = \sigma(id_t)(\text{"stc"}).}$

   By property of the $\texttt{Inject}_{\uparrow}$ relation, the $\mathcal{H}$-VHDL rising edge and stabilize relations, and $\texttt{comp}(id_t, \text{"transition"}, gm_t, ipm_t, opm_t) \in d.cs$, we can deduce $\sigma(id_t)(\text{"stc"}) = \sigma_0(id_t)(\text{"stc"})$.

   Rewriting $\sigma(id_t)(\text{"stc"})$ as $\sigma_0(id_t)(\text{"stc"})$, $\boxed{s_0.I(t) = \sigma_0(id_t)(\text{"stc"}).}$

   Appealing to Lemma 9, $s_0.I(t) = \sigma_0(id_t)(\text{"stc"})$.

□

C.2.3 First rising edge and reset orders

Lemma 18 (First rising edge equal reset orders). *For all $sitpn, d, \gamma, \Delta, \sigma_e, \sigma_0, \sigma_i, \sigma_{\uparrow}, \sigma, E_c, E_p, \tau$ that verify the hypotheses of Definition 35, then*
$\forall t \in T, id_t \in Comps(\Delta)$ *s.t.* $\gamma(t) = id_t, s_0.reset_t(t) = \sigma(id_t)(\text{"s\_reinit\_time\_counter"})$.

Proof. Given a $t \in T$ and an $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, let us show that $\boxed{s_0.reset_t(t) = \sigma(id_t)(\text{"}srtc\text{"}).}$
By construction and by definition of id_t, there exist gm_t, ipm_t, opm_t s.t. $\texttt{comp}(id_t, \text{"}transition\text{"}, gm_t, ipm_t, opm_t) \in d.cs$.
By property of the stabilize relation, $\texttt{comp}(id_t, \text{"}transition\text{"}, gm_t, ipm_t, opm_t) \in d.cs$, and through the examination of the `reinit_time_counter_evaluation` process defined in the transition design architecture, we can deduce:

$$\sigma(id_t)(\text{"}srtc\text{"}) = \sum_{i=0}^{\Delta(id_t)(\text{"}input\_arcs\_number\text{"})-1} \sigma(id_t)(\text{"}reinit\_time\text{"})[i] \tag{C.4}$$

Rewriting the goal with Equation (C.4): $\boxed{s_0.reset_t(t) = \sum_{i=0}^{\Delta(id_t)(\text{"}ian\text{"})-1} \sigma(id_t)(\text{"}rt\text{"})[i].}$
Let us perform case analysis on $input(t)$; there are two cases:

- **CASE** $input(t) = \emptyset$:

  By construction, $<\texttt{input\_arcs\_number} \Rightarrow 1> \in gm_t$, and by property of the $\mathcal{H}$-VHDL elaboration relation, we can deduce $\Delta(id_t)(\text{"}ian\text{"}) = 1$.

  By construction, $< \texttt{reinit\_time}(0) \Rightarrow \texttt{false} > \in ipm_t$, and by property of the $\mathcal{H}$-VHDL stabilize relation, $\sigma(id_t)(\text{"}rt\text{"})[0] = \texttt{false}$.

  Rewriting the goal with $\Delta(id_t)(\text{"}ian\text{"}) = 1$ and $\sigma(id_t)(\text{"}rt\text{"})[0] = \texttt{false}$, $\boxed{s_0.reset_t(t) = \texttt{false}.}$

  By definition of s_0, $s_0.reset_t(t) = \texttt{false}.$

- **CASE** $input(t) \neq \emptyset$:

  By construction, $<\texttt{input\_arcs\_number} \Rightarrow |input(t)|> \in gm_t$, and by property of the $\mathcal{H}$-VHDL elaboration relation, we can deduce $\Delta(id_t)(\text{"}ian\text{"}) = |input(t)|$.

  Rewriting $\Delta(id_t)(\text{"}ian\text{"})$ as $|input(t)|$, $\boxed{s_0.reset_t(t) = \sum_{i=0}^{|input(t)|-1} \sigma(id_t)(\text{"}rt\text{"})[i].}$

  By definition of s_0, $s_0.reset_t(t) = \texttt{false}$. Rewriting $s_0.reset_t(t)$ as $\texttt{false}$, $\boxed{\sum_{i=0}^{|input(t)|-1} \sigma(id_t)(\text{"}rt\text{"})[i] = \texttt{false}.}$

  Given a $i \in [0, |input(t)| - 1]$, let us show $\boxed{\sigma(id_t)(\text{"}rt\text{"})[i] = \texttt{false}.}$

  By construction, and since $input(t) \neq \emptyset$, there exist a $p \in input(t)$, an $id_p \in Comps(\Delta)$ s.t. $\gamma(p) = id_p$, a gm_p, an ipm_p, an opm_p s.t. $\texttt{comp}(id_p, \text{"}place\text{"}, gm_p, ipm_p, opm_p) \in d.cs$, and there exist a $j \in [0, |output(p)| - 1]$ and an $id_{ji} \in Sigs(\Delta)$ s.t. $<\texttt{reinit\_transition\_time(j)} \Rightarrow \texttt{id}_{\texttt{ji}}$ opm_p and $<\texttt{reinit\_time(i)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in ipm_t$.

  By property of the stabilize relation, $<\texttt{reinit\_transition\_time(j)} \Rightarrow id_{ji}> \in opm_p$ and $<\texttt{reinit\_time(i)} \Rightarrow id_{ji}> \in ipm_t$, we can deduce $\sigma(id_t)(\text{"}rt\text{"})[i] = \sigma(id_{ji}) = \sigma(id_p)(\text{"}rtt\text{"})[j]$.

Rewriting $\sigma(id_t)("rt")[i]$ as $\sigma(id_{ji})$ and $\sigma(id_{ji})$ as $\sigma(id_p)("rtt")[j]$, $\boxed{\sigma(id_p)("rtt")[j] = \texttt{false}.}$

By property of the $\mathcal{H}$-VHDL rising edge and stabilize relations, $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$, and through the examination of the process defined in the place design architecture, we can deduce:

$$\begin{aligned}\sigma(id_p)("rtt")[j] =&((\sigma_0(id_p)("oat")[j] = \texttt{basic} + \sigma_0(id_p)("oat")[j] = \texttt{test})\\ &.(\sigma_0(id_p)("sm") - \sigma_0(id_p)("sots") < \sigma_0(id_p)("oaw")[j])\\ &.(\sigma_0(id_p)("sots") > 0))\\ &+ (\sigma_0(id_p)("otf")[j])\end{aligned} \tag{C.5}$$

Rewriting the goal with Equation (C.5),

$$\boxed{\begin{aligned}\texttt{false} =&((\sigma_0(id_p)("oat")[j] = \texttt{basic} + \sigma_0(id_p)("oat")[j] = \texttt{test})\\ &.(\sigma_0(id_p)("sm") - \sigma_0(id_p)("sots") < \sigma_0(id_p)("oaw")[j])\\ &.(\sigma_0(id_p)("sots") > 0))\\ &+ (\sigma_0(id_p)("otf")[j])\end{aligned}}$$

By construction, there exists an $id_{fj} \in Sigs(\Delta)$ s.t. $<\texttt{fired}\Rightarrow \texttt{id}_{\texttt{fj}}> \in opm_t$ and $<\texttt{output\_transitions\_fired(j)}\Rightarrow \texttt{id}_{\texttt{fj}}> \in ipm_p$.

By property of the initialization relation, $<\texttt{fired}\Rightarrow \texttt{id}_{\texttt{fj}}> \in opm_t$ and $<\texttt{output\_transitions\_fired(j)}\Rightarrow \texttt{id}_{\texttt{fj}}> \in ipm_p$, we can deduce $\sigma_0(id_p)("otf")[j] = \sigma_0(id_{fj}) = \sigma_0(id_t)("fired")$.

Appealing to Lemma 14, we can deduce $\sigma_0(id_t)("fired") = \texttt{false}$ and consequently $\sigma_0(id_p)("otf")[j] = \texttt{false}$.

Rewriting $\sigma_0(id_p)("otf")[j]$ as $\texttt{false}$ and simplifying the goal,

$$\boxed{\begin{aligned}false =&((\sigma_0(id_p)("oat")[j] = \texttt{BASIC} + \sigma_0(id_p)("oat")[j] = \texttt{TEST})\\ &.(\sigma_0(id_p)("sm") - \sigma_0(id_p)("sots") < \sigma_0(id_p)("oaw")[j])\\ &.(\sigma_0(id_p)("sots") > 0))\end{aligned}}$$

Appealing to Lemma 8, we can deduce $\sigma_0(id_p)("sots") = 0$.

Rewriting $\sigma_0(id_p)("sots")$ as 0 and simplifying the goal, tautology.

□

C.2.4 First rising edge and action executions

Lemma 19 (First rising edge equal action executions). *For all $sitpn, d, \gamma, \Delta, \sigma_e, \sigma_0, \sigma_i, \sigma_\uparrow, \sigma, E_c, E_p, \tau$ that verify the hypotheses of Definition 35, then*
$\forall a \in \mathcal{A}, id_a \in Outs(\Delta)$ *s.t.* $\gamma(a) = id_a$, $s_0.ex(a) = \sigma(id_a)$.

Proof. Given an $a \in \mathcal{A}$ and an $id_a \in Outs(\Delta)$ *s.t.* $\gamma(a) = id_a$, let us show that $s_0.ex(a) = \sigma(id_a)$.
By construction, id_a is an output port identifier of Boolean type in the $\mathcal{H}$-VHDL design d. The generated `action` process assigns a value to the output port id_a only during the initialization phase or a falling edge phase.
By property of the $\texttt{Inject}_{\uparrow}$, $\mathcal{H}$-VHDL rising edge and stabilize relations, we can deduce $\sigma(id_a) = \sigma_0(id_a)$.
Rewriting $\sigma(id_a)$ as $\sigma_0(id_a)$, $s_0.ex(a) = \sigma_0(id_a)$. Appealing to Lemma 12, $s_0.ex(a) = \sigma_0(id_a)$. □

C.2.5 First rising edge and function executions

Lemma 20 (First rising edge equal function executions). *For all* $sitpn, d, \gamma, \Delta, \sigma_e, \sigma_0, \sigma_i, \sigma_{\uparrow}, \sigma, E_c, E_p, \tau$ *that verify the hypotheses of Definition 35, then*
$\forall f \in \mathcal{F}, id_f \in Outs(\Delta)$ *s.t.* $\gamma(f) = id_f$, $s_0.ex(f) = \sigma(id_f)$.

Proof. Given an $f \in \mathcal{F}$ and an $id_f \in Outs(\Delta)$ s.t. $\gamma(f) = id_f$, let us show that $s_0.ex(f) = \sigma(id_f)$.
Rewriting $s_0.ex(f)$ as `false`, by definition of s_0, $\sigma(id_f) = \texttt{false}$.
By construction, id_f is an output port identifier of Boolean type in the $\mathcal{H}$-VHDL design d. The generated `function` process assigns a value to the output port id_f only during the initialization phase or during a rising edge phase.
By construction, the `function` process is defined in the behavior of design d, i.e. $\texttt{ps}(\text{"}function\text{"}, \emptyset, sl, ss) \in d.cs$.
Let $trs(f)$ be the set of transitions associated to function f, i.e $trs(f) = \{t \in T \mid \mathbb{F}(t, f) = true\}$.
Let us perform case analysis on $trs(f)$; there are two cases:

- **CASE** $trs(f) = \emptyset$:

  By construction, $\texttt{id}_\texttt{f} \Leftarrow \texttt{false} \in ss_{\uparrow}$ where $ss_{\uparrow}$ is the part of the ''`function`'' process body executed during a rising edge phase (i.e. a rising edge block statement).

  By property of the $\mathcal{H}$-VHDL rising edge and the stabilize relation, $\sigma(id_f) = \texttt{false}$.

- **CASE** $trs(f) \neq \emptyset$:

  By construction, $\texttt{id}_\texttt{f} \Leftarrow \texttt{id}_{\texttt{ft}_0} + \cdots + \texttt{id}_{\texttt{ft}_n} \in ss_{\uparrow}$ where $ss_{\uparrow}$ is the part of the ''`function`'' process body executed during the rising edge phase, and $n = |trs(f)| - 1$, and for all $i \in [0, n-1]$, id_{ft_i} is a internal signal of design d.

  By property of the $\texttt{Inject}_{\uparrow}$, the $\mathcal{H}$-VHDL rising edge and stabilize relations, we can deduce $\sigma(id_f) = \sigma_0(id_{ft_0}) + \cdots + \sigma_0(id_{ft_n})$.

  Rewriting $\sigma(id_f)$ as $\sigma_0(id_{ft_0}) + \cdots + \sigma_0(id_{ft_n})$, $\sigma_0(id_{ft_0}) + \cdots + \sigma_0(id_{ft_n}) = \texttt{false}$.

  By construction, for all id_{ft_i}, there exist a $t_i \in trs(f)$ and an id_{t_i} s.t. $\gamma(t_i) = id_{t_i}$.

  By construction and by definition of id_{t_i}, there exist gm_{t_i}, ipm_{t_i} and opm_{t_i} s.t. $\texttt{comp}(id_{t_i}, \text{"}transition\text{"}, gm_{t_i}, ipm_{t_i}, opm_{t_i}) \in d.cs$.

By construction, $<\texttt{fired} \Rightarrow \texttt{id}_{\texttt{ft}_i}> \in opm_{t_i}$, and by property of the initialization relation $\sigma_0(id_{ft_i}) = \sigma_0(id_{t_i})("fired")$.

Rewriting $\sigma_0(id_{ft_i})$ as $\sigma_0(id_{t_i})("fired")$, $\boxed{\sigma_0(id_{t_0})("fired") + \cdots + \sigma_0(id_{t_n})("fired") = \texttt{false}.}$

Appealing to Lemma 14, we can deduce $\sigma_0(id_{t_i})("fired") = \texttt{false}$.

Rewriting all $\sigma_0(id_{t_i})("fired")$ as $\texttt{false}$ and simplifying the goal, tautology.

□

C.2.6 First rising edge and sensitization

Lemma 21 (First rising edge equal sensitized). *For all $sitpn, d, \gamma, \Delta, \sigma_e, \sigma_0, \sigma_i, \sigma_\uparrow, \sigma, E_c, E_p, \tau$ that verify the hypotheses of Definition 35, then*
$\forall t \in T, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t, t \in Sens(s_0.M) \Leftrightarrow \sigma(id_t)("s\_enabled") = \texttt{true}.$

Proof. See the proof of Lemma 30. □

Lemma 22 (First rising edge not equal sensitized). *For all $sitpn, d, \gamma, \Delta, \sigma_e, \sigma_0, \sigma_i, \sigma_\uparrow, \sigma, E_c, E_p, \tau$ that verify the hypotheses of Definition 35, then*
$\forall t \in T, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t, t \notin Sens(s_0.M) \Leftrightarrow \sigma(id_t)("s\_enabled") = \texttt{false}.$

Proof. See the proof of Lemma 31. □

C.2.7 First rising edge and condition combination

Lemma 23 (First rising edge equal condition combination). *For all $sitpn, d, \gamma, \Delta, \sigma_e, \sigma_0, \sigma_i, \sigma_\uparrow, \sigma, E_c, E_p, \tau$ that verify the hypotheses of Definition 35, then*
$\forall t \in T, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t,$

$$\sigma(id_t)("s\_condition\_combination") = \prod_{c \in conds(t)} \begin{cases} E_c(\tau, c) & if\ \mathbb{C}(t,c) = 1 \\ \texttt{not}(E_c(\tau, c)) & if\ \mathbb{C}(t,c) = -1 \end{cases}$$

where $conds(t) = \{c \in \mathcal{C} \mid \mathbb{C}(t,c) = 1 \vee \mathbb{C}(t,c) = -1\}$.

Proof. See the proof of Lemma 25. □

C.3 Rising Edge

Definition 36 (Rising edge hypotheses). *Given an $sitpn \in SITPN, d \in design, \gamma \in WM(sitpn, d)$, $E_c \in \mathbb{N} \to \mathcal{C} \to \mathbb{B}$, $\Delta \in ElDesign(d, \mathcal{D}_\mathcal{H})$, $E_p \in (\mathbb{N} \times \{\uparrow, \downarrow\}) \to Ins(\Delta) \to value$, $\tau \in \mathbb{N}$, $s, s' \in S(sitpn)$, $\sigma_e, \sigma, \sigma_i, \sigma_\uparrow, \sigma' \in \Sigma(\Delta)$, assume that:*

- $\lfloor sitpn \rfloor_\mathcal{H} = (d, \gamma)$ *and* $\gamma \vdash E_p \overset{env}{=} E_c$ *and* $\mathcal{D}_\mathcal{H}, \emptyset \vdash \mathrm{d} \xrightarrow{elab} \Delta, \sigma_e$
- $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$

- $E_c, \tau \vdash s \xrightarrow{\uparrow} s'$
- $\texttt{Inject}_{\uparrow}(\sigma, E_p, \tau, \sigma_i)$ *and* $\mathcal{D}_{\mathcal{H}}, \Delta, \sigma_i \vdash \text{d.cs} \xrightarrow{\uparrow} \sigma_{\uparrow}$ *and* $\mathcal{D}_{\mathcal{H}}, \Delta, \sigma_{\uparrow} \vdash \text{d.cs} \xrightarrow{\rightsquigarrow} \sigma'$
- *State* σ *is a stable design state:* $\mathcal{D}_{\mathcal{H}}, \Delta, \sigma \vdash \text{d.cs} \xrightarrow{comb} \sigma$

C.3.1 Rising edge and Marking

Lemma 24 (Rising edge equal marking). *For all* $sitpn$, d, γ, E_c, E_p, τ, Δ, σ_e, s, s', σ, σ_i, $\sigma_{\uparrow}$, σ' *that verify the hypotheses of Def. 36, then* $\forall p, id_p$ *s.t.* $\gamma(p) = id_p$, $s'.M(p) = \sigma'(id_p)(\text{"s\_marking"})$.

Proof. Given a $p \in P$, let us show $\boxed{s'.M(p) = \sigma'(id_p)(\text{"s\_marking"}).}$
By construction and by definition of id_p, there exist gm_p, ipm_p, opm_p s.t. $\texttt{comp}(id_p, \text{"place"}, gm_p, ipm_p, opm_p) \in d.cs$.
By definition of the SITPN state transition relation on rising edge:

$$s'.M(p) = s.M(p) - \sum_{t \in Fired(s)} pre(p,t) + \sum_{t \in Fired(s)} post(t,p) \tag{C.6}$$

By property of the $\texttt{Inject}_{\uparrow}$, the $\mathcal{H}$-VHDL rising edge and the stabilize relations, $\texttt{comp}(id_p, \text{"place"}, gm_p, ipm_p, opm_p) \in d.cs$, and through the examination of the `marking` process defined in the place design architecture, we can deduce:

$$\begin{aligned}\sigma'(id_p)(\text{"sm"}) = \sigma(id_p)(\text{"sm"}) - \sigma(id_p)(\text{"s\_output\_token\_sum"}) \\ + \sigma(id_p)(\text{"s\_input\_token\_sum"})\end{aligned} \tag{C.7}$$

Rewriting the goal with C.6 and C.7,

$$\boxed{s.M(p) - \sum_{t \in Fired(s)} pre(p,t) + \sum_{t \in Fired(s)} post(t,p) = \sigma(id_p)(\text{"sm"}) - \sigma(id_p)(\text{"sots"}) + \sigma(id_p)(\text{"sits"}).}$$

By definition of the Full post falling edge state similarity relation, we can deduce $s.M(p) = \sigma(id_p)(\text{"sm"})$, $\sum_{t \in Fired(s)} pre(p,t) = \sigma(id_p)(\text{"sots"})$ and $\sum_{t \in Fired(s)} post(t,p) = \sigma(id_p)(\text{"sits"})$, and thus,

$$s.M(p) - \sum_{t \in Fired(s)} pre(p,t) + \sum_{t \in Fired(s)} post(t,p) = \sigma(id_p)(\text{"sm"}) - \sigma(id_p)(\text{"sots"}) + \sigma(id_p)(\text{"sits"}).$$

□

C.3.2 Rising edge and condition combination

Lemma 25 (Rising edge equal condition combination). *For all* $sitpn$, d, γ, E_c, E_p, τ, Δ, σ_e, s, s', σ, σ_i, $\sigma_{\uparrow}$, σ' *that verify the hypotheses of Def. 36, then*
$\forall t \in T, id_t \in Comps(\Delta)$ *s.t.* $\gamma(t) = id_t$,

$$\sigma'(id_t)(\text{"s\_condition\_combination"}) = \prod_{c \in conds(t)} \begin{cases} E_c(\tau, c) & if\ \mathbb{C}(t,c) = 1 \\ \texttt{not}(E_c(\tau, c)) & if\ \mathbb{C}(t,c) = -1 \end{cases}$$
where $conds(t) = \{c \in \mathcal{C} \mid \mathbb{C}(t,c) = 1 \vee \mathbb{C}(t,c) = -1\}$.

Proof. Given a t and an id_t s.t. $\gamma(t) = id_t$, let us show

$$\boxed{\sigma'(id_t)(\text{"s\_condition\_combination"}) = \prod_{c \in conds(t)} \begin{cases} E_c(\tau, c) & if\ \mathbb{C}(t,c) = 1 \\ \texttt{not}(E_c(\tau, c)) & if\ \mathbb{C}(t,c) = -1 \end{cases}}.$$

By construction and by definition of id_t, there exist gm_t, ipm_t, opm_t s.t. $\texttt{comp}(id_t, \text{"transition"}, gm_t, ipm_t, opm_t) \in d.cs$.
By property of the $\mathcal{H}$-VHDL stabilize relation, $\texttt{comp}(id_t, \text{"transition"}, gm_t, ipm_t, opm_t) \in d.cs$, and through the examination of the `condition_evaluation` process defined in the transition design architecture, we can deduce:

$$\sigma'(id_t)(\text{"scc"}) = \prod_{i=0}^{\Delta(id_t)(\text{"conditions\_number"})-1} \sigma'(id_t)(\text{"input\_conditions"})[i] \tag{C.8}$$

Rewriting the goal with C.8,

$$\boxed{\prod_{i=0}^{\Delta(id_t)(\text{"cn"})-1} \sigma'(id_t)(\text{"ic"})[i] = \prod_{c \in conds(t)} \begin{cases} E_c(\tau, c) & if\ \mathbb{C}(t,c) = 1 \\ \texttt{not}(E_c(\tau, c)) & if\ \mathbb{C}(t,c) = -1 \end{cases}}.$$

Let us perform case analysis on $conds(t)$; there are two cases:

- **CASE** $conds(t) = \emptyset$: $\boxed{\prod_{i=0}^{\Delta(id_t)(\text{"cn"})-1} \sigma'(id_t)(\text{"ic"})[i] = \texttt{true}.}$

  By construction, $<\texttt{conditions\_number} \Rightarrow 1> \in gm_t$ and $<\texttt{input\_conditions}(0) \Rightarrow \texttt{true}> \in ipm_t$.

  By property of the stabilize relation, $<\texttt{conditions\_number} \Rightarrow 1> \in gm_t$ and $<\texttt{input\_conditions(0)} \Rightarrow \texttt{true}> \in ipm_t$, we can deduce $\Delta(id_t)(\text{"cn"}) = 1$ and $\sigma'(id_t)(\text{"ic"})[0] = \texttt{true}$.

  Rewriting the goal with $\Delta(id_t)(\text{"cn"}) = 1$ and $\sigma'(id_t)(\text{"ic"})[0] = \texttt{true}$, tautology.

- **CASE** $conds(t) \neq \emptyset$:
  By construction, $<\texttt{conditions\_number} \Rightarrow |\texttt{conds(t)}|> \in gm_t$, and by property of the stabilize relation, we can deduce $\Delta(id_t)(\text{"cn"}) = |conds(t)|$.

  Rewriting the goal with $\Delta(id_t)(\text{"cn"}) = |conds(t)|$:

  $$\boxed{\prod_{i=0}^{|conds(t)|-1} \sigma'(id_t)(\text{"ic"})[i] = \prod_{c \in conds(t)} \begin{cases} E_c(\tau, c) & if\ \mathbb{C}(t,c) = 1 \\ \texttt{not}(E_c(\tau, c)) & if\ \mathbb{C}(t,c) = -1 \end{cases}}$$

  Let us reason by induction on the left product term:

– **BASE CASE**: $\prod_{i=0}^{|conds(t)|-1} \sigma'(id_t)("ic")[i] = 0$ and $|conds(t)| - 1 < 0$. Thus, we can deduce that $|conds(t)| = 0$ which contradicts $conds(t) \neq \emptyset$.

– **INDUCTION CASE**:

$$\forall conds' \subseteq \mathcal{C}, \prod_{i=1}^{|conds(t)|-1} \sigma'(id_t)("ic")[i] = \prod_{c \in conds'} \begin{cases} E_c(\tau, c) & if\ \mathbb{C}(t,c) = 1 \\ \texttt{not}(E_c(\tau, c)) & if\ \mathbb{C}(t,c) = -1 \end{cases}$$

$$\sigma'(id_t)("ic")[0] \,.\, \prod_{i=1}^{|conds(t)|-1} \sigma'(id_t)("ic")[i] = \prod_{c \in conds(t)} \begin{cases} E_c(\tau, c) & if\ \mathbb{C}(t,c) = 1 \\ \texttt{not}(E_c(\tau, c)) & if\ \mathbb{C}(t,c) = -1 \end{cases}$$

By construction, for all $i \in [0, |conds(t)| - 1]$, there exists $c \in conds(t)$ and an $id_c \in Ins(\Delta)$ such that

* $\gamma(c) = id_c$
* $\mathbb{C}(t,c) = 1$ implies $<\texttt{input\_conditions(i)} \Rightarrow \texttt{id}_\texttt{c}> \in ipm_t$
* $\mathbb{C}(t,c) = -1$ implies $<\texttt{input\_conditions(i)} \Rightarrow \texttt{not id}_\texttt{c}> \in ipm_t$

For $i = 0$, let us take such a $c \in conds(t)$ and an id_c with the above properties. By definition of $c \in conds(t)$, we can deduce $\mathbb{C}(t,c) = 1 \vee \mathbb{C}(t,c) = -1$. Let us perform case analysis on $\mathbb{C}(t,c) = 1 \vee \mathbb{C}(t,c) = -1$:

* **CASE** $\mathbb{C}(t,c) = 1$:
  Then, we have $<\texttt{input\_conditions(0)} \Rightarrow \texttt{id}_\texttt{c}> \in ipm_t$ and by property of the stabilize relation, we can deduce $\sigma(id_t)("ic")[0] = \sigma'(id_c)$.
  Rewriting the goal with $\sigma(id_t)("ic")[0] = \sigma'(id_c)$:
  $$\sigma'(id_c) \,.\, \prod_{i=1}^{|conds(t)|-1} \sigma'(id_t)("ic")[i] = \prod_{c \in conds(t)} \begin{cases} E_c(\tau, c) & if\ \mathbb{C}(t,c) = 1 \\ \texttt{not}(E_c(\tau, c)) & if\ \mathbb{C}(t,c) = -1 \end{cases}$$
  By property of the $\texttt{Inject}_\uparrow$ relation and $id_c \in Ins(\Delta)$, we can deduce $\sigma'(id_c) = E_p(\tau, \uparrow)(id_c)$.
  By property of $\gamma \vdash E_p \overset{env}{=} E_c$, we can deduce $E_p(\tau, \uparrow)(id_c) = E_c(\tau, c)$.
  Rewriting the goal with $\sigma'(id_c) = E_p(\tau, \uparrow)(id_c)$ and $E_p(\tau, \uparrow)(id_c) = E_c(\tau, c)$:
  $$E_c(\tau, c) \,.\, \prod_{i=1}^{|conds(t)|-1} \sigma'(id_t)("ic")[i] = \prod_{c \in conds(t)} \begin{cases} E_c(\tau, c) & if\ \mathbb{C}(t,c) = 1 \\ \texttt{not}(E_c(\tau, c)) & if\ \mathbb{C}(t,c) = -1 \end{cases}$$
  By definition of the $\prod$ operator, we can rewrite the right term of the goal as follows:
  $$E_c(\tau, c) \,.\, \prod_{i=1}^{|conds(t)|-1} \sigma'(id_t)("ic")[i] = E_c(\tau, c) \,.\, \prod_{c' \in conds(t) \setminus \{c\}} \begin{cases} E_c(\tau, c') & if\ \mathbb{C}(t,c') = 1 \\ \texttt{not}(E_c(\tau, c')) & if\ \mathbb{C}(t,c') = -1 \end{cases}$$
  Appealing to the induction hypothesis, tautology.
* **CASE** $\mathbb{C}(t,c) = -1$:
  Then, we have $<\texttt{input\_conditions(0)} \Rightarrow \texttt{not id}_\texttt{c}> \in ipm_t$ and by property of the stabilize relation, we can deduce $\sigma(id_t)("ic")[0] = \texttt{not}\ \sigma'(id_c)$.

Rewriting the goal with $\sigma(id_t)(''ic'')[0] = \texttt{not}\ \sigma'(id_c)$:

$$\boxed{\texttt{not}\ \sigma'(id_c)\ .\ \prod_{i=1}^{|conds(t)|-1}\sigma'(id_t)(''ic'')[i] = \prod_{c\in conds(t)}\begin{cases}E_c(\tau,c) & if\ \mathbb{C}(t,c)=1\\ \texttt{not}(E_c(\tau,c)) & if\ \mathbb{C}(t,c)=-1\end{cases}}$$

By property of the $\texttt{Inject}_{\uparrow}$ relation and $id_c \in Ins(\Delta)$, we can deduce $\sigma'(id_c) = E_p(\tau,\uparrow)(id_c)$.
By property of $\gamma \vdash E_p \overset{env}{=} E_c$, we can deduce $E_p(\tau,\uparrow)(id_c) = E_c(\tau,c)$.
Rewriting the goal with $\sigma'(id_c) = E_p(\tau,\uparrow)(id_c)$ and $E_p(\tau,\uparrow)(id_c) = E_c(\tau,c)$:

$$\boxed{\texttt{not}\ E_c(\tau,c)\ .\ \prod_{i=1}^{|conds(t)|-1}\sigma'(id_t)(''ic'')[i] = \prod_{c\in conds(t)}\begin{cases}E_c(\tau,c) & if\ \mathbb{C}(t,c)=1\\ \texttt{not}(E_c(\tau,c)) & if\ \mathbb{C}(t,c)=-1\end{cases}}$$

By definition of the $\prod$ operator, we can rewrite the right term of the goal as follows:

$$\boxed{\texttt{not}\ E_c(\tau,c)\ .\ \prod_{i=1}^{|conds(t)|-1}\sigma'(id_t)(''ic'')[i] = \texttt{not}\ E_c(\tau,c)\ .\ \prod_{c'\in conds(t)\setminus\{c\}}\begin{cases}E_c(\tau,c') & if\ \mathbb{C}(t,c')=\\ \texttt{not}(E_c(\tau,c')) & if\ \mathbb{C}(t,c')=\end{cases}}$$

Appealing to the induction hypothesis, tautology.

□

C.3.3 Rising edge and time counters

Lemma 26 (Rising edge equal time counters). *For all sitpn, d, γ, E_c, E_p, τ, Δ, σ_e, s, s', σ, σ_i, $\sigma_\uparrow$, σ' that verify the hypotheses of Def. 36, then*
$\forall t \in T_i, id_t \in Comps(\Delta)$ *s.t.* $\gamma(t) = id_t$,
$(upper(I_s(t)) = \infty \wedge s'.I(t) \leq lower(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(''s\_time\_counter''))$
$\wedge(upper(I_s(t)) = \infty \wedge s'.I(t) > lower(I_s(t)) \Rightarrow \sigma'(id_t)(''s\_time\_counter'') = lower(I_s(t)))$
$\wedge(upper(I_s(t)) \neq \infty \wedge s'.I(t) > upper(I_s(t)) \Rightarrow \sigma'(id_t)(''s\_time\_counter'') = upper(I_s(t)))$
$\wedge(upper(I_s(t)) \neq \infty \wedge s'.I(t) \leq upper(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(''s\_time\_counter''))$.

Proof. Given a $t \in T_i$ and an $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, let us show

$$\boxed{\begin{array}{l}(upper(I_s(t)) = \infty \wedge s'.I(t) \leq lower(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(''s\_time\_counter''))\\ \wedge(upper(I_s(t)) = \infty \wedge s'.I(t) > lower(I_s(t)) \Rightarrow \sigma'(id_t)(''s\_time\_counter'') = lower(I_s(t)))\\ \wedge(upper(I_s(t)) \neq \infty \wedge s'.I(t) > upper(I_s(t)) \Rightarrow \sigma'(id_t)(''s\_time\_counter'') = upper(I_s(t)))\\ \wedge(upper(I_s(t)) \neq \infty \wedge s'.I(t) \leq upper(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(''s\_time\_counter''))\end{array}}$$

By construction and by definition of id_t, there exist gm_t, ipm_t, opm_t s.t. $\texttt{comp}(id_t, ''transition'', gm_t, ipm_t, opm_t) \in d.cs$.
Then, there are 4 points to show:

1. $\boxed{upper(I_s(t)) = \infty \wedge s'.I(t) \leq lower(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(''s\_time\_counter'')}$

   Assuming that $upper(I_s(t)) = \infty$ and $s'.I(t) \leq lower(I_s(t))$, let us show
   $\boxed{s'.I(t) = \sigma'(id_t)(''s\_time\_counter'').}$

By property of the $\texttt{Inject}_{\uparrow}$, $\mathcal{H}$-VHDL rising edge and stabilize relations, $\texttt{comp}(id_t, "transition", gm_t, ipm_t, opm_t) \in d.cs$, and through the examination of the `time_counter` process defined in the transition design architecture, we can deduce $\sigma'(id_t)("stc") = \sigma(id_t)("stc")$.

By property of $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$, we can deduce $s.I(t) = \sigma(id_t)("stc")$.

Rewriting the goal with $\sigma'(id_t)("stc") = \sigma(id_t)("stc")$ and $s.I(t) = \sigma(id_t)("stc")$, tautology.

2. $\boxed{upper(I_s(t)) = \infty \wedge s'.I(t) > lower(I_s(t)) \Rightarrow \sigma'(id_t)("s\_time\_counter") = lower(I_s(t).}$

   Proved in the same fashion as 1.

3. $\boxed{upper(I_s(t)) \neq \infty \wedge s'.I(t) > upper(I_s(t)) \Rightarrow \sigma'(id_t)("s\_time\_counter") = upper(I_s(t).}$

   Proved in the same fashion as 1.

4. $\boxed{upper(I_s(t)) \neq \infty \wedge s'.I(t) \leq upper(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)("s\_time\_counter")}$

   Proved in the same fashion as 1.

□

C.3.4 Rising edge and reset orders

Lemma 27 (Rising edge equal reset orders). *For all $sitpn$, d, γ, E_c, E_p, τ, Δ, σ_e, s, s', σ, σ_i, $\sigma_{\uparrow}$, σ' that verify the hypotheses of Def. 36, then*
$\forall t \in T_i, id_t \in Comps(\Delta)$ *s.t.* $\gamma(t) = id_t$, $s'.reset_t(t) = \sigma'(id_t)("s\_reinit\_time\_counter")$

Proof. Given a $t \in T_i$ and an $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, let us show
$\boxed{s'.reset_t(t) = \sigma'(id_t)("s\_reinit\_time\_counter").}$
By construction and by definition of id_t, there exist gm_t, ipm_t, opm_t s.t. $\texttt{comp}(id_t, "transition", gm_t, ipm_t, opm_t) \in d.cs$.
By property of the $\mathcal{H}$-VHDL stabilize relation, $\texttt{comp}(id_t, "transition", gm_t, ipm_t, opm_t) \in d.cs$, and through the examination of the `reinit_time_counter_evaluation` process defined in the transition design architecture, we can deduce:

$$\sigma'(id_t)("srtc") = \sum_{i=0}^{\Delta(id_t)("input\_arcs\_number")-1} \sigma'(id_t)("reinit\_time")[i] \tag{C.9}$$

Rewriting the goal with (C.9), $\boxed{s'.reset_t(t) = \sum_{i=0}^{\Delta(id_t)("ian")-1} \sigma'(id_t)("rt")[i].}$

Let us perform case analysis on $input(t)$; there are two cases:

- **CASE** $input(t) = \emptyset$:

  By construction, $<\texttt{input\_arcs\_number} \Rightarrow 1> \in gm_t$, and by property of the elaboration relation, we can deduce $\Delta(id_t)("ian") = 1$.

By construction, there exists an $id_{ft} \in Sigs(\Delta)$ s.t. $<\texttt{reinit\_time}(0) \Rightarrow \texttt{id}_{\texttt{ft}}> \in ipm_t$ and $<\texttt{fired} \Rightarrow \texttt{id}_{\texttt{ft}}> \in opm_t$, and by property of the stabilize relation and $\texttt{comp}(id_t, "transition", gm_t, ipm_t, opm_t) \in d.cs$, we can deduce $\sigma'(id_t)("rt")[0] = \sigma'(id_{ft}) = \sigma'(id_t)("fired")$.

Rewriting the goal with $\Delta(id_t)("ian") = 1$ and $\sigma'(id_t)("rt")[0] = \sigma'(id_{ft}) = \sigma'(id_t)("fired")$:

$$\boxed{s'.reset_t(t) = \sigma'(id_t)("fired").}$$

By property of the stabilize relation, $\texttt{comp}(id_t, "transition", gm_t, ipm_t, opm_t) \in d.cs$, and through the examination of the `fired_evaluation` process, we can deduce:

$$\sigma'(id_t)("fired") = \sigma'(id_t)("s\_firable") \,.\, \sigma'(id_t)("s\_priority\_combination") \tag{C.10}$$

Rewriting the goal with (C.10):

$$\boxed{s'.reset_t(t) = \sigma'(id_t)("s\_firable") \,.\, \sigma'(id_t)("s\_priority\_combination").}$$

By property of the stabilize relation, $\texttt{comp}(id_t, "transition", gm_t, ipm_t, opm_t) \in d.cs$, and through the examination of the `priority_authorization_evaluation` process defined in the transition design architecture, we can deduce:

$$\sigma'(id_t)("spc") = \prod_{i=0}^{\Delta(id_t)("ian")-1} \sigma'(id_t)("priority\_authorizations")[i] \tag{C.11}$$

As $\Delta(id_t)("ian") = 1$, we can deduce $\prod_{i=0}^{\Delta(id_t)("ian")-1} \sigma'(id_t)("pauths")[i] = \sigma'(id_t)("pauths")[0]$.

Rewriting the goal with (C.11) and $\prod_{i=0}^{\Delta(id_t)("ian")-1} \sigma'(id_t)("pauths")[i] = \sigma'(id_t)("pauths")[0]$:

$$\boxed{s'.reset_t(t) = \sigma'(id_t)("s\_firable") \,.\, \sigma'(id_t)("pauths")[0].}$$

By construction, $<\texttt{priority\_authorizations}(0) \Rightarrow \texttt{true}> \in ipm_t$, and by property of the stabilize relation and $\texttt{comp}(id_t, "transition", gm_t, ipm_t, opm_t) \in d.cs$, we can deduce $\sigma'(id_t)("pauths")[0] = \texttt{true}$.

Rewriting the goal with $\sigma'(id_t)("pauths")[0] = \texttt{true}$, and simplifying the equation:

$$\boxed{s'.reset_t(t) = \sigma'(id_t)("s\_firable").}$$

Let us perform case analysis on $t \in Fired(s)$ or $t \notin Fired(s)$:

– **CASE** $t \in Fired(s)$:

By property of $E_c, \tau \vdash s \xrightarrow{\uparrow} s'$ (Rule (8)), we can deduce $s'.reset_t(t) = \texttt{true}$.

Rewriting the goal with $s'.reset_t(t) = \texttt{true}$: $\boxed{\sigma'(id_t)("s\_firable") = \texttt{true}.}$

By property of the stabilize, the $\mathcal{H}$-VHDL rising edge and the $\texttt{Inject}_{\uparrow}$ relations, $\texttt{comp}(id_t, "transition", gm_t, ipm_t, opm_t) \in d.cs$, and through the examination of the `firable` process defined in the transition design architecture, we can deduce $\sigma(id_t)("s\_firable") = \sigma'(id_t)("s\_firable")$.

Rewriting the goal with $\sigma(id_t)(\text{"}s\_firable\text{"}) = \sigma'(id_t)(\text{"}s\_firable\text{"})$, $\boxed{\sigma(id_t)(\text{"}s\_firable\text{"}) = \texttt{true}.}$

By property of $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$, we can deduce $t \in Firable(s) \Leftrightarrow \sigma(id_t)(\text{"}sfa\text{"}) = \texttt{true}$.

Rewriting the goal with $t \in Firable(s) \Leftrightarrow \sigma(id_t)(\text{"}sfa\text{"}) = \texttt{true}$, $\boxed{t \in Firable(s).}$

By property of $t \in Fired(s)$, $t \in Firable(s)$.

– **CASE** $t \notin Fired(s)$:

By property of $input(t) = \emptyset$, there does not exist any input place connected to t by a `basic` or `test` arc. Thus, by property of $E_c, \tau \vdash s \xrightarrow{\uparrow} s'$ (Rule (8)), we can deduce $s'.reset_t(t) = \texttt{false}$.

Rewriting the goal with $s'.reset_t(t) = \texttt{false}$: $\boxed{\sigma'(id_t)(\text{"}s\_firable\text{"}) = \texttt{false}.}$

By property of the stabilize, the $\mathcal{H}$-VHDL rising edge and the $\texttt{Inject}_{\uparrow}$ relations, $\texttt{comp}(id_t, \text{"}transition\text{"}, gm_t, ipm_t, opm_t) \in d.cs$, and through the examination of the `firable` process defined in the transition design architecture, we can deduce $\sigma(id_t)(\text{"}sfa\text{"}) = \sigma'(id_t)(\text{"}sfa\text{"})$.

Rewriting the goal with $\sigma(id_t)(\text{"}sfa\text{"}) = \sigma'(id_t)(\text{"}sfa\text{"})$, $\boxed{\sigma(id_t)(\text{"}sfa\text{"}) = \texttt{false}.}$

By property of $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$, we can deduce $t \notin Firable(s) \Leftrightarrow \sigma(id_t)(\text{"}sfa\text{"}) = \texttt{false}$.

By property of $t \notin Fired(s)$ and $input(t) = \emptyset$, $t \notin Firable(s)$.

- **CASE** $input(t) \neq \emptyset$:

By construction, $<\texttt{input\_arcs\_number} \Rightarrow |input(t)|> \in gm_t$, and by property of the elaboration relation, we can deduce $\Delta(id_t)(\text{"}ian\text{"}) = |input(t)|$.

Rewriting the goal with $\Delta(id_t)(\text{"}ian\text{"}) = |input(t)|$, $\boxed{s'.reset_t(t) = \sum_{i=0}^{|input(t)|-1} \sigma'(id_t)(\text{"}rt\text{"})[i].}$

Let us perform case analysis on $t \in Fired(s)$ or $t \notin Fired(s)$:

– **CASE** $t \in Fired(s)$:

By property of $E_c, \tau \vdash s \xrightarrow{\uparrow} s'$ (Rule (8)), we can deduce $s'.reset_t(t) = \texttt{true}$.

Rewriting the goal with $s'.reset_t(t) = \texttt{true}$, $\boxed{\sum_{i=0}^{|input(t)|-1} \sigma'(id_t)(\text{"}rt\text{"})[i] = \texttt{true}.}$

To prove the goal, let us show $\boxed{\exists i \in [0, |input(t)| - 1] \text{ s.t. } \sigma'(id_t)(\text{"}rt\text{"})[i] = \texttt{true}.}$

By construction, and $input(t) \neq \emptyset$, there exist $p \in input(t)$ and $id_p \in Comps(\Delta)$ s.t. $\gamma(p) = id_p$.

By construction and by definition of id_p, there exist gm_p, ipm_p, opm_p s.t. $\texttt{comp}(id_p, \text{"}place\text{"}, gm_p, ipm_p, opm_p) \in d.cs$.

By construction, there exist an $i \in [0, |input(t)| - 1]$, a $j \in [0, |output(p)| - 1]$ and $id_{ji} \in Sigs(\Delta)$ s.t. $<\texttt{reinit\_transition\_time(j)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in opm_p$ and

$<\texttt{reinit\_time(i)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in ipm_t$. Let us take such an i, j and id_{ji}, and let us use i to prove the goal: $\boxed{\sigma'(id_t)(''rt'')[i] = \texttt{true}.}$

By property of the stabilize relation, $<\texttt{reinit\_transition\_time(j)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in opm_p$ and $<\texttt{reinit\_time(i)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in ipm_t$, we can deduce $\sigma'(id_t)(''rt'')[i] = \sigma'(id_{ji}) = \sigma'(id_p)(''rtt'')[j]$.

Rewriting the goal with $\sigma'(id_t)(''rt'')[i] = \sigma'(id_{ji}) = \sigma'(id_p)(''rtt'')[j]$, $\boxed{\sigma'(id_p)(''rtt'')[j] = \texttt{true}.}$

By property of the $\texttt{Inject}_{\uparrow}$, the $\mathcal{H}$-VHDL rising edge and the stabilize relations, $\texttt{comp}(id_p, ''place'', gm_p, ipm_p, opm_p) \in d.cs$, and through the examination of the `reinit_transitions_time_evaluation` process defined in the place design architecture, we can deduce:

$$\begin{aligned}\sigma'(id_p)(''rtt'')[j] = &\big((\sigma(id_p)(''oat'')[j] = \texttt{basic} + \sigma(id_p)(''oat'')[j] = \texttt{test})\\ &.(\sigma(id_p)(''sm'') - \sigma(id_p)(''sots'') < \sigma(id_p)(''oaw'')[j])\\ &.(\sigma(id_p)(''sots'') > 0)\big)\\ &+ \sigma(id_p)(''otf'')[j]\end{aligned} \tag{C.12}$$

Rewriting the goal with (C.12),

$$\boxed{\begin{aligned}\texttt{true} = &((\sigma(id_p)(''oat'')[j] = \texttt{basic} + \sigma(id_p)(''oat'')[j] = \texttt{test})\\ &.(\sigma(id_p)(''sm'') - \sigma(id_p)(''sots'') < \sigma(id_p)(''oaw'')[j])\\ &.(\sigma(id_p)(''sots'') > 0))\\ &+ (\sigma(id_p)(''otf'')[j])\end{aligned}}$$

By construction, there exists $id_{ft} \in Sigs(\Delta)$ s.t. $<\texttt{output\_transitions\_fired(j)} \Rightarrow \texttt{id}_{\texttt{ft}}> \in ipm_p$ and $<\texttt{fired} \Rightarrow \texttt{id}_{\texttt{ft}}> \in opm_t$. By property of state σ as being a stable state, we can deduce $\sigma(id_t)(''fired'') = \sigma(id_{ft}) = \sigma(id_p)(''otf'')[j]$.

Rewriting the goal with $\sigma(id_t)(''fired'') = \sigma(id_{ft}) = \sigma(id_p)(''otf'')[j]$,

$$\boxed{\begin{aligned}\texttt{true} = &((\sigma(id_p)(''oat'')[j] = \texttt{basic} + \sigma(id_p)(''oat'')[j] = \texttt{test})\\ &.(\sigma(id_p)(''sm'') - \sigma(id_p)(''sots'') < \sigma(id_p)(''oaw'')[j])\\ &.(\sigma(id_p)(''sots'') > 0))\\ &+ \sigma(id_t)(''fired'')\end{aligned}}$$

By property of $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$, we can deduce $t \in Fired(s) \Leftrightarrow \sigma(id_t)(''fired'') = \texttt{true}$.

Rewriting the goal with $t \in Fired(s) \Leftrightarrow \sigma(id_t)(''fired'') = \texttt{true}$ and simplify the goal, then tautology.

– **CASE** $t \notin Fired(s)$: Then, there are two cases that will determine the value of $s'.reset_t(t)$. Either there exists a place p with an output token sum greater than zero, that is connected to t by an `basic` or `test` arc, and such that the transient marking of p disables t; or such a place does not exist (the predicate is decidable).

∗ **CASE** there exists such a place p as described above:
Then, let us take such a place p and $\omega \in \mathbb{N}^*$ s.t.:

1. $\sum_{t_i \in Fired(s)} pre(p, t_i) > 0$
2. $pre(p,t) = (\omega, \texttt{basic}) \vee pre(p,t) = (\omega, \texttt{test})$
3. $s.M(p) - \sum_{t_i \in Fired(s)} pre(p, t_i) < \omega$

We will only consider the case where $pre(p,t) = (\omega, \texttt{basic})$; the proof is the similar when $pre(p,t) = (\omega, \texttt{test})$.

Assuming that p exists, and by property of $E_c, \tau \vdash s \xrightarrow{\uparrow} s'$ (Rule (8)), we can deduce $s'.reset_t(t) = \texttt{true}$.

Rewriting the goal with $s'.reset_t(t) = \texttt{true}$, $\boxed{\sum_{i=0}^{|input(t)|-1} \sigma'(id_t)(''rt'')[i] = \texttt{true}.}$

To prove the goal, let us show $\boxed{\exists i \in [0, |input(t)| - 1] \text{ s.t. } \sigma'(id_t)(''rt'')[i] = \texttt{true}.}$
By construction, there exists $id_p \in Comps(\Delta)$ s.t. $\gamma(p) = id_p$.
By construction and by definition of id_p, there exist gm_p, ipm_p, opm_p s.t. $\texttt{comp}(id_p, ''place'', gm_p, ipm_p, opm_p) \in d.cs$.
By construction, there exist an $i \in [0, |input(t)| - 1]$, a $j \in [0, |output(p)| - 1]$ and $id_{ji} \in Sigs(\Delta)$ s.t. $<\texttt{reinit\_transition\_time(j)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in opm_p$ and $<\texttt{reinit\_time(i)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in ipm_t$. Let us take such an i, j and id_{ji}, and let us use i to prove the goal: $\boxed{\sigma'(id_t)(''rt'')[i] = \texttt{true}.}$
By property of the stabilize relation, $<\texttt{reinit\_transition\_time(j)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in opm_p$ and $<\texttt{reinit\_time(i)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in ipm_t$, we can deduce $\sigma'(id_t)(''rt'')[i] = \sigma'(id_{ji}) = \sigma'(id_p)(''rtt'')[j]$.
Rewriting the goal with $\sigma'(id_t)(''rt'')[i] = \sigma'(id_{ji}) = \sigma'(id_p)(''rtt'')[j]$, $\boxed{\sigma'(id_p)(''rtt'')[j] = \texttt{true}.}$
By property of the $\texttt{Inject}_{\uparrow}$, the $\mathcal{H}$-VHDL rising edge and the stabilize relation, and through the examination of the `reinit_transitions_time_evaluation` process defined in the place design architecture, we can deduce:

$$\begin{aligned}\sigma'(id_p)(''rtt'')[j] = &((\sigma(id_p)(''oat'')[j] = \texttt{basic} + \sigma(id_p)(''oat'')[j] = \texttt{test})\\ &.(\sigma(id_p)(''sm'') - \sigma(id_p)(''sots'') < \sigma(id_p)(''oaw'')[j])\\ &.(\sigma(id_p)(''sots'') > 0))\\ &+ \sigma(id_p)(''otf'')[j]\end{aligned} \tag{C.13}$$

Rewriting the goal with (C.13),

$$\boxed{\begin{aligned}\texttt{true} = &((\sigma(id_p)(''oat'')[j] = \texttt{basic} + \sigma(id_p)(''oat'')[j] = \texttt{test})\\ &.(\sigma(id_p)(''sm'') - \sigma(id_p)(''sots'') < \sigma(id_p)(''oaw'')[j])\\ &.(\sigma(id_p)(''sots'') > 0))\\ &+ \sigma(id_p)(''otf'')[j]\end{aligned}}$$

By construction, $<\texttt{output\_arcs\_types(j)} \Rightarrow \texttt{basic}> \in ipm_p$ and $<\texttt{output\_arcs\_weights(j)} \Rightarrow \omega> \in ipm_p$.
By property of the stabilize relation and $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$, we can deduce $\sigma'(id_p)("oat")[j] = \texttt{basic}$ and $\sigma'(id_p)("oaw")[j] = \omega$.

By property of $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$, we can deduce $\sigma(id_p)("sm") = s.M(p)$ and $\sigma(id_p)("sots") = \sum_{t_i \in Fired(s)} pre(p, t_i)$.
Rewriting the goal with $\sigma'(id_p)("oat")[j] = \texttt{basic}$, $\sigma'(id_p)("oaw")[j] = \omega$, $\sigma(id_p)("sm") = s.M(p)$ and $\sigma(id_p)("sots") = \sum_{t_i \in Fired(s)} pre(p, t_i)$, and simplifying the goal:

$$\boxed{\left((s.M(p) - \sum_{t_i \in Fired(s)} pre(p, t_i) < \omega) \,.\, (\sum_{t_i \in Fired(s)} pre(p, t_i) > 0)\right) + \sigma(id_t)("fired") = \texttt{true}}$$

We assumed that $s.M(p) - \sum_{t_i \in Fired(s)} pre(p, t_i) < \omega$ and $\sum_{t_i \in Fired(s)} pre(p, t_i) > 0$. Thus, by assumption:

$$\left((s.M(p) - \sum_{t_i \in Fired(s)} pre(p, t_i) < \omega) \,.\, (\sum_{t_i \in Fired(s)} pre(p, t_i) > 0)\right) + \sigma(id_t)("fired") = \texttt{true}$$

* **CASE** such a place does not exist:
Then, let us assume that, for all place $p \in P$

1. $\sum_{t_i \in Fired(s)} pre(p, t_i) = 0$
2. or $\forall \omega \in \mathbb{N}^*,\ pre(p, t) = (\omega, \texttt{basic}) \vee pre(p, t) = (\omega, \texttt{test}) \Rightarrow s.M(p) - \sum_{t_i \in Fired(s)} pre(p, t_i) \geq \omega$.

In that case, by property of $E_c, \tau \vdash s \xrightarrow{\uparrow} s'$ (Rule (8)), we can deduce $s'.reset_t(t) = \texttt{false}$.
Rewriting the goal with $s'.reset_t(t) = \texttt{false}$: $\boxed{\sum_{i=0}^{|input(t)|-1} \sigma'(id_t)("rt")[i] = \texttt{false}.}$
To prove the goal, let us show $\boxed{\forall i \in [0, |input(t)| - 1],\ \sigma'(id_t)("rt")[i] = \texttt{false}.}$
Given an $i \in [0, |input(t)| - 1]$, let us show $\boxed{\sigma'(id_t)("rt")[i] = \texttt{false}.}$
By construction, there exist a $p \in input(t)$, an $id_p \in Comps(\Delta)$, gm_p, ipm_p, opm_p, a $j \in [0, |output(p)| - 1]$, an $id_{ji} \in Sigs(\Delta)$ s.t. $\gamma(p) = id_p$ and $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$ and $<\texttt{reinit\_transition\_time(j)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in opm_p$ and $<\texttt{reinit\_time(i)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in ipm_t$. Let us take such a p, id_p, gm_p, ipm_p, opm_p, j and id_{ji}.
By property of the stabilize relation, $<\texttt{reinit\_transition\_time(j)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in opm_p$ and $<\texttt{reinit\_time(i)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in ipm_t$, we can deduce $\sigma'(id_t)("rt")[i] = \sigma'(id_{ji}) = \sigma'(id_p)("rtt")[j]$.
Rewriting the goal with $\sigma'(id_t)("rt")[i] = \sigma'(id_{ji}) = \sigma'(id_p)("rtt")[j]$: $\boxed{\sigma'(id_p)("rtt")[j] = \texttt{false}.}$
By property of the $\texttt{Inject}_{\uparrow}$, the $\mathcal{H}$-VHDL rising edge and the stabilize relations, $\texttt{comp}(id_p,$

$"place", gm_p, ipm_p, opm_p) \in d.cs$, and through the examination of the `reinit_transitions_time_evaluation` process defined in the place design architecture, we can deduce:

$$\begin{aligned}\sigma'(id_p)("rtt")[j] = &\big((\sigma(id_p)("oat")[j] = \texttt{basic} + \sigma(id_p)("oat")[j] = \texttt{test})\\ &.(\sigma(id_p)("sm") - \sigma(id_p)("sots") < \sigma(id_p)("oaw")[j])\\ &.(\sigma(id_p)("sots") > 0)\big)\\ &+ \sigma(id_p)("otf")[j]\end{aligned} \tag{C.14}$$

Rewriting the goal with (C.14),

$$\boxed{\begin{aligned}\texttt{false} = &((\sigma(id_p)("oat")[j] = \texttt{basic} + \sigma(id_p)("oat")[j] = \texttt{test})\\ &.(\sigma(id_p)("sm") - \sigma(id_p)("sots") < \sigma(id_p)("oaw")[j])\\ &.(\sigma(id_p)("sots") > 0))\\ &+ \sigma(id_p)("otf")[j])\end{aligned}}$$

By construction, there exists $id_{ft} \in Sigs(\Delta)$ s.t. $<$`output_transitions_fired(j)` $\Rightarrow$ `id`$_{\texttt{ft}}>$ $\in$ ipm_p and $<$`fired` $\Rightarrow$ `id`$_{\texttt{ft}}>$ $\in$ opm_t. By property of state σ as being a stable state, we can deduce $\sigma(id_t)("fired") = \sigma(id_{ft}) = \sigma(id_p)("otf")[j]$.
Rewriting the goal with $\sigma(id_t)("fired") = \sigma(id_{ft}) = \sigma(id_p)("otf")[j]$:

$$\boxed{\begin{aligned}\texttt{false} = &((\sigma(id_p)("oat")[j] = \texttt{basic} + \sigma(id_p)("oat")[j] = \texttt{test})\\ &.(\sigma(id_p)("sm") - \sigma(id_p)("sots") < \sigma(id_p)("oaw")[j])\\ &.(\sigma(id_p)("sots") > 0))\\ &+ \sigma(id_t)("fired")\end{aligned}}$$

By property of $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$, we can deduce $t \notin Fired(s) \Leftrightarrow \sigma(id_t)("fired") = \texttt{false}$
Rewriting the goal with $t \notin Fired(s) \Leftrightarrow \sigma(id_t)("fired") = \texttt{false}$ and simplifying the goal:

$$\boxed{\begin{aligned}\texttt{false} = &((\sigma(id_p)("oat")[j] = \texttt{basic} + \sigma(id_p)("oat")[j] = \texttt{test})\\ &.(\sigma(id_p)("sm") - \sigma(id_p)("sots") < \sigma(id_p)("oaw")[j])\\ &.(\sigma(id_p)("sots") > 0))\end{aligned}}$$

Then, based on the assumptions made at the beginning of case, there are two cases:

1. **CASE** $\sum_{t_i \in Fired(s)} pre(p, t_i) = 0$:

   By property of $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$, we can deduce $\sum_{t_i \in Fired(s)} pre(p, t_i) = \sigma(id_p)("sots")$.

   Rewriting the goal with $\sum_{t_i \in Fired(s)} pre(p, t_i) = \sigma(id_p)("sots")$ and $\sum_{t_i \in Fired(s)} pre(p, t_i) = 0$, and simplifying the goal: tautology.

2. **CASE** $\forall \omega \in \mathbb{N}^*,\ pre(p,t) = (\omega, \texttt{basic}) \vee pre(p,t) = (\omega, \texttt{test}) \Rightarrow s.M(p) - \sum_{t_i \in Fired(s)} pre(p,t_i) \geq \omega$:

   Let us perform case analysis on $pre(p,t)$; there are two cases:

   (a) **CASE** $pre(p,t) = (\omega, \texttt{basic})$ or $pre(p,t) = (\omega, \texttt{basic})$:
   By construction, $<\texttt{output\_arcs\_weights(j)} \Rightarrow \omega> \in ipm_p$.
   By property of stable state σ and $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$, we can deduce $\sigma(id_p)("oaw")[j] = \omega$.
   By property of $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$, we can deduce $\sigma(id_p)("sm") = s.M(p)$ and $\sigma(id_p)("sots") = \sum_{t_i \in Fired(s)} pre(p,t_i)$.
   Rewriting the goal with $\sigma(id_p)("oaw")[j] = \omega$, $\sigma(id_p)("sm") = s.M(p)$ and $\sigma(id_p)("sots") = \sum_{t_i \in Fired(s)} pre(p,t_i)$:

   $$\boxed{\begin{aligned}\texttt{false} = &((\sigma(id_p)("oat")[j] = \texttt{basic} + \sigma(id_p)("oat")[j] = \texttt{test})\\ &.(s.M(p) - \sum_{t_i \in Fired(s)} pre(p,t_i) < \omega)\\ &.(\sum_{t_i \in Fired(s)} pre(p,t_i) > 0))\end{aligned}}$$

   We assumed that $s.M(p) - \sum_{t_i \in Fired(s)} pre(p,t_i) \geq \omega$, and then we can deduce $s.M(p) - \sum_{t_i \in Fired(s)} pre(p,t_i) < \omega = \texttt{false}$.
   Rewriting the goal with $s.M(p) - \sum_{t_i \in Fired(s)} pre(p,t_i) < \omega = \texttt{false}$, and simplifying the goal, tautology.

   (b) **CASE** $pre(p,t) = (\omega, \texttt{inhib})$:
   By construction, $<\texttt{output\_arcs\_types(j)} \Rightarrow \texttt{inhib}> \in ipm_p$.
   By property of stable state σ and $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$, we can deduce $\sigma(id_p)("oat")[j] = \texttt{inhib}$.
   Rewriting the goal with $\sigma(id_p)("oat")[j] = \texttt{inhib}$, and simplifying the goal, tautology.

□

C.3.5 Rising edge and action executions

Lemma 28 (Rising edge equal action executions). *For all $sitpn$, d, γ, E_c, E_p, τ, Δ, σ_e, s, s', σ, σ_i, $\sigma_\uparrow$, σ' that verify the hypotheses of Def. 36, then*
$\forall a \in \mathcal{A}, id_a \in Outs(\Delta)\ s.t.\ \gamma(a) = id_a,\ s'.ex(a) = \sigma'(id_a)$.

Proof. Given an $a \in \mathcal{A}$ and an $id_a \in Outs(\Delta)\ s.t.\ \gamma(a) = id_a$, let us show $\boxed{s'.ex(a) = \sigma'(id_a).}$

By property of $E_c, \tau \vdash s \xrightarrow{\uparrow} s'$, we can deduce $s.ex(a) = s'.ex(a)$.
By construction, id_a is an output port identifier of Boolean type in the $\mathcal{H}$-VHDL design d. The generated ''`action`'' process is responsible for the assignment of the *id* a only during the initialization phase or during a falling edge phase.
By property of the $\mathcal{H}$-VHDL $\texttt{Inject}_{\uparrow}$, rising edge, stabilize relations, and the ''`action`'' process, we can deduce $\sigma(id_a) = \sigma'(id_a)$.
Rewriting the goal with $s.ex(a) = s'.ex(a)$ and $\sigma(id_a) = \sigma'(id_a)$, $\boxed{s.ex(a) = \sigma(id_a).}$

By property of $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$, $s.ex(a) = \sigma(id_a).$ □

C.3.6 Rising edge and function executions

Lemma 29 (Rising edge equal function executions). *For all sitpn, d, γ, E_c, E_p, τ, Δ, σ_e, s, s', σ, σ_i, $\sigma_{\uparrow}$, σ' that verify the hypotheses of Def. 36, then*
$\forall f \in \mathcal{F}, id_f \in Outs(\Delta)$ *s.t.* $\gamma(f) = id_f$, $s'.ex(f) = \sigma'(id_f)$.

Proof. Given an $f \in \mathcal{F}$ and an $id_f \in Outs(\Delta)$ s.t. $\gamma(f) = id_f$, let us show $\boxed{s'.ex(f) = \sigma'(id_f).}$

By property of $E_c, \tau \vdash s \xrightarrow{\uparrow} s'$ (Rule (9)):

$$s'.ex(f) = \sum_{t \in Fired(s)} \mathbb{F}(t, f) \tag{C.15}$$

By construction, id_f is an output port identifier of Boolean type in the $\mathcal{H}$-VHDL design d. The generated `function` process assigns a value to the output port id_f only during the initialization phase or during a rising edge phase.
By construction, the `function` process is defined in the behavior of design d, i.e.
$\texttt{ps}(''function'', \emptyset, sl, ss) \in d.cs$.
Let $trs(f)$ be the set of transitions associated to function f, i.e $trs(f) = \{t \in T \mid \mathbb{F}(t, f) = \texttt{true}\}$.
Let us perform case analysis on $trs(f)$; there are two cases:

- **CASE** $trs(f) = \emptyset$:

  By construction, $\texttt{id}_\texttt{f} \Leftarrow \texttt{false} \in ss_{\uparrow}$ where $ss_{\uparrow}$ is the part of the `function` process body executed during a rising edge phase.

  By property of the $\mathcal{H}$-VHDL rising edge, the stabilize relations and $\texttt{ps}(''function'', \emptyset, sl, ss) \in d.cs$, we can deduce $\sigma'(id_f) = \texttt{false}$.

  By property of $\sum_{t \in Fired(s)} \mathbb{F}(t, f)$ and $trs(f) = \emptyset$, we can deduce $\sum_{t \in Fired(s)} \mathbb{F}(t, f) = \texttt{false}$.

  Rewriting the goal with (C.15), $\sigma'(id_f) = \texttt{false}$ and $\sum_{t \in Fired(s)} \mathbb{F}(t, f) = \texttt{false}$: tautology.

- **CASE** $trs(f) \neq \emptyset$:

  By construction, $\texttt{id}_\texttt{f} \Leftarrow \texttt{id}_{\texttt{ft}_0} + \cdots + \texttt{id}_{\texttt{ft}_n} \in ss_{\uparrow}$, where $id_{ft_i} \in Sigs(\Delta)$, $ss_{\uparrow}$ is the part of the `function` process body executed during a rising edge phase, and $n = |trs(f)| - 1$.

By property of the $\texttt{Inject}_{\uparrow}$, the $\mathcal{H}$-VHDL rising edge, the stabilize relations, and $\texttt{ps}(\text{"}function\text{"}, \emptyset, sl, ss) \in d.cs$, we can deduce:

$$\sigma'(id_f) = \sigma(id_{ft_0}) + \cdots + \sigma(id_{ft_n}) \tag{C.16}$$

Rewriting the goal with (C.15) and (C.16), $\boxed{\sum_{t \in Fired(s)} \mathbb{F}(t,f) = \sigma(id_{ft_0}) + \cdots + \sigma(id_{ft_n}).}$

Let us reason on the value of $\sigma(id_{ft_0}) + \cdots + \sigma(id_{ft_n})$; there are two cases:

– **CASE** $\sigma(id_{ft_0}) + \cdots + \sigma(id_{ft_n}) = \texttt{true}$:

Then, we can rewrite the goal as follows: $\boxed{\sum_{t \in Fired(s)} \mathbb{F}(t,f) = \texttt{true}.}$

To prove the above goal, let us show $\boxed{\exists t \in Fired(s)\ s.t.\ \mathbb{F}(t,f) = \texttt{true}.}$

From $\sigma(id_{ft_0}) + \cdots + \sigma(id_{ft_n}) = \texttt{true}$, we can deduce $\exists id_{ft_i}\ s.t.\ \sigma(id_{ft_i}) = \texttt{true}$. Let us take such an id_{ft_i}.

By construction, there exist a $t \in trs(f)$, an $id_t \in Comps(\Delta)$, gm_t, ipm_t, opm_t such that:

* $\gamma(t) = id_t$
* $\texttt{comp}(id_t, \text{"}transition\text{"}, gm_t, ipm_t, opm_t) \in d.cs$
* $<\texttt{fired} \Rightarrow \texttt{id}_{\texttt{ft}_\texttt{i}}> \in opm_t$

By property of σ as being a stable design state, and $\texttt{comp}(id_t, \text{"}transition\text{"}, gm_t, ipm_t, opm_t) \in d.cs$, we can deduce $\sigma(id_t)(\text{"}fired\text{"}) = \sigma(id_{ft_i})$, and thus that $\sigma(id_t)(\text{"}fired\text{"}) = \texttt{true}$.

By property of $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$, we can deduce $t \in Fired(s)$.

Let us use t to prove the goal: $\boxed{\mathbb{F}(t,f) = \texttt{true}.}$

By definition of $t \in trs(f)$, $\mathbb{F}(t,f) = \texttt{true}.$

– **CASE** $\sigma(id_{ft_0}) + \cdots + \sigma(id_{ft_n}) = \texttt{false}$:

Then, we can rewrite the goal as follows: $\boxed{\sum_{t \in Fired(s)} \mathbb{F}(t,f) = \texttt{false}.}$

To prove the above goal, let us show $\boxed{\forall t \in Fired(s)\ s.t.\ \mathbb{F}(t,f) = \texttt{false}.}$

Given a $t \in Fired(s)$, let us show $\boxed{\mathbb{F}(t,f) = \texttt{false}.}$

Let us perform case analysis on $\mathbb{F}(t,f)$; there are 2 cases:

* **CASE** $\mathbb{F}(t,f) = \texttt{false}.$
* **CASE** $\mathbb{F}(t,f) = \texttt{true}$:

  By construction, there exist an $id_t \in Comps(\Delta)$, gm_t, ipm_t, opm_t and $id_{ft_i} \in Sigs(\Delta)$ such that:

  · $\gamma(t) = id_t$

· $\texttt{comp}(id_t, "transition", gm_t, ipm_t, opm_t) \in d.cs$
· $<\texttt{fired} \Rightarrow \texttt{id}_{\texttt{ft}_i}> \in opm_t$

By property of stable design state σ and $\texttt{comp}(id_t, "transition", gm_t, ipm_t, opm_t) \in d.cs$, we can deduce $\sigma(id_t)("fired") = \sigma(id_{ft_i})$.

By property of $\gamma \vdash s \overset{\downarrow}{\approx} \sigma$, we can deduce $t \in Fired(s) \Leftrightarrow \sigma(id_t)("fired") = \texttt{true}$.
Since $t \in Fired(s)$, we can deduce $\sigma(id_t)("fired") = \texttt{true}$, and from $\sigma(id_t)("fired") = \sigma(id_{ft_i})$, we can deduce $\sigma(id_{ft_i}) = \texttt{true}$.
Then, $\sigma(id_{ft_i}) = \texttt{true}$ contradicts $\sigma(id_{ft_0}) + \cdots + \sigma(id_{ft_n}) = \texttt{false}$.

□

C.3.7 Rising edge and sensitization

Lemma 30 (Rising edge equal sensitized). *For all $sitpn$, d, γ, E_c, E_p, τ, Δ, σ_e, s, s', σ, σ_i, $\sigma_\uparrow$, σ' that verify the hypotheses of Def. 36, then*
$\forall t \in T, id_t \in Comps(\Delta)$ *s.t.* $\gamma(t) = id_t, t \in Sens(s'.M) \Leftrightarrow \sigma'(id_t)("s\_enabled") = \texttt{true}$.

Proof. Given a $t \in T$ and an $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, let us show
$t \in Sens(s'.M) \Leftrightarrow \sigma'(id_t)("s\_enabled") = \texttt{true}$.

By construction and by definition of id_t, there exist gm_t, ipm_t, opm_t s.t. $\texttt{comp}(id_t, "transition", gm_t, ipm_t, opm_t) \in d.cs$.. Then, the proof is in two parts:

1. Assuming that $t \in Sens(s'.M)$, let us show $\sigma'(id_t)("s\_enabled") = \texttt{true}$.

   By property of the stabilize relation, $\texttt{comp}(id_t, "transition", gm_t, ipm_t, opm_t) \in d.cs$, and through the examination of the `enable_evaluation` process defined in the transition design architecture:

   $$\sigma'(id_t)("se") = \prod_{i=0}^{\Delta(id_t)("ian")-1} \sigma'(id_t)("input\_arcs\_valid")[i] \tag{C.17}$$

   Rewriting the goal with (C.17), $\prod_{i=0}^{\Delta(id_t)("ian")-1} \sigma'(id_t)("iav")[i] = \texttt{true}$.

   To prove the goal, let us show that $\forall i \in [0, \Delta(id_t)("ian") - 1],\ \sigma'(id_t)("iav")[i] = \texttt{true}$.

   Given an $i \in [0, \Delta(id_t)("ian") - 1]$, let us show $\sigma'(id_t)("iav")[i] = \texttt{true}$.

   Let us perform case analysis on $input(t)$.

   - **CASE** $input(t) = \emptyset$:
     By construction, $<\texttt{input\_arcs\_number} \Rightarrow 1> \in gm_t$ and
     $<\texttt{input\_arcs\_valid}(0) \Rightarrow \texttt{true}> \in ipm_t$.

By property of the elaboration and stabilize relations and $\texttt{comp}(id_t, "transition", gm_t, ipm_t, opm_t) \in d.cs$, we can deduce $\Delta(id_t)("ian") = 1$ and $\sigma'(id_t)("iav")[0] = \texttt{true}$.
Thanks to $\Delta(id_t)("ian") = 1$, we can deduce that $i = 0$.
Rewriting the goal with $\sigma'(id_t)("iav")[0] = \texttt{true}$, tautology.

- **CASE** $input(t) \neq \emptyset$:
  By construction, $<\texttt{input\_arcs\_number} \Rightarrow |input(t)|> \in gm_t$.
  By property of the elaboration relation and $\texttt{comp}(id_t, "transition", gm_t, ipm_t, opm_t) \in d.cs$, we can deduce $\Delta(id_t)("ian") = |input(t)|$.
  Thanks to $\Delta(id_t)("ian") = |input(t)|$, we know that $i \in [0, |input(t)| - 1]$.
  By construction, there exist a $p \in input(t)$, $id_p \in Comps(\Delta)$, gm_p, ipm_p, opm_p, $j \in [0, |output(p)| - 1]$ and $id_{ji} \in Sigs(\Delta)$ s.t. $\gamma(p) = id_p$ and
  $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$ and $<\texttt{output\_arcs\_valid(j)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in opm_p$ and $<\texttt{input\_arcs\_valid(i)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in ipm_t$.
  By property of the stabilize relation, $\texttt{comp}(id_t, "transition", gm_t, ipm_t, opm_t) \in d.cs$ and $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$, we can deduce $\sigma'(id_t)("iav")[i] = \sigma'(id_{ji}) = \sigma'(id_p)("oav")[j]$.
  Rewriting the goal with $\sigma'(id_t)("iav")[i] = \sigma'(id_{ji}) = \sigma'(id_p)("oav")[j]$:
  $\boxed{\sigma'(id_p)("oav")[j] = \texttt{true}.}$
  By property of the stabilize relation, $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$, and through the examination of the `marking_validation_evaluation` process defined in the place design architecture, we can deduce:

$$\begin{aligned}\sigma'(id_p)("oav")[j] = &\big((\sigma'(id_p)("oat")[j] = \texttt{basic} + \sigma'(id_p)("oat")[j] = \texttt{test})\\ &\quad . \sigma'(id_p)("sm") \geq \sigma'(id_p)("oaw")[j]\big)\\ &+ \big(\sigma'(id_p)("oat")[j] = \texttt{inhib} \,.\, \sigma'(id_p)("sm") < \sigma'(id_p)("oaw")[j]\big)\end{aligned} \tag{C.18}$$

  Rewriting the goal with (C.18),

$$\boxed{\begin{aligned}\texttt{true} = &\big((\sigma'(id_p)("oat")[j] = \texttt{basic} + \sigma'(id_p)("oat")[j] = \texttt{test})\\ &\quad . \sigma'(id_p)("sm") \geq \sigma'(id_p)("oaw")[j]\big)\\ &+ \big(\sigma'(id_p)("oat")[j] = \texttt{inhib} \,.\, \sigma'(id_p)("sm") < \sigma'(id_p)("oaw")[j]\big)\end{aligned}}$$

  Let us perform case analysis on $pre(p,t)$; there are 3 cases:

  – **CASE** $pre(p,t) = (\omega, \texttt{basic})$:

    By construction, $<\texttt{output\_arcs\_types(j)} \Rightarrow \texttt{basic}> \in ipm_p$ and
    $<\texttt{output\_arcs\_weights(j)} \Rightarrow \omega> \in ipm_p$.
    By property of the stabilize relation and $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$, we can deduce $\sigma'(id_p)("oat")[j] = \texttt{basic}$ and $\sigma'(id_p)("oaw")[j] = \omega$.

Rewriting the goal with $\sigma'(id_p)(\text{"oat"})[j] = \texttt{basic}$ and $\sigma'(id_p)(\text{"oaw"})[j] = \omega$, and simplifying the goal:
$\boxed{\sigma'(id_p)(\text{"sm"}) \geq \omega = \texttt{true}.}$
Appealing to Lemma 24, we can deduce $s'.M(p) = \sigma'(id_p)(\text{"sm"})$.
Rewriting the goal with $s'.M(p) = \sigma'(id_p)(\text{"sm"})$: $\boxed{s'.M(p) \geq \omega = \texttt{true}.}$
By definition of $t \in Sens(s'.M)$, $s'.M(p) \geq \omega = \texttt{true}.$

- **CASE** $pre(p,t) = (\omega, \texttt{test})$: same as above.
- **CASE** $pre(p,t) = (\omega, \texttt{inhib})$:
  By construction, $<\texttt{output\_arcs\_types(j)} \Rightarrow \texttt{inhib}> \in ipm_p$ and $<\texttt{output\_arcs\_weights(j)} \Rightarrow \omega> \in ipm_p$.
  By property of the stabilize relation and $\texttt{comp}(id_p, \text{"place"}, gm_p, ipm_p, opm_p) \in d.cs$, we can deduce $\sigma'(id_p)(\text{"oat"})[j] = \texttt{inhib}$ and $\sigma'(id_p)(\text{"oaw"})[j] = \omega$.
  Rewriting the goal with $\sigma'(id_p)(\text{"oat"})[j] = \texttt{inhib}$ and $\sigma'(id_p)(\text{"oaw"})[j] = \omega$, and simplifying the goal: $\boxed{\sigma'(id_p)(\text{"sm"}) < \omega = \texttt{true}.}$
  Appealing to Lemma 24, we can deduce $s'.M(p) = \sigma'(id_p)(\text{"sm"})$.
  Rewriting the goal with $s'.M(p) = \sigma'(id_p)(\text{"sm"})$: $\boxed{s'.M(p) < \omega = \texttt{true}.}$
  By definition of $t \in Sens(s'.M)$, $s'.M(p) < \omega = \texttt{true}.$

2. Assuming that $\sigma'(id_t)(\text{"s\_enabled"}) = \texttt{true}$, let us show $\boxed{t \in Sens(s'.M).}$

   By definition of $t \in Sens(s'.M)$, let us show

   $$\boxed{\forall p \in P, \omega \in \mathbb{N}^*, \left(pre(p,t) = (\omega, \texttt{basic}) \vee pre(p,t) = (\omega, \texttt{test}) \Rightarrow s'.M(p) \geq \omega\right) \wedge \left(pre(p,t) = (\omega, \texttt{inhib}) \Rightarrow s'.M(p) < \omega\right)}$$

   Given a $p \in P$ and an $\omega \in \mathbb{N}^*$, let us show
   $\boxed{pre(p,t) = (\omega, \texttt{basic}) \vee pre(p,t) = (\omega, \texttt{test}) \Rightarrow s'.M(p) \geq \omega}$ and
   $\boxed{pre(p,t) = (\omega, \texttt{inhib}) \Rightarrow s'.M(p) < \omega.}$

   (a) Assuming $pre(p,t) = (\omega, \texttt{basic}) \vee pre(p,t) = (\omega, \texttt{test})$, let us show $\boxed{s'.M(p) \geq \omega.}$
   The proceeding is the same for $pre(p,t) = (\omega, \texttt{basic})$ and $pre(p,t) = (\omega, \texttt{test})$. Therefore, we will only cover the case where $pre(p,t) = (\omega, \texttt{basic})$.
   By property of the stabilize relation and $\texttt{comp}(id_t, \text{"transition"}, gm_t, ipm_t, opm_t) \in d.cs$, equation (C.17) holds.
   Rewriting $\sigma'(id_t)(\text{"se"}) = \texttt{true}$ with (C.17), we can deduce:
   $$\prod_{i=0}^{\Delta(id_t)(\text{"ian"})-1} \sigma'(id_t)(\text{"iav"})[i] = \texttt{true}.$$
   Then, we can deduce that $\forall i \in [0, \Delta(id_t)(\text{"ian"}) - 1]$, $\sigma'(id_t)(\text{"iav"})[i] = \texttt{true}$.

By construction, there exist an $id_p \in Comps(\Delta)$, gm_p, ipm_p, opm_p, $i \in [0, |input(t)| - 1]$, $j \in [0, |output(p)| - 1]$ and $id_{ji} \in Sigs(\Delta)$ s.t. $\gamma(p) = id_p$ and $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$ and $<\texttt{output\_arcs\_valid(j)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in opm_p$ and $<\texttt{input\_arcs\_valid(i)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in ipm_t$. Let us take such an $id_p \in Comps(\Delta)$, gm_p, ipm_p, opm_p, $i \in [0, |input(t)| - 1]$, $j \in [0, |output(p)| - 1]$ and $id_{ji} \in Sigs(\Delta)$.

By construction, $<\texttt{input\_arcs\_number} \Rightarrow |input(t)|> \in gm_t$.

By property of the elaboration relation and $\texttt{comp}(id_t, "transition", gm_t, ipm_t, opm_t) \in d.cs$, we can deduce $\Delta(id_t)("ian") = |input(t)|$.

Thanks to $\Delta(id_t)("ian") = |input(t)|$, we can deduce that $\forall i \in [0, |input(t)| - 1]$, $\sigma'(id_t)("iav")[i] = \texttt{true}$.

Having such an $i \in [0, |input(t)| - 1]$, we can deduce that $\sigma'(id_t)("iav")[i] = \texttt{true}$.

By property of the stabilize relation, $\texttt{comp}(id_t, "transition", gm_t, ipm_t, opm_t) \in d.cs$ and $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$, we can deduce $\sigma'(id_t)("iav")[i] = \sigma'(id_{ji}) = \sigma'(id_p)("oav")[j]$.

Thanks to $\sigma'(id_t)("iav")[i] = \sigma'(id_{ji}) = \sigma'(id_p)("oav")[j]$, we can deduce that $\sigma'(id_p)("oav")[j] = \texttt{true}$.

By property of the stabilize relation and $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$, equation (C.18) holds. Thanks to (C.18), we can deduce that:

$$\begin{aligned}\texttt{true} = &\big((\sigma'(id_p)("oat")[j] = \texttt{basic} + \sigma'(id_p)("oat")[j] = \texttt{test}) \\ &\quad . \sigma'(id_p)("sm") \geq \sigma'(id_p)("oaw")[j]\big) \\ &+ \big(\sigma'(id_p)("oat")[j] = \texttt{inhib} \,.\, \sigma'(id_p)("sm") < \sigma'(id_p)("oaw")[j]\big)\end{aligned} \tag{C.19}$$

By construction, $<\texttt{output\_arcs\_types(j)} \Rightarrow \texttt{basic}> \in ipm_p$ and $<\texttt{output\_arcs\_weights(j)} \Rightarrow \omega> \in ipm_p$.

By property of the stabilize relation and $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$, we can deduce $\sigma'(id_p)("oat")[j] = \texttt{basic}$ and $\sigma'(id_p)("oaw")[j] = \omega$.

Thanks to $\sigma'(id_p)("oat")[j] = \texttt{basic}$, $\sigma'(id_p)("oaw")[j] = \omega$, and simplifying Equation (C.19), we can deduce $\sigma'(id_p)("sm") \geq \omega = \texttt{true}$.

Appealing to Lemma 24, $s'.M(p) \geq \omega$.

(b) Assuming $pre(p,t) = (\omega, \texttt{inhib})$, let us show $\boxed{s'.M(p) < \omega.}$

The proceeding is the same as in the preceding case. Here, we will start the proof where the two cases are diverging, i.e:

By construction, $<\texttt{output\_arcs\_types(j)} \Rightarrow \texttt{inhib}> \in ipm_p$ and $<\texttt{output\_arcs\_weights(j)} \Rightarrow \omega> \in ipm_p$.

By property of the stabilize relation and $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$, we can deduce $\sigma'(id_p)("oat")[j] = \texttt{inhib}$ and $\sigma'(id_p)("oaw")[j] = \omega$.

Thanks to $\sigma'(id_p)("oat")[j] = \texttt{inhib}$ and $\sigma'(id_p)("oaw")[j] = \omega$, and simplifying Equation (C.19), we can deduce $\sigma'(id_p)("sm") < \omega = \texttt{true}$.

Appealing to Lemma 24, $s'.M(p) < \omega$.

□

Lemma 31 (Rising edge equal not sensitized). *For all* $sitpn$, d, γ, E_c, E_p, τ, Δ, σ_e, s, s', σ, σ_i, $\sigma_\uparrow$, σ' *that verify the hypotheses of Def. 36, then*
$\forall t \in T, id_t \in Comps(\Delta)$ *s.t.* $\gamma(t) = id_t, t \notin Sens(s'.M) \Leftrightarrow \sigma'(id_t)(\text{"s\_enabled"}) = \texttt{false}$.

Proof. Proving the above lemma is trivial by appealing to Lemma 30 and by reasoning on contrapositives. □

C.4 Falling Edge

C.4.1 Falling Edge and marking

Lemma 32 (Falling edge equal marking). *For all* $sitpn$, d, γ, Δ, σ_e, E_c, E_p, τ, s, s', σ, σ_i, $\sigma_\downarrow$, σ' *that verify the hypotheses of Definition 27, then* $\forall p \in P, id_p \in Comps(\Delta)$ *s.t.* $\gamma(p) = id_p$, $s'.M(p) = \sigma'(id_p)(\text{"s\_marking"})$.

Proof. Given a $p \in P$ and an $id \in Comps(\Delta)$ s.t. $\gamma(p) = id_p$, let us show
$\boxed{s'.M(p) = \sigma'(id_p)(\text{"s\_marking"}).}$
By definition of $E_c, \tau \vdash sitpn, s \xrightarrow{\downarrow} s'$, we can deduce $s.M(p) = s'.M(p)$.
By property of the $\texttt{Inject}_\downarrow$ relation, the $\mathcal{H}$-VHDL falling edge relation, the stabilize relation and $\texttt{comp}(id_p, \text{"place"}, gm_p, ipm_p, opm_p) \in d.cs$, and through the examination of the `marking` process defined in the place design architecture, we can deduce $\sigma'(id_p)(\text{"s\_marking"}) = \sigma(id_p)(\text{"s\_marking"})$.
Rewriting the goal with $s.M(p) = s'.M(p)$ and $\sigma'(id_p)(\text{"sm"}) = \sigma(id_p)(\text{"sm"})$: $\boxed{s.M(p) = \sigma(id_p)(\text{"s\_ma}$
By definition of $\gamma, E_c, \tau \vdash s \overset{\downarrow}{\approx} \sigma$: $s.M(p) = \sigma(id_p)(\text{"s\_marking"})$.

□

Lemma 33 (Falling Edge Equal Output Token Sum). *For all* $sitpn$, d, γ, Δ, σ_e, E_c, E_p, τ, s, s', σ, σ_i, $\sigma_\downarrow$, σ' *that verify the hypotheses of Definition 27, then* $\forall p, id_p$ *s.t.* $\gamma(p) = id_p$, $\sum_{t \in Fired(s')} pre(p,t) = \sigma'(id_p)(\text{"s\_output\_token\_sum"})$.

Proof. Given a $p \in P$ and an $id_p \in Comps(\Delta)$, let us show
$\boxed{\sum_{t \in Fired(s')} pre(p,t) = \sigma'(id_p)(\text{"s\_output\_token\_sum"}).}$
By construction and by definition of id_p, there exist gm_p, ipm_p, opm_p s.t. $\texttt{comp}(id_p, \text{"place"}, gm_p, ipm_p, opm_p) \in d.cs$.
By property of the stabilize relation, $\texttt{comp}(id_p, \text{"place"}, gm_p, ipm_p, opm_p) \in d.cs$, and through the examination of the `output_tokens_sum` process defined in the place design architecture:

$$\sigma'(id_p)(\text{"sots"}) = \sum_{i=0}^{\Delta(id_p)(\text{"oan"})-1} \begin{cases} \sigma'(id_p)(\text{"oaw"})[i] \texttt{ if } (\sigma'(id_p)(\text{"otf"})[i] \\ \qquad\qquad . \sigma'(id_p)(\text{"oat"})[i] = \texttt{basic}) \\ 0 \textit{ otherwise} \end{cases} \tag{C.20}$$

Rewriting the goal with (C.20):

$$\sum_{t \in Fired(s')} pre(p,t) = \sum_{i=0}^{\Delta(id_p)("oan")-1} \begin{cases} \sigma'(id_p)("oaw")[i] \texttt{ if } (\sigma'(id_p)("otf")[i] \\ \qquad\qquad\qquad\qquad . \sigma'(id_p)("oat")[i] = \texttt{basic}) \\ 0 \textit{ otherwise} \end{cases}$$

Let us unfold the definition of the left sum term:

$$\sum_{t \in Fired(s')} \begin{cases} \omega \texttt{ if } pre(p,t) = (\omega, \texttt{basic}) \\ 0 \textit{ otherwise} \end{cases} = \sum_{i=0}^{\Delta(id_p)("oan")-1} \begin{cases} \sigma'(id_p)("oaw")[i] \texttt{ if } (\sigma'(id_p)("otf")[i] \\ \qquad\qquad\qquad\qquad . \sigma'(id_p)("oat")[i] = \texttt{basic}) \\ 0 \textit{ otherwise} \end{cases}$$

To ease the reading, let us define functions $f \in Fired(s') \to \mathbb{N}$ and $g \in [0, |output(p)| - 1] \to \mathbb{N}$

s.t. $f(t) = \begin{cases} \omega \texttt{ if } pre(p,t) = (\omega, \texttt{basic}) \\ 0 \textit{ otherwise} \end{cases}$ and $g(i) = \begin{cases} \sigma'(id_p)("oaw")[i] \texttt{ if } (\sigma'(id_p)("otf")[i] \\ \qquad\qquad\qquad\qquad . \sigma'(id_p)("oat")[i] = \texttt{basic}) \\ 0 \textit{ otherwise} \end{cases}$

Then, the goal is: $\sum_{t \in Fired(s')} f(t) = \sum_{i=0}^{\Delta(id_p)("oan")-1} g(i)$

Let us perform case analysis on $output(p)$; there are two cases:

- **CASE** $output(p) = \emptyset$:

  By construction, $<\texttt{output\_arcs\_number} \Rightarrow 1> \in gm_p$, $<\texttt{output\_arcs\_types(0)} \Rightarrow \texttt{basic}> \in ipm_p$, $<\texttt{output\_transitions\_fired(0)} \Rightarrow \texttt{true}> \in ipm_p$, and $<\texttt{output\_arcs\_weights(0)} \Rightarrow 0> \in ipm_p$.

  By property of the elaboration relation and $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$, we can deduce $\Delta(id_p)("oan") = 1$.

  By property of the stabilize relation and $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$, we can deduce $\sigma'(id_p)("oat")[0] = \texttt{basic}$, $\sigma'(id_p)("otf")[0] = \texttt{true}$ and $\sigma'(id_p)("oaw")[0] = 0$.

  By property of $output(p) = \emptyset$, we can deduce

  $$\sum_{t \in Fired(s')} \begin{cases} \omega \texttt{ if } pre(p,t) = (\omega, \texttt{basic}) \\ 0 \textit{ otherwise} \end{cases} = 0$$

  Rewriting the goal with $\Delta(id_p)("oan") = 1$, $\sigma'(id_p)("oat")[0] = \texttt{basic}$, $\sigma'(id_p)("otf")[0] = \texttt{true}$, $\sigma'(id_p)("oaw")[0] = 0$ and $\sum_{t \in Fired(s')} \begin{cases} \omega \texttt{ if } pre(p,t) = (\omega, \texttt{basic}) \\ 0 \textit{ otherwise} \end{cases} = 0$, tautology.

- **CASE** $output(p) \neq \emptyset$:

By construction, $<\texttt{output\_arcs\_number} \Rightarrow |output(p)|> \in gm_p$, and by property of the elaboration relation, we can deduce $\Delta(id_p)("oan") = |output(p)|$.

Rewriting the goal with $\Delta(id_p)("oan") = |output(p)|$: $\boxed{\sum_{t \in Fired(s')} f(t) = \sum_{i=0}^{|output(p)|-1} g(i).}$

Let us reason by induction on the right sum term of the goal.

– **BASE CASE**:

In that case, $0 > |output| - 1$ and $\sum_{i=0}^{|output(p)|-1} g(i) = 0$.

As $0 > |output| - 1$, then $|output(p)| = 0$, thus contradicting $output(p) \neq \emptyset$.

– **INDUCTION CASE**:

In that case, $0 \leq |output(p)| - 1$.

$$\forall F \subseteq Fired(s'), g(0) + \sum_{t \in F} f(t) = g(0) + \sum_{i=1}^{|output(p)|-1} g(i)$$

$$\boxed{\sum_{t \in Fired(s')} f(t) = g(0) + \sum_{i=1}^{|output(p)|-1} g(i)}$$

By definition of g:

$$g(0) = \begin{cases} \sigma'(id_p)("oaw")[0] \texttt{ if } (\sigma'(id_p)("otf")[0] \\ \qquad\qquad\qquad\qquad . \sigma'(id_p)("oat")[0] = \texttt{basic}) \\ 0 \ otherwise \end{cases} \tag{C.21}$$

Let us perform case analysis on the value of $\sigma'(id_p)("otf")[0] \,.\, \sigma'(id_p)("oat")[0] = \texttt{basic}$; there are two cases:

1. $(\sigma'(id_p)("otf")[0] \,.\, \sigma'(id_p)("oat")[0] = \texttt{basic}) = \texttt{false}$:
   In that case, $g(0) = 0$, and then we can apply the induction hypothesis with $F = Fired(s')$ to solve the goal: $\sum_{t \in Fired(s')} f(t) = \sum_{i=1}^{|output(p)|-1} g(i).$
2. $(\sigma'(id_p)("otf")[0] \,.\, \sigma'(id_p)("oat")[0] = \texttt{basic}) = \texttt{true}$:
   In that case, $g(0) = \sigma'(id_p)("oaw")[0]$, $\sigma'(id_p)("otf")[0] = \texttt{true}$ and $\sigma'(id_p)("oat")[0] = \texttt{basic}$.
   By construction, there exist a $t \in output(t)$, $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, and there exist gm_t, ipm_t, opm_t s.t. $\texttt{comp}(id_t, "transition", gm_t, ipm_t, opm_t) \in d.cs$, and there exist an $\omega \in \mathbb{N}^*$, an $a \in \{\texttt{basic}, \texttt{test}, \texttt{inhib}\}$ and an $id_{ft} \in Sigs(\Delta)$ such that:
   * $pre(p, t) = (\omega, a)$

* <$\texttt{output\_arcs\_types}(0) \Rightarrow a$> $\in ipm_p$
* <$\texttt{output\_arcs\_weights}(0) \Rightarrow \omega$> $\in ipm_p$
* <$\texttt{fired} \Rightarrow id_{ft}$> $\in opm_t$
* <$\texttt{output\_transitions\_fired}(0) \Rightarrow \texttt{id}_{\texttt{ft}}$> $\in ipm_p$

By property of the stabilize relation, $\sigma'(id_p)(''oat'')[0] = \texttt{basic}$ and <$\texttt{output\_arcs\_types}(0) \Rightarrow \texttt{a}$> $\in ipm_p$, we can deduce $pre(p,t) = (\omega, \texttt{basic})$.
By property of the stabilize relation, <$\texttt{fired} \Rightarrow \texttt{id}_{\texttt{ft}}$> $\in opm_t$, <$\texttt{output\_transitions\_fired}(0) \Rightarrow \texttt{id}_{\texttt{ft}}$> $\in ipm_p$ and $\sigma'(id_p)(''otf'')[0] = \texttt{true}$, we can deduce $\sigma'(id_t)(''fired'') = \texttt{true}$.
Appealing to Lemma 4, and thanks to $\sigma'(id_t)(''fired'') = \texttt{true}$, we can deduce $t \in Fired(s')$.
With $t \in Fired(s')$, we can rewrite the left sum term of the goal as follows:

$$f(t) + \sum_{t' \in Fired(s') \setminus \{t\}} f(t') = g(0) + \sum_{i=1}^{|output(p)|-1} g(i)$$

We know that $g(0) = \sigma'(id_p)(''oaw'')[0]$, and by property of the stabilize relation and <$\texttt{output\_arcs\_weights}(0) \Rightarrow \omega$> $\in ipm_p$, we can deduce $\sigma'(id_p)(''oaw'')[0] = \omega$.
Rewriting the goal with $\sigma'(id_p)(''oaw'')[0] = \omega$:

$$f(t) + \sum_{t' \in Fired(s') \setminus \{t\}} f(t') = \omega + \sum_{i=1}^{|output(p)|-1} g(i)$$

By definition of f, and as $pre(p,t) = (\omega, \texttt{basic})$, then $f(t) = \omega$; thus, rewriting the goal:

$$\omega + \sum_{t' \in Fired(s') \setminus \{t\}} f(t') = \omega + \sum_{i=1}^{|output(p)|-1} g(i)$$

Then, knowing that $g(0) = \omega$, we can apply the induction hypothesis with $F = Fired(s') \setminus \{t\}$: $g(0) + \sum_{t' \in Fired(s') \setminus \{t\}} f(t') = g(0) + \sum_{i=1}^{|output(p)|-1} g(i)$.

□

Lemma 34 (Falling Edge Equal Input Token Sum). *For all $sitpn$, d, γ, Δ, σ_e, E_c, E_p, τ, s, s', σ, σ_i, $\sigma_\downarrow$, σ' that verify the hypotheses of Definition 27, then $\forall p, id_p$ s.t. $\gamma(p) = id_p$, $\sum_{t \in Fired(s')} post(t,p) = \sigma'_p(''s\_input\_token\_sum'')$.*

Proof. Given a $p \in P$ and an $id_p \in Comps(\Delta)$, let us show

$$\sum_{t \in Fired(s')} post(t,p) = \sigma'(id_p)(''s\_input\_token\_sum'').$$

By construction and by definition of id_p, there exist gm_p, ipm_p, opm_p s.t. $\texttt{comp}(id_p, ''place'', gm_p, ipm_p, opm_p) \in d.cs$.

By property of the stabilize relation, $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$, and through the examination of the `input_tokens_sum` process defined in the place design architecture:

$$\sigma'(id_p)("sits") = \sum_{i=0}^{\Delta(id_p)("ian")-1} \begin{cases} \sigma'(id_p)("iaw")[i] \texttt{ if } \sigma'(id_p)("itf")[i] \\ 0 \textit{ otherwise} \end{cases} \tag{C.22}$$

Rewriting the goal with (C.22):

$$\sum_{t \in Fired(s')} post(t,p) = \sum_{i=0}^{\Delta(id_p)("ian")-1} \begin{cases} \sigma'(id_p)("iaw")[i] \texttt{ if } \sigma'(id_p)("otf")[i] \\ 0 \textit{ otherwise} \end{cases}$$

Let us unfold the definition of the left sum term:

$$\sum_{t \in Fired(s')} \begin{cases} \omega \texttt{ if } post(t,p) = \omega \\ 0 \textit{ otherwise} \end{cases} = \sum_{i=0}^{\Delta(id_p)("ian")-1} \begin{cases} \sigma'(id_p)("iaw")[i] \texttt{ if } \sigma'(id_p)("itf")[i] \\ 0 \textit{ otherwise} \end{cases}$$

Let us perform case analysis on $input(p)$; there are two cases:

- **CASE** $input(p) = \emptyset$:

  By construction, $<$`input_arcs_number`$\Rightarrow$ `1`$> \in gm_p$, $<$`input_transitions_fired(0)`$\Rightarrow$ `true`$> \in ipm_p$, and $<$`input_arcs_weights(0)`$\Rightarrow$ `0`$> \in ipm_p$.

  By property of the elaboration relation and $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$, we can deduce $\Delta(id_p)("ian") = 1$.

  By property of the stabilize relation and $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$, we can deduce $\sigma'(id_p)("itf")[0] = \texttt{true}$ and $\sigma'(id_p)("iaw")[0] = 0$.

  By property of $input(p) = \emptyset$, we can deduce $\sum_{t \in Fired(s')} \begin{cases} \omega \texttt{ if } post(t,p) = \omega \\ 0 \textit{ otherwise} \end{cases} = 0$.

  Rewriting the goal with $\Delta(id_p)("ian") = 1$, $\sigma'(id_p)("itf")[0] = \texttt{true}$, $\sigma'(id_p)("iaw")[0] = 0$, and $\sum_{t \in Fired(s')} \begin{cases} \omega \texttt{ if } post(t,p) = \omega \\ 0 \textit{ otherwise} \end{cases} = 0$, and simplifying the goal: tautology.

- **CASE** $input(p) \neq \emptyset$:

  By construction, $<$`input_arcs_number` $\Rightarrow |input(p)|> \in gm_p$, and by property of the elaboration relation, we can deduce $\Delta(id_p)("ian") = |input(p)|$.

To ease the reading, let us define functions $f \in Fired(s') \to \mathbb{N}$ and $g \in [0, |input(p)| - 1] \to \mathbb{N}$ s.t. $f(t) = \begin{cases} \omega \texttt{ if } post(t,p) = \omega \\ 0 \textit{ otherwise} \end{cases}$ and $g(i) = \begin{cases} \sigma'(id_p)(\text{"}iaw\text{"})[i] \texttt{ if } \sigma'(id_p)(\text{"}itf\text{"})[i] \\ 0 \textit{ otherwise} \end{cases}$

Then, the goal is: $\boxed{\sum_{t \in Fired(s')} f(t) = \sum_{i=0}^{\Delta(id_p)(\text{"}ian\text{"})-1} g(i)}$

Rewriting the goal with $\Delta(id_p)(\text{"}ian\text{"}) = |input(p)|$: $\boxed{\sum_{t \in Fired(s')} f(t) = \sum_{i=0}^{|input(p)|-1} g(i).}$

Let us reason by induction on the right sum term of the goal.

– **BASE CASE**: In that case, $0 > |input(p)| - 1$ and $\sum_{i=0}^{|input(p)|-1} g(i) = 0$.

As $0 > |input(p)| - 1$, then $|input(p)| = 0$, thus contradicting $input(p) \neq \emptyset$.

– **INDUCTION CASE**: In that case, $0 \leq |input(p)| - 1$.

$$\forall F \subseteq Fired(s'),\ g(0) + \sum_{t \in F} f(t) = g(0) + \sum_{i=1}^{|input(p)|-1} g(i)$$

$$\boxed{\sum_{t \in Fired(s')} f(t) = g(0) + \sum_{i=1}^{|input(p)|-1} g(i)}$$

By definition of g, we can deduce $g(0) = \begin{cases} \sigma'(id_p)(\text{"}iaw\text{"})[0] \texttt{ if } \sigma'(id_p)(\text{"}itf\text{"})[0] \\ 0 \textit{ otherwise} \end{cases}$

Let us perform case analysis on the value of $\sigma'(id_p)(\text{"}itf\text{"})[0]$; there are two cases:

1. $\sigma'(id_p)(\text{"}itf\text{"})[0] = \texttt{false}$:
   In that case, $g(0) = 0$, and then we can apply the induction hypothesis with $F = Fired(s')$ to solve the goal: $\sum_{t \in Fired(s')} f(t) = \sum_{i=1}^{|input(p)|-1} g(i).$
2. $\sigma'(id_p)(\text{"}itf\text{"})[0] = \texttt{true}$:
   In that case, $g(0) = \sigma'(id_p)(\text{"}iaw\text{"})[0]$ and $\sigma'(id_p)(\text{"}itf\text{"})[0] = \texttt{true}$.
   By construction, there exist a $t \in input(t)$, an $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, gm_t, ipm_t, opm_t s.t. $\texttt{comp}(id_t, \text{"}transition\text{"}, gm_t, ipm_t, opm) \in d.cs$, an $\omega \in \mathbb{N}^*$ and an $id_{ft} \in Sigs(\Delta)$ such that:
   * $post(t,p) = \omega$
   * $<\texttt{input\_arcs\_weights}(0) \Rightarrow \omega> \in ipm_p$
   * $<\texttt{fired} \Rightarrow id_{ft}> \in opm_t$
   * $<\texttt{input\_transitions\_fired}(0) \Rightarrow \texttt{id}_{\texttt{ft}}> \in ipm_p$

By property of the stabilize relation, $<\texttt{fired}\Rightarrow \texttt{id}_{\texttt{ft}}> \in opm_t$, $<\texttt{input\_transitions\_fired(0)}\Rightarrow \texttt{id}_{\texttt{ft}}> \in ipm_p$ and $\sigma'(id_p)("itf")[0] = \texttt{true}$, we can deduce $\sigma'(id_t)("fired") = \texttt{true}$.
Appealing to Lemma 4 and $\sigma'(id_t)("fired") = \texttt{true}$, we can deduce $t \in Fired(s')$.
As $t \in Fired(s')$, we can rewrite the left sum term of the goal as follows:

$$f(t) + \sum_{t' \in Fired(s') \setminus \{t\}} f(t') = g(0) + \sum_{i=1}^{|input(p)|-1} g(i)$$

We know that $g(0) = \sigma'(id_p)("iaw")[0]$, and by property of the stabilize relation and $<\texttt{input\_arcs\_weights(0)}\Rightarrow \omega> \in ipm_p$, we can deduce $\sigma'(id_p)("iaw")[0] = \omega$.
Rewriting the goal with $\sigma'(id_p)("iaw")[0] = \omega$:

$$f(t) + \sum_{t' \in Fired(s') \setminus \{t\}} f(t') = \omega + \sum_{i=1}^{|input(p)|-1} g(i)$$

By definition of f, and as $post(t,p) = \omega$, then $f(t) = \omega$; thus, rewriting the goal:

$$\omega + \sum_{t' \in Fired(s') \setminus \{t\}} f(t') = \omega + \sum_{i=1}^{|input(p)|-1} g(i)$$

Then, knowing that $g(0) = \omega$, we can apply the induction hypothesis with $F = Fired(s') \setminus \{t\}$: $g(0) + \sum_{t' \in Fired(s') \setminus \{t\}} f(t') = g(0) + \sum_{i=1}^{|input(p)|-1} g(i)$.

□

C.4.2 Falling edge and time counters

Lemma 35 (Falling edge equal time counters). *For all sitpn, d, γ, Δ, σ_e, E_c, E_p, τ, s, s', σ, σ_i, $\sigma_\downarrow$, σ' that verify the hypotheses of Definition 27, then $\forall t \in T_i, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$,*

$$\begin{aligned}
&\big(upper(I_s(t)) = \infty \wedge s'.I(t) \leq lower(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)("s_time_counter")\big)\\
&\wedge\big(upper(I_s(t)) = \infty \wedge s'.I(t) > lower(I_s(t)) \Rightarrow \sigma'(id_t)("s_time_counter") = lower(I_s(t))\big)\\
&\wedge\big(upper(I_s(t)) \neq \infty \wedge s'.I(t) > upper(I_s(t)) \Rightarrow \sigma'(id_t)("s_time_counter") = upper(I_s(t))\big)\\
&\wedge\big(upper(I_s(t)) \neq \infty \wedge s'.I(t) \leq upper(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)("s_time_counter")\big).
\end{aligned}$$

Proof. Given a $t \in T_i$ and an $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, let us show

$$\begin{aligned}
&\big(upper(I_s(t)) = \infty \wedge s'.I(t) \leq lower(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)("s_time_counter")\big)\\
&\wedge\big(upper(I_s(t)) = \infty \wedge s'.I(t) > lower(I_s(t)) \Rightarrow \sigma'(id_t)("s_time_counter") = lower(I_s(t))\big)\\
&\wedge\big(upper(I_s(t)) \neq \infty \wedge s'.I(t) > upper(I_s(t)) \Rightarrow \sigma'(id_t)("s_time_counter") = upper(I_s(t))\big)\\
&\wedge\big(upper(I_s(t)) \neq \infty \wedge s'.I(t) \leq upper(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)("s_time_counter")\big)
\end{aligned}$$

By construction and by definition of id_t, there exist gm_t, ipm_t, opm_t s.t. $\texttt{comp}(id_t, "transition", gm_t, ipm_t, opm_t) \in d.cs$.
By property of the elaboration, $\texttt{Inject}_\downarrow$, $\mathcal{H}$-VHDL rising edge and stabilize relations, $\texttt{comp}(id_t, "transition", gm_t, ipm_t, opm_t) \in d.cs$, and through the examination of the `time_counter` process

defined in the transition design architecture, we can deduce:

$$\sigma(id_t)(\text{"se"}) = \texttt{true} \wedge \Delta(id_t)(\text{"tt"}) \neq \texttt{NOT\_TEMPORAL} \wedge \sigma(id_t)(\text{"srtc"}) = \texttt{false} \\ \wedge \sigma(id_t)(\text{"stc"}) < \Delta(id_t)(\text{"mtc"}) \Rightarrow \sigma'(id_t)(\text{"stc"}) = \sigma(id_t)(\text{"stc"}) + 1 \tag{C.23}$$

$$\sigma(id_t)(\text{"se"}) = \texttt{true} \wedge \Delta(id_t)(\text{"tt"}) \neq \texttt{NOT\_TEMPORAL} \wedge \sigma(id_t)(\text{"srtc"}) = \texttt{false} \\ \wedge \sigma(id_t)(\text{"stc"}) \geq \Delta(id_t)(\text{"mtc"}) \Rightarrow \sigma'(id_t)(\text{"stc"}) = \sigma(id_t)(\text{"stc"}) \tag{C.24}$$

$$\sigma(id_t)(\text{"se"}) = \texttt{true} \wedge \Delta(id_t)(\text{"tt"}) \neq \texttt{NOT\_TEMPORAL} \\ \wedge \sigma(id_t)(\text{"srtc"}) = \texttt{true} \Rightarrow \sigma'(id_t)(\text{"stc"}) = 1 \tag{C.25}$$

$$\sigma(id_t)(\text{"se"}) = \texttt{false} \vee \Delta(id_t)(\text{"tt"}) = \texttt{NOT\_TEMPORAL} \Rightarrow \sigma'(id_t)(\text{"stc"}) = 0 \tag{C.26}$$

Then, there are 4 points to show:

1. $\boxed{upper(I_s(t)) = \infty \wedge s'.I(t) \leq lower(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(\text{"s\_time\_counter"})}$

   Assuming $upper(I_s(t)) = \infty$ and $s'.I(t) \leq lower(I_s(t))$, let us show $\boxed{s'.I(t) = \sigma'(id_t)(\text{"s\_time\_counter"}).}$

   Let us perform case analysis on $t \in Sens(s.M)$; there are two cases:

   (a) **CASE** $t \notin Sens(s.M)$:

   By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we can deduce $\sigma(id_t)(\text{"se"}) = \texttt{false}$.
   Appealing to (C.26) and $\sigma(id_t)(\text{"se"}) = \texttt{false}$, we can deduce $\sigma'(id_t)(\text{"stc"}) = 0$.
   By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$ (Rule (3)), we can deduce $s'.I(t) = 0$.
   Rewriting the goal with $\sigma'(id_t)(\text{"stc"}) = 0$ and $s'.I(t) = 0$: tautology.

   (b) **CASE** $t \in Sens(s.M)$:

   By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we can deduce $\sigma(id_t)(\text{"se"}) = \texttt{true}$.
   By construction, and as $upper(I_s(t)) = \infty$, $<\texttt{transition\_type} \Rightarrow \texttt{TEMP\_A\_INF}> \in gm_t$. By property of the elaboration relation, we have $\Delta(id_t)(\text{"tt"}) = \texttt{TEMP\_A\_INF}$.
   Let us perform case analysis on $s.reset_t(t)$; there are two cases:

   i. **CASE** $s.reset_t(t) = \texttt{true}$:

   By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, $\sigma(id_t)(\text{"srtc"}) = \texttt{true}$.
   Appealing to (C.25), $\sigma(id_t)(\text{"se"}) = \texttt{true}$, $\Delta(id_t)(\text{"tt"}) = \texttt{TEMP\_A\_INF}$ and $\sigma(id_t)(\text{"srtc"}) = \texttt{true}$, we can deduce $\sigma'(id_t)(\text{"stc"}) = 1$.

By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$(Rule (3)), we can deduce $s'.I(t) = 1$.
Rewriting the goal with $\sigma'(id_t)(''stc'') = 1$ and $s'.I(t) = 1$: tautology.

ii. **CASE** $s.reset_t(t) = \texttt{false}$:

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $\sigma(id_t)(''srtc'') = \texttt{false}$.
As $upper(I_s(t)) = \infty$, there exists an $a \in \mathbb{N}^*$ s.t. $I_s(t) = [a, \infty]$. Let us take such an $a \in \mathbb{N}^*$. By construction, $<\texttt{maximal\_time\_counter} \Rightarrow a> \in gm_t$, and by property of the elaboration relation, we have $\Delta(id_t)(''mtc'') = a$.

By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$(Rule (4)), and knowing that $t \in Sens(s.M)$, $s.reset_t(t) = \texttt{false}$ and $upper(I_s(t)) = \infty$, we can deduce $s'.I(t) = s.I(t) + 1$.
Rewriting the goal with $s'.I(t) = s.I(t) + 1$: $\boxed{s.I(t) + 1 = \sigma'(id_t)(''stc'').}$
We assumed that $s'.I(t) \leq lower(I_s(t))$, and as $s'.I(t) = s.I(t) + 1$, then $s.I(t) + 1 \leq lower(I_s(t))$, then $s.I(t) < lower(I_s(t))$, then $s.I(t) < a$ since $a = lower(I_s(t))$.

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, and knowing that $s.I(t) < lower(I_s(t))$ and $upper(I_s(t)) = \infty$, we can deduce $s.I(t) = \sigma(id_t)(''stc'')$.
Appealing to $\Delta(id_t)(''mtc'') = a$, $s.I(t) = \sigma(id_t)(''stc'')$ and $s.I(t) < a$, we can deduce $\sigma(id_t)(''stc'') < \Delta(id_t)(''mtc'')$.
Appealing to (C.23), $\sigma(id_t)(''stc'') < \Delta(id_t)(''mtc'')$, $\sigma(id_t)(''srtc'') = \texttt{false}$ and $\sigma(id_t)(''se'') = \texttt{true}$, we can deduce: $\sigma'(id_t)(''stc'') = \sigma(id_t)(''stc'') + 1$.
Rewriting the goal with $\sigma'(id_t)(''stc'') = \sigma(id_t)(''stc'') + 1$ and $s.I(t) = \sigma(id_t)(''stc'')$:
tautology.

2. $\boxed{upper(I_s(t)) = \infty \wedge s'.I(t) > lower(I_s(t)) \Rightarrow \sigma'(id_t)(''s\_time\_counter'') = lower(I_s(t).}$

Assuming that $upper(I_s(t)) = \infty$ and $s'.I(t) > lower(I_s(t))$, let us show
$\boxed{\sigma'(id_t)(''s\_time\_counter'') = lower(I_s(t)).}$

As $upper(I_s(t)) = \infty$, there exists an $a \in \mathbb{N}^*$ s.t. $I_s(t) = [a, \infty]$. Let us take such an $a \in \mathbb{N}^*$.

By construction, $<\texttt{maximal\_time\_counter} \Rightarrow a> \in gm_t$, and $<\texttt{transition\_type} \Rightarrow \texttt{TEMP\_A\_INF}> \in gm_t$ by property of the elaboration relation, we can deduce $\Delta(id_t)(''mtc'') = a$ and $\Delta(id_t)(''tt'') = \texttt{TEMP\_A\_INF}$.

Let us perform case analysis on $t \in Sens(s.M)$:

(a) **CASE** $t \notin Sens(s.M)$:

By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$ (Rule (6)), and knowing that $t \in Sens(s.M)$, we can deduce $s'.I(t) = 0$. Since $lower(I_s(t)) \in \mathbb{N}^*$, then $lower(I_s(t)) > 0$.
Contradicts $s'.I(t) > lower(I_s(t))$.

(b) **CASE** $t \in Sens(s.M)$:

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\sim} \sigma$ and $t \in Sens(s.M)$, we can deduce $\sigma(id_t)(''se'') = \texttt{true}$.
Let us perform case analysis on $s.reset_t(t)$; there are two cases:

i. **CASE** $s.reset_t(t) = \texttt{true}$:

By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$: $s'.I(t) = 1$.
We assumed that $s'.I(t) > lower(I_s(t))$, then $1 > lower(I_s(t))$.
Contradicts $lower(I_s(t)) > 0$.

ii. **CASE** $s.reset_t(t) = \texttt{false}$:

By property of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$ and $s.reset_t(t) = \texttt{false}$, we can deduce $\sigma(id_t)(''srtc'') = \texttt{false}$.
By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$ (Rule (4)), and knowing that $s'.I(t) > lower(I_s(t))$, we can deduce

$$\begin{aligned} s'.I(t) = s.I(t) + 1 &\Rightarrow s.I(t) + 1 > lower(I_s(t)) \\ &\Rightarrow s.I(t) \geq lower(I_s(t)) \end{aligned}$$

Let us perform case analysis on $s.I(t) \geq lower(I_s(t))$:

A. **CASE** $s.I(t) > lower(I_s(t))$: $\boxed{\sigma'(id_t)(''stc'') = lower(I_s(t)).}$

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we can deduce $\sigma(id_t)(''stc'') = lower(I_s(t))$.
Appealing to (C.24), we can deduce $\sigma'(id_t)(''stc'') = \sigma(id_t)(''stc'')$.
Rewriting the goal with $\sigma'(id_t)(''stc'') = \sigma(id_t)(''stc'')$ and $\sigma(id_t)(''stc'') = lower(I_s(t))$:
tautology.

B. **CASE** $s.I(t) = lower(I_s(t))$: $\boxed{\sigma'(id_t)(''stc'') = lower(I_s(t)).}$

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we can deduce $s.I(t) = \sigma(id_t)(''stc'')$.
Appealing to (C.24), we can deduce $\sigma'(id_t)(''stc'') = \sigma(id_t)(''stc'')$.
Rewriting the goal with $\sigma'(id_t)(''stc'') = \sigma(id_t)(''stc'')$, $s.I(t) = \sigma(id_t)(''stc'')$ and $s.I(t) = lower(I_s(t))$: tautology.

3. $\boxed{upper(I_s(t)) \neq \infty \wedge s'.I(t) > upper(I_s(t)) \Rightarrow \sigma'(id_t)(''s\_time\_counter'') = upper(I_s(t)).}$

Assuming that $upper(I_s(t)) \neq \infty$ and $s'.I(t) > upper(I_s(t))$, let us show
$\boxed{\sigma'(id_t)(''s\_time\_counter'') = upper(I_s(t)).}$

As $upper(I_s(t)) \neq \infty$, there exists an $a \in \mathbb{N}^*$, and a $b \in \mathbb{N}^*$ s.t. $I_s(t) = [a, b]$. Let us take such an a and b.

By construction, $<\texttt{maximal\_time\_counter} \Rightarrow b> \in gm_t$ and there exists $tt \in \{\texttt{TEMP\_A\_A}, \texttt{TEMP\_A\_B}\}$ s.t. $<\texttt{transition\_type} \Rightarrow tt> \in gm_t$.

By property of the elaboration relation and $\texttt{comp}(id_t, ''transition'', gm_t, ipm_t, opm_t) \in d.cs$, we can deduce $\Delta(id_t)(''mtc'') = b = upper(I_s(t))$ and $\Delta(id_t)(''tt'') \neq \texttt{NOT\_TEMP}$.

Let us perform case analysis on $t \in Sens(s.M)$:

(a) **CASE** $t \notin Sens(s.M)$:

By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$ (Rule (6)), and knowing that $t \in Sens(s.M)$, then $s'.I(t) = 0$. Since $upper(I_s(t)) \in \mathbb{N}^*$, then $upper(I_s(t)) > 0$.

Contradicts $s'.I(t) > upper(I_s(t))$.

(b) **CASE** $t \in Sens(s.M)$:

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$ and $t \in Sens(s.M)$, we can deduce $\sigma(id_t)(''se'') = \texttt{true}$.
Let us perform case analysis on $s.reset_t(t)$; there are two cases:

i. **CASE** $s.reset_t(t) = \texttt{true}$:

By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$ (Rule (3)), we can deduce $s'.I(t) = 1$.
We assumed that $s'.I(t) > upper(I_s(t))$, then we can deduce $1 > upper(I_s(t))$.
Contradicts $upper(I_s(t)) > 0$.

ii. **CASE** $s.reset_t(t) = \texttt{false}$:

By property of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$ and $s.reset_t(t) = \texttt{false}$, we can deduce $\sigma(id_t)(''srtc'') = \texttt{false}$.
Let us perform case analysis on $s.I(t) > upper(I_s(t))$ or $s.I(t) \leq upper(I_s(t))$:

A. **CASE** $s.I(t) > upper(I_s(t))$: $\boxed{\sigma'(id_t)(''stc'') = upper(I_s(t)).}$

By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$(Rule (5)), we can deduce $s'.I(t) = s.I(t)$.
By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we can deduce $\sigma(id_t)(''stc'') = upper(I_s(t))$.
Appealing to (C.24), we have $\sigma'(id_t)(''stc'') = \sigma(id_t)(''stc'')$.
Rewriting the goal with $\sigma'(id_t)(''stc'') = \sigma(id_t)(''stc'')$ and $\sigma(id_t)(''stc'') = upper(I_s(t))$:
tautology.

B. **CASE** $s.I(t) \leq upper(I_s(t))$: $\boxed{\sigma'(id_t)(''stc'') = upper(I_s(t)).}$

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we can deduce $s.I(t) = \sigma(id_t)(''stc'')$.
Let us perform case analysis on $s.I(t) \leq upper(I_s(t))$; there are two cases:

- **CASE** $s.I(t) = upper(I_s(t))$:

  Appealing to $\Delta(id_t)(''mtc'') = b = upper(I_s(t))$, $s.I(t) = \sigma(id_t)(''stc'')$ and $s.I(t) = upper(I_s(t))$, we can deduce $\Delta(id_t)(''mtc'') \leq \sigma(id_t)(''stc'')$.
  Appealing to $\Delta(id_t)(''mtc'') \leq \sigma(id_t)(''stc'')$ and (C.24), we can deduce $\sigma'(id_t)(''stc'') = \sigma(id_t)(''stc'')$.
  Rewriting the goal with $\sigma'(id_t)(''stc'') = \sigma(id_t)(''stc'')$, $s.I(t) = \sigma(id_t)(''stc'')$ and $s.I(t) = upper(I_s(t))$: tautology.

- **CASE** $s.I(t) < upper(I_s(t))$:

  By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$(Rule (4)), we can deduce $s'.I(t) = s.I(t) + 1$.

From $s'.I(t) = s.I(t) + 1$ and $s.I(t) < upper(I_s(t))$, we can deduce $s'.I(t) \leq upper(I_s(t))$; contradicts $s'.I(t) > upper(I_s(t))$.

4. $\boxed{upper(I_s(t)) \neq \infty \wedge s'.I(t) \leq upper(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(\text{"s\_time\_counter"}).}$

Assuming that $upper(I_s(t)) \neq \infty$ and $s'.I(t) \leq upper(I_s(t))$, let us show $\boxed{s'.I(t) = \sigma'(id_t)(\text{"s\_time\_counter"}).}$

As $upper(I_s(t)) \neq \infty$, there exists an $a \in \mathbb{N}^*$, and a $b \in \mathbb{N}^*$ s.t. $I_s(t) = [a, b]$. Let us take such an a and b.

By construction, <maximal_time_counter $\Rightarrow b>$ $\in gm_t$ and there exists $tt \in \{$TEMP_A_A, TEMP_A_B$\}$ s.t. <transition_type $\Rightarrow tt>$ $\in gm_t$; by property of the elaboration relation, we can deduce $\Delta(id_t)(\text{"mtc"}) = b = upper(I_s(t))$ and $\Delta(id_t)(\text{"tt"}) \neq$ NOT_TEMP.

Let us perform case analysis on $t \in Sens(s.M)$:

(a) **CASE** $t \notin Sens(s.M)$:

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $\sigma(id_t)(\text{"se"}) =$ false.
Appealing (C.26) and $\sigma(id_t)(\text{"se"}) =$ false, we have $\sigma'(id_t)(\text{"stc"}) = 0$.
By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$(Rule (6)), we have $s'.I(t) = 0$.
Rewriting the goal with $\sigma'(id_t)(\text{"stc"}) = 0$ and $s'.I(t) = 0$: tautology.

(b) **CASE** $t \in Sens(s.M)$:

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $\sigma(id_t)(\text{"se"}) =$ true.
Let us perform case analysis on $s.reset_t(t)$:

i. **CASE** $s.reset_t(t) =$ true:

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $\sigma(id_t)(\text{"srtc"}) =$ true.
Appealing to (C.25), $\Delta(id_t)(\text{"tt"}) \neq$ NOT_TEMP, $\sigma(id_t)(\text{"se"}) =$ true and $\sigma(id_t)(\text{"srtc"}) =$ true, we have $\sigma'(id_t)(\text{"stc"}) = 1$.
By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$(Rule (3)), we have $s'.I(t) = 1$.
Rewriting the goal with $\sigma'(id_t)(\text{"stc"}) = 1$ and $s'.I(t) = 1$, tautology.

ii. **CASE** $s.reset_t(t) =$ false:

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $\sigma(id_t)(\text{"srtc"}) =$ false.
Let us perform case analysis on $s.I(t) > upper(I_s(t))$ or $s.I(t) \leq upper(I_s(t))$:
A. **CASE** $s.I(t) > upper(I_s(t))$:

By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$, we have $s.I(t) = s'.I(t)$, and thus, $s'.I(t) > upper(I_s(t))$. Contradicts $s'.I(t) \leq upper(I_s(t))$.

B. **CASE** $s.I(t) \leq upper(I_s(t))$:

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $s.I(t) = \sigma(id_t)(\text{"}stc\text{"})$.

- **CASE** $s.I(t) < upper(I_s(t))$:
  From $s.I(t) < upper(I_s(t))$, $s.I(t) = \sigma(id_t)(\text{"}stc\text{"})$ and $\Delta(id_t)(\text{"}mtc\text{"}) = b = upper(I_s(t))$, we can deduce $\sigma(id_t)(\text{"}stc\text{"}) < \Delta(id_t)(\text{"}mtc\text{"})$.
  From (C.23), $\sigma(id_t)(\text{"}se\text{"}) = \texttt{true}$, $\Delta(id_t)(\text{"}tt\text{"}) \neq \texttt{NOT\_TEMP}$, $\sigma(id_t)(\text{"}srtc\text{"}) = \texttt{false}$ and $\sigma(id_t)(\text{"}stc\text{"}) < \Delta(id_t)(\text{"}mtc\text{"})$, we can deduce $\sigma'(id_t)(\text{"}stc\text{"}) = \sigma(id_t)(\text{"}stc\text{"}) + 1$.
  By definition of $E_c, \tau \vdash s \overset{\downarrow}{\to} s'$(Rule (4)), we can deduce $s'.I(t) = s.I(t) + 1$.
  Rewriting the goal with $\sigma'(id_t)(\text{"}stc\text{"}) = \sigma(id_t)(\text{"}stc\text{"}) + 1$ and $s'.I(t) = s.I(t) + 1$, tautology.
- **CASE** $s.I(t) = upper(I_s(t))$:
  By definition of $E_c, \tau \vdash s \overset{\downarrow}{\to} s'$(Rule (4)), we know that $s'.I(t) = s.I(t) + 1$. We assumed that $s'.I(t) \leq upper(I_s(t))$; thus, $s.I(t) + 1 \leq upper(I_s(t))$.
  Contradicts $s.I(t) = upper(I_s(t))$.

□

C.4.3 Falling edge and condition values

Lemma 36 (Falling edge equal condition values)**.** *For all sitpn, $d, \gamma, \Delta, \sigma_e, E_c, E_p, \tau, s, s', \sigma, \sigma_i, \sigma_\downarrow$, σ' that verify the hypotheses of Definition 27, then $\forall c \in \mathcal{C}, id_c \in Ins(\Delta)$ s.t. $\gamma(c) = id_c$, $s'.cond(c) = \sigma'(id_c)$.*

Proof. Given a $c \in \mathcal{C}$ and an $id_c \in Ins(\Delta)$ s.t. $\gamma(c) = id_c$, let us show $\boxed{s'.cond(c) = \sigma'(id_c).}$

By definition of $E_c, \tau \vdash s \overset{\downarrow}{\to} s'$(Rule (1)), we have $s'.cond(c) = E_c(\tau, c)$.
By property of the $\texttt{Inject}_\downarrow$, the $\mathcal{H}$-VHDL falling edge, the stabilize relations and $id_c \in Ins(\Delta)$, we have $\sigma'(id_c) = E_p(\tau, \downarrow)(id_c)$.
Rewriting the goal with $s'.cond(c) = E_c(\tau, c)$ and $\sigma'(id_c) = E_p(\tau, \downarrow)(id_c)$: $\boxed{E_c(\tau, c) = E_p(\tau, \downarrow)(id_c)}$
By definition of $\gamma \vdash E_p \overset{env}{=} E_c$: $E_c(\tau, c) = E_p(\tau, \downarrow)(id_c)$.

□

C.4.4 Falling and action executions

Lemma 37 (Falling edge equal action executions)**.** *For all sitpn, $d, \gamma, \Delta, \sigma_e, E_c, E_p, \tau, s, s', \sigma, \sigma_i, \sigma_\downarrow$, σ' that verify the hypotheses of Definition 27, then $\forall a \in \mathcal{A}, id_a \in Outs(\Delta)$ s.t. $\gamma(a) = id_a$, $s'.ex(a) = \sigma'(id_a)$.*

Proof. Given an $a \in \mathcal{A}$ and an $id_a \in Outs(\Delta)$ s.t. $\gamma(a) = id_a$, let us show $\boxed{s'.ex(a) = \sigma'(id_a).}$

By property of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$(Rule (2)):

$$s'.ex(a) = \sum_{p \in marked(s.M)} \mathbb{A}(p,a) \tag{C.27}$$

By construction, the generated `action` process is a part of design d's behavior, i.e there exist an $sl \subseteq Sigs(\Delta)$ and an $ss_a \in ss$ s.t. $\texttt{ps}(\text{"}action\text{"}, \emptyset, sl, ss) \in d.cs$.
By construction id_a is only assigned in the body of the `action` process during the initialization or a falling edge phase.
Let $pls(a)$ be the set of actions associated to action a, i.e $pls(a) = \{p \in P \mid \mathbb{A}(p,a) = true\}$.
Then, depending on $pls(a)$, there are two cases of assignment of output port id_a:

- **CASE** $pls(a) = \emptyset$:
  By construction, $\texttt{id}_\texttt{a} \Leftarrow \texttt{false} \in ss_{a\downarrow}$ where $ss_{a\downarrow}$ is the part of the ''`action`'' process body executed during a falling edge phase.

  By property of the $\mathcal{H}$-VHDL falling edge relation, the stabilize relation and $\texttt{ps}(\text{"}action\text{"}, \emptyset, sl, ss_a) \in d.cs$, we can deduce $\sigma'(id_a) = false$.

  By property of $\sum_{p \in marked(s.M)} \mathbb{A}(p,a)$ and $pls(a) = \emptyset$, we can deduce $\sum_{p \in marked(s.M)} \mathbb{A}(p,a) =$ `false`.

  Rewriting the goal with (C.27), $\sigma'(id_a) = false$ and $\sum_{p \in marked(s.M)} \mathbb{A}(p,a) = \texttt{false}$, tautology.

- **CASE** $pls(a) \neq \emptyset$:
  By construction, $\texttt{id}_\texttt{a} \Leftarrow \texttt{id}_{\texttt{mp}_0} + \cdots + \texttt{id}_{\texttt{mp}_\texttt{n}} \in ss_{a\downarrow}$, where $id_{mp_i} \in Sigs(\Delta)$, $ss_{a\downarrow}$ is the part of the `action` process body executed during the falling edge phase, and $n = |pls(a)| - 1$.

  By property of the $\texttt{Inject}_\downarrow$, the $\mathcal{H}$-VHDL falling edge relation, the stabilize relation, and $\texttt{ps}(\text{"}action\text{"}, \emptyset, sl, ss) \in d.cs$:

  $$\sigma'(id_a) = \sigma(id_{mp_0}) + \cdots + \sigma(id_{mp_n}) \tag{C.28}$$

  Rewriting the goal with (C.27) and (C.28): $\boxed{\sum_{p \in marked(s.M)} \mathbb{A}(p,a) = \sigma(id_{mp_0}) + \cdots + \sigma(id_{mp_n}).}$

  Let us reason on the value of $\sigma(id_{mp_0}) + \cdots + \sigma(id_{mp_n})$; there are two cases:

  - **CASE** $\sigma(id_{mp_0}) + \cdots + \sigma(id_{mp_n}) = \texttt{true}$:
    Then, we can rewrite the goal as follows: $\boxed{\sum_{p \in marked(s.M)} \mathbb{A}(p,a) = \texttt{true}.}$

    To prove the above goal, let us show $\boxed{\exists p \in marked(s.M)\ s.t.\ \mathbb{A}(p,a) = \texttt{true}.}$

    From $\sigma(id_{mp_0}) + \cdots + \sigma(id_{mp_n}) = \texttt{true}$, we can deduce that $\exists id_{mp_i}\ s.t.\ \sigma(id_{mp_i}) = \texttt{true}$.
    Let us take an id_{mp_i} s.t. $\sigma(id_{mp_i}) = \texttt{true}$.
    By construction, there exist a $p \in pls(a)$, an $id_p \in Comps(\Delta)$, gm_p, ipm_p and opm_p such that:

* $\gamma(p) = id_p$
* $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$
* $<\texttt{marked} \Rightarrow \texttt{id}_{\texttt{mp}_i}> \in opm_p$

Let us take such a p, id_p, gm_p, ipm_p and opm_p.
By property of stable σ and $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$, we can deduce $\sigma(id_{mp_i}) = \sigma(id_p)("marked")$.
By property of stable σ, $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$, and through the examination of the `determine_marked` process defined in the place design architecture, we can deduce:

$$\sigma(id_p)("marked") = \sigma(id_p)("sm") > 0 \tag{C.29}$$

From $\sigma(id_{mp_i}) = \sigma(id_p)("marked")$, (C.29) and $\sigma(id_{mp_i}) = \texttt{true}$, we can deduce that $\sigma(id_p)("marked") = \texttt{true}$ and $(\sigma(id_p)("sm") > 0) = \texttt{true}$.

By property of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $s.M(p) = \sigma(id_p)("sm")$.
From $s.M(p) = \sigma(id_p)("sm")$ and $(\sigma(id_p)("sm") > 0) = \texttt{true}$, we can deduce $p \in marked(s.M)$, i.e $s.M(p) > 0$.

Let us use p to prove the goal: $\boxed{\mathbb{A}(p,a) = \texttt{true}.}$

By definition of $p \in pls(a)$, $\mathbb{A}(p,a) = \texttt{true}.$

– **CASE** $\sigma(id_{mp_0}) + \cdots + \sigma(id_{mp_n}) = \texttt{false}$:

Then, we can rewrite the goal as follows: $\boxed{\sum_{p \in marked(s.M)} \mathbb{A}(p,a) = \texttt{false}.}$

To prove the above goal, let us show $\boxed{\forall p \in marked(s.M)\ s.t.\ \mathbb{A}(p,a) = \texttt{false}.}$

Given a $p \in marked(s.M)$, let us show $\boxed{\mathbb{A}(p,a) = \texttt{false}.}$

Let us perform case analysis on $\mathbb{A}(p,a)$; there are 2 cases:

* **CASE** $\mathbb{A}(p,a) = \texttt{false}.$
* **CASE** $\mathbb{A}(p,a) = \texttt{true}$:
  By construction, there exist an $id_p \in Comps(\Delta)$, gm_{tp}, ipm_p, opm_p and $id_{mp_i} \in Sigs(\Delta)$ such that:
  · $\gamma(p) = id_p$
  · $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$
  · $<\texttt{marked} \Rightarrow \texttt{id}_{\texttt{mp}_i}> \in opm_p$
  Let us take such a id_p, gm_p, ipm_p, opm_p and id_{mp_i}.
  By property of stable σ, $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$, and $<\texttt{marked} \Rightarrow \texttt{id}_{\texttt{mp}_i}> \in opm_p$, we can deduce $\sigma(id_{mp_i}) = \sigma(id_p)("marked")$.
  By property of stable σ, $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$, and through the examination of the `determine_marked` process defined in the place design architecture, we can deduce:

$$\sigma(id_p)("marked") = (\sigma(id_p)("sm") > 0) \tag{C.30}$$

From $\sigma(id_{mp_0}) + \cdots + \sigma(id_{mp_n}) = \texttt{false}$, we can deduce $\sigma(id_{mp_i}) = \texttt{false}$.
From $\sigma(id_p)(\text{"}marked\text{"}) = \texttt{false}$, we can deduce $(\sigma(id_p)(\text{"}sm\text{"}) > 0) = \texttt{false}$.
By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $s.M(p) = \sigma(id_p)(\text{"}sm\text{"})$, and thus, we can deduce that $s.M(p) = 0$ (equivalent to $(s.M(p) > 0) = \texttt{false}$).
Contradicts $p \in marked(s.M)$ (i.e, $s.M(p) > 0$).

□

C.4.5 Falling edge and function executions

Lemma 38 (Falling edge equal function executions). *For all sitpn, d, γ, Δ, σ_e, E_c, E_p, τ, s, s', σ, σ_i, $\sigma_\downarrow$, σ' that verify the hypotheses of Definition 27, then $\forall f \in \mathcal{F}, id_f \in Outs(\Delta)$ s.t. $\gamma(f) = id_f$, $s'.ex(f) = \sigma'(id_f)$.*

Proof. Given an $f \in \mathcal{F}$ and an $id_f \in Outs(\Delta)$ s.t. $\gamma(f) = id_f$, let us show $\boxed{s'.ex(f) = \sigma'(id_f).}$

By property of $E_c, \tau \vdash s \overset{\downarrow}{\to} s'$, we can deduce $s.ex(f) = s'.ex(f)$.
By construction, id_f is an output port identifier of boolean type in the $\mathcal{H}$-VHDL design d assigned by the `function` process only during the initialization or during a rising edge phase.
By property of the $\mathcal{H}$-VHDL $\texttt{Inject}_\uparrow$, rising edge, stabilize relations, and the `function` process, we can deduce $\sigma(id_f) = \sigma'(id_f)$.
Rewriting the goal with $s.ex(f) = s'.ex(f)$ and $\sigma(id_f) = \sigma'(id_f)$, $\boxed{s.ex(f) = \sigma(id_f).}$
By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, $s.ex(f) = \sigma(id_f)$. □

C.4.6 Falling edge and firable transitions

Lemma 39 (Falling edge equal firable). *For all sitpn, d, γ, Δ, σ_e, E_c, E_p, τ, s, s', σ, σ_i, $\sigma_\downarrow$, σ' that verify the hypotheses of Definition 27, then $\forall t \in T, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, $t \in Firable(s') \Leftrightarrow \sigma'(id_t)(\text{"}s\_firable\text{"}) = \texttt{true}$.*

Proof. Given a $t \in T$ and $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, let us show that
$\boxed{t \in Firable(s') \Leftrightarrow \sigma'(id_t)(\text{"}s\_firable\text{"}) = \texttt{true}.}$

The proof is in two parts:

1. Assuming that $t \in Firable(s')$, let us show $\boxed{\sigma'(id_t)(\text{"}s\_firable\text{"}) = \texttt{true}.}$

   Appealing to Lemma 40: $\sigma'(id_t)(\text{"}s\_firable\text{"}) = \texttt{true}.$

2. Assuming that $\sigma'(id_t)(\text{"}s\_firable\text{"}) = \texttt{true}$, let us show $\boxed{t \in Firable(s').}$

   Appealing to Lemma 41: $t \in Firable(s')$.

□

Lemma 40 (Falling edge equal firable 1)**.** *For all sitpn, d, γ, Δ, σ_e, E_c, E_p, τ, s, s', σ, σ_i, $\sigma_\downarrow$, σ' that verify the hypotheses of Definition 27, then* $\forall t \in T, id_t \in Comps(\Delta)$ *s.t.* $\gamma(t) = id_t, t \in Firable(s') \Rightarrow \sigma'(id_t)(\text{"}s\_firable\text{"}) = \texttt{true}$.

Proof. Given a $t \in T$ and $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, and assuming that $t \in Firable(s')$, let us show $\boxed{\sigma'(id_t)(\text{"}s\_firable\text{"}) = \texttt{true}.}$

By construction and by definition of id_t, there exist gm_t, ipm_t, opm_t s.t. $\texttt{comp}(id_t, \text{"}transition\text{"}, gm_t, ipm_t, opm_t) \in d.cs$.
By property of the $\texttt{Inject}_\downarrow$ relation, the $\mathcal{H}$-VHDL falling edge relation, the stabilize relation, $\texttt{comp}(id_t, \text{"}transition\text{"}, gm_t, ipm_t, opm_t) \in d.cs$, and through the examination of the `firable` process defined in the transition design architecture, we can deduce:

$$\sigma'(id_t)(\text{"}sfa\text{"}) = \sigma(id_t)(\text{"}se\text{"}) \,.\, \sigma(id_t)(\text{"}scc\text{"}) \,.\, \texttt{checktc}(\Delta(id_t), \sigma(id_t)) \tag{C.31}$$

Term $\texttt{checktc}(\Delta(id_t), \sigma(id_t))$ is defined as follows:

$$\begin{aligned}\texttt{checktc}(\Delta(id_t), \sigma(id_t)) = \Big(&\texttt{not}\ \sigma(id_t)(\text{"}srtc\text{"}) \,. \\ &\big[(\Delta(id_t)(\text{"}tt\text{"}) = \texttt{TEMP\_A\_B} \,.\, (\sigma(id_t)(\text{"}stc\text{"}) \geq \sigma(id_t)(\text{"}A\text{"}) - 1) \\ &\qquad .\, (\sigma(id_t)(\text{"}stc\text{"}) \leq \sigma(id_t)(\text{"}B\text{"}) - 1)\big) \\ &+ (\Delta(id_t)(\text{"}tt\text{"}) = \texttt{TEMP\_A\_A} \,.\, (\sigma(id_t)(\text{"}stc\text{"}) = \sigma(id_t)(\text{"}A\text{"}) - 1)) \\ &+ (\Delta(id_t)(\text{"}tt\text{"}) = \texttt{TEMP\_A\_INF} \,.\, (\sigma(id_t)(\text{"}stc\text{"}) \geq \sigma(id_t)(\text{"}A\text{"}) - 1))\big]\Big) \\ &+ \big(\sigma(id_t)(\text{"}srtc\text{"}) \,.\, \Delta(id_t)(\text{"}tt\text{"}) \neq \texttt{NOT\_TEMP} \,.\, \sigma(id_t)(\text{"}A\text{"}) = 1\big) \\ &+ \Delta(id_t)(\text{"}tt\text{"}) = \texttt{NOT\_TEMP}\end{aligned} \tag{C.32}$$

Rewriting the goal with (C.31): $\boxed{\sigma(id_t)(\text{"}se\text{"}) \,.\, \sigma(id_t)(\text{"}scc\text{"}) \,.\, \texttt{checktc}(\Delta(id_t), \sigma(id_t)) = \texttt{true}.}$
Then, there are three points to prove:

1. $\boxed{\sigma(id_t)(\text{"}se\text{"}) = \texttt{true}}$:

   From $t \in Firable(s')$, we can deduce $t \in Sens(s'.M)$. By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$, we have $s.M = s'.M$, and thus, we can deduce $t \in Sens(s.M)$.

   By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we know that $t \in Sens(s.M)$ implies $\sigma(id_t)(\text{"}se\text{"}) = \texttt{true}$.

2. $\boxed{\sigma(id_t)(\text{"}scc\text{"}) = \texttt{true}}$:

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$:

$$\sigma(id_t)(\text{"scc"}) = \prod_{c \in conds(t)} \begin{cases} E_c(\tau, c) & if\ \mathbb{C}(t,c) = 1 \\ \texttt{not}(E_c(\tau, c)) & if\ \mathbb{C}(t,c) = -1 \end{cases} \tag{C.33}$$

where $conds(t) = \{c \in \mathcal{C} \mid \mathbb{C}(t,c) = 1 \vee \mathbb{C}(t,c) = -1\}$.

Rewriting the goal with (C.33): $\boxed{\prod_{c \in conds(t)} \begin{cases} E_c(\tau, c) & if\ \mathbb{C}(t,c) = 1 \\ \texttt{not}(E_c(\tau, c)) & if\ \mathbb{C}(t,c) = -1 \end{cases} = \texttt{true}.}$

To ease the reading, let us define $f(c) = \begin{cases} E_c(\tau, c) & if\ \mathbb{C}(t,c) = 1 \\ \texttt{not}(E_c(\tau, c)) & if\ \mathbb{C}(t,c) = -1 \end{cases}$.

Let us reason by induction on the left term of the goal:

- **BASE CASE**: $\texttt{true} = \texttt{true}.$
- **INDUCTION CASE**:

  $$\prod_{c' \in conds(t) \setminus \{c\}} f(c') = \texttt{true}$$

  $$\boxed{f(c) \cdot \prod_{c' \in conds(t) \setminus \{c\}} f(c') = \texttt{true}.}$$

  Rewriting the goal with the induction hypothesis, simplifying the goal, and unfolding the definition of $f(c)$: $\boxed{\begin{cases} E_c(\tau, c) & if\ \mathbb{C}(t,c) = 1 \\ \texttt{not}(E_c(\tau, c)) & if\ \mathbb{C}(t,c) = -1 \end{cases} = \texttt{true}.}$

  As $c \in conds(t)$, let us perform case analysis on $\mathbb{C}(t,c) = 1 \vee \mathbb{C}(t,c) = -1$:

  (a) **CASE** $\mathbb{C}(t,c) = 1$: $\boxed{E_c(\tau, c) = \texttt{true}.}$

  By definition of $t \in Firable(s')$, we can deduce that $s'.cond(c) = \texttt{true}$. By definition of $E_c, \tau \vdash s \overset{\downarrow}{\to} s'$ (Rule (1)), we have $s'.cond(c) = E_c(\tau, c)$. Thus, $E_c(\tau, c) = \texttt{true}.$

  (b) $\mathbb{C}(t,c) = -1$: $\boxed{\texttt{not}\ E_c(\tau, c) = \texttt{true}.}$

  By definition of $t \in Firable(s')$, we can deduce that $s'.cond(c) = \texttt{false}$. By definition of $E_c, \tau \vdash s \overset{\downarrow}{\to} s'$ (Rule (1)), we have $s'.cond(c) = E_c(\tau, c)$. Thus, $\texttt{not}\ E_c(\tau, c) = \texttt{true}.$

3. $\boxed{\texttt{checktc}(\Delta(id_t), \sigma(id_t)) = \texttt{true}}$:

By definition of $t \in Firable(s')$, we have $t \notin T_i \vee s'.I(t) \in I_s(t)$. Let us perform case analysis on $t \notin T_i \vee s'.I(t) \in I_s(t)$:

(a) **CASE** $t \notin T_i$: $\boxed{\texttt{checktc}(\Delta(id_t), \sigma(id_t)) = \texttt{true}}$

By construction, $<\texttt{transition\_type} \Rightarrow \texttt{NOT\_TEMP}> \in gm_t$, and by property of the elaboration relation, we have $\Delta(id_t)(''tt'') = \texttt{NOT\_TEMP}$.
From $\Delta(id_t)(''tt'') = \texttt{NOT\_TEMP}$, and by definition of $\texttt{checktc}(\Delta(id_t), \sigma(id_t))$, we can deduce $\texttt{checktc}(\Delta(id_t), \sigma(id_t)) = \texttt{true}$.

(b) **CASE** $s'.I(t) \in I_s(t)$: $\boxed{\texttt{checktc}(\Delta(id_t), \sigma(id_t)) = \texttt{true}}$

From $s'.I(t) \in I_s(t)$, we can deduce that $t \in T_i$. Thus, by construction, there exists $tt \in \{\texttt{TEMP\_A\_B}, \texttt{TEMP\_A\_A}, \texttt{TEMP\_A\_INF}\}$ s.t. $<\texttt{transition\_type} \Rightarrow tt> \in gm_t$. By property of the elaboration relation, we have $\Delta(id_t)(''tt'') = tt$, and thus, we know $\Delta(id_t)(''tt'') \neq \texttt{NOT\_TEMP}$. Therefore, we can simplfy the term $\texttt{checktc}(\Delta(id_t), \sigma(id_t))$ as follows:

$$\begin{aligned}\texttt{checktc}(\Delta(id_t), \sigma(id_t)) = \Big(&\texttt{not}\ \sigma(id_t)(''srtc'')\ .\\ &\big[(\Delta(id_t)(''tt'') = \texttt{TEMP\_A\_B}\ .\ (\sigma(id_t)(''stc'') \geq \sigma(id_t)(''A'') - 1)\\ &\qquad .\ (\sigma(id_t)(''stc'') \leq \sigma(id_t)(''B'') - 1))\\ &+(\Delta(id_t)(''tt'') = \texttt{TEMP\_A\_A}\ .\\ &\quad (\sigma(id_t)(''stc'') = \sigma(id_t)(''A'') - 1))\\ &+(\Delta(id_t)(''tt'') = \texttt{TEMP\_A\_INF}\ .\\ &\quad (\sigma(id_t)(''stc'') \geq \sigma(id_t)(''A'') - 1))\big]\Big)\\ &+ \big(\sigma(id_t)(''srtc'')\ .\ \sigma(id_t)(''A'') = 1\big)\end{aligned} \tag{C.34}$$

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $s.reset_t(t) = \sigma(id_t)(''srtc'')$.
Let us perform case analysis on the value $s.reset_t(t)$:

i. **CASE** $s.reset_t(t) = \texttt{true}$: $\boxed{\texttt{checktc}(\Delta(id_t), \sigma(id_t)) = \texttt{true}}$

From $s.reset_t(t) = \sigma(id_t)(''srtc'')$, we can deduce that $\sigma(id_t)(''srtc'') = \texttt{true}$.
From $\sigma(id_t)(''srtc'') = \texttt{true}$, we can simplify the term $\texttt{checktc}(\Delta(id_t), \sigma(id_t))$ as follows:

$$\texttt{checktc}(\Delta(id_t), \sigma(id_t)) = \big(\sigma(id_t)(''A'') = 1\big) \tag{C.35}$$

Rewriting the goal with (C.35), and simplifying the goal: $\boxed{\sigma(id_t)(''A'') = 1.}$

By definition of $E_c, \tau \vdash s \overset{\downarrow}{\rightarrow} s'$ (Rule (3)), from $t \in Sens(s.M)$ and $s.reset_t(t) = \texttt{true}$, we can deduce $s'.I(t) = 1$. We know that $s'.I(t) \in I_s(t)$, and thus, we have $1 \in I_s(t)$. By definition of $1 \in I_s(t)$, there exist an $a \in \mathbb{N}^*$ and a $ni \in \mathbb{N}^* \sqcup \{\infty\}$ s.t. $I_s(t) = [a, ni]$ and $1 \in [a, ni]$.

By definition of $1 \in [a, ni]$, we have $a \leq 1$, and since $a \in \mathbb{N}^*$, we can deduce $a = 1$.
By construction, $<$`time_A_value` $\Rightarrow a>$ $\in ipm_t$, and by property of stable σ, we have $\sigma(id_t)(''A'') = a = 1$.

ii. **CASE** $s.reset_t(t) =$ `false`: $\boxed{\texttt{checktc}(\Delta(id_t), \sigma(id_t)) = \texttt{true}}$

From $s.reset_t(t) = \sigma(id_t)(''srtc'')$, we can deduce $\sigma(id_t)(''srtc'') =$ `false`.
From $\sigma(id_t)(''srtc'') =$ `false`, we can simplify the term $\texttt{checktc}(\Delta(id_t), \sigma(id_t))$ as follows:

$$\begin{array}{c} \texttt{checktc}(\Delta(id_t), \sigma(id_t)) \\ = \\ \big(\Delta(id_t)(''tt'') = \texttt{TEMP\_A\_B} \;\; . (\sigma(id_t)(''stc'') \geq \sigma(id_t)(''A'') - 1) \\ \qquad\qquad\qquad\qquad\quad . (\sigma(id_t)(''stc'') \leq \sigma(id_t)(''B'') - 1)\big) \\ +(\Delta(id_t)(''tt'') = \texttt{TEMP\_A\_A} \,.\, (\sigma(id_t)(''stc'') = \sigma(id_t)(''A'') - 1)) \\ +(\Delta(id_t)(''tt'') = \texttt{TEMP\_A\_INF} \,.\, (\sigma(id_t)(''stc'') \geq \sigma(id_t)(''A'') - 1)) \end{array} \tag{C.36}$$

Let us perform case analysis on $I_s(t)$; there are two cases:

- **CASE** $I_s(t) = [a, b]$ where $a, b \in \mathbb{N}^*$; then, either $a = b$ or $a \neq b$:
  - **CASE** $a = b$:
    Then, we have $I_s(t) = [a, a]$, and by construction $<$`transition_type`$\Rightarrow$ `TEMP_A_A`$>$ $\in gm_t$. By property of the elaboration relation, we have $\Delta(id_t)(''tt'') =$`TEMP_A_A`; thus we can simplify the `checktc` term as follows:

$$\texttt{checktc}(\Delta(id_t), \sigma(id_t)) = (\sigma(id_t)(''stc'') = \sigma(id_t)(''A'') - 1) \tag{C.37}$$

    Rewriting the goal with (C.37), and simplifying the goal:
    $\boxed{\sigma(id_t)(''stc'') = \sigma(id_t)(''A'') - 1.}$

    From $s'.I(t) \in [a, a]$, we can deduce that $s'.I(t) = a$. Let us perform case analysis on $s.I(t) < upper(I_s(t))$ or $s.I(t) \geq upper(I_s(t))$:
    - **CASE** $s.I(t) < upper(I_s(t))$:
      By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $s.I(t) = \sigma(id_t)(''stc'')$. By definition of $E_c, \tau \vdash s \overset{\downarrow}{\rightarrow} s'$ (Rule (4)), we have $s'.I(t) = s.I(t) + 1$. From $s'.I(t) = a$ and $s'.I(t) = s.I(t) + 1$, we can deduce $a - 1 = s.I(t)$.
      By construction, $<$`time_A_value` $\Rightarrow a>$ $\in ipm_t$, and by property of stable σ, we have $\sigma(id_t)(''A'') = a$.
      Rewriting the goal with $\sigma(id_t)(''A'') = a$, $s.I(t) = \sigma(id_t)(''stc'')$, and $a - 1 = s.I(t)$: tautology.
    - **CASE** $s.I(t) \geq upper(I_s(t))$:
      In the case where $s.I(t) > upper(I_s(t))$, then $s.I(t) > a$. By definition of $E_c, \tau \vdash s \overset{\downarrow}{\rightarrow} s'$ (Rule (5)), we have $s.I(t) = s'.I(t) = a$. Then, $a > a$ is a contradiction.

In the case where $s.I(t) = upper(I_s(t))$, then $s.I(t) = a$. By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$ (Rule (4)), we have $s'.I(t) = s.I(t) + 1$. Then, we have $s'.I(t) = a$ and $s'.I(t) = a + 1$. Then, $a = a + 1$ is a contradiction.

– **CASE** $a \neq b$: $\boxed{\texttt{checktc}(\Delta(id_t), \sigma(id_t)) = \texttt{true}}$
Then, we have $I_s(t) = [a, b]$, and by construction <transition_type⇒ TEMP_A_B> $\in gm_t$. By property of the elaboration relation, we have $\Delta(id_t)(''tt'')$ =TEMP_A_B; thus we can simplify the term checktc as follows:

$$\begin{gathered}\texttt{checktc}(\Delta(id_t), \sigma(id_t)) \\ = \\ (\sigma(id_t)(''stc'') \geq \sigma(id_t)(''A'') - 1) \,.\, (\sigma(id_t)(''stc'') \leq \sigma(id_t)(''B'') - 1)\end{gathered} \tag{C.38}$$

Rewriting the goal with (C.38), and simplifying the goal:
$\boxed{(\sigma(id_t)(''stc'') \geq \sigma(id_t)(''A'') - 1) \wedge (\sigma(id_t)(''stc'') \leq \sigma(id_t)(''B'') - 1).}$

Let us perform case analysis on $s.I(t) < upper(I_s(t))$ or $s.I(t) \geq upper(I_s(t))$:

∗ **CASE** $s.I(t) < upper(I_s(t))$:

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $s.I(t) = \sigma(id_t)(''stc'')$. By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$(Rule (4)), we have $s'.I(t) = s.I(t) + 1$. By definition of $s'.I(t) \in [a, b]$:
$\Rightarrow a \leq s'.I(t) \leq b$.
$\Rightarrow a \leq s'.I(t) \wedge s'.I(t) \leq b$
$\Rightarrow a \leq s.I(t) + 1 \wedge s.I(t) + 1 \leq b$
$\Rightarrow a - 1 \leq s.I(t) \wedge s.I(t) \leq b - 1$
By construction, <time_A_value ⇒a> $\in ipm_t$ and <time_B_value ⇒b> $\in ipm_t$, and by property of stable σ, we have $\sigma(id_t)(''A'') = a$ and $\sigma(id_t)(''B'') = b$.
Rewriting the goal with $\sigma(id_t)(''A'') = a$, $\sigma(id_t)(''B'') = b$ and $s.I(t) = \sigma(id_t)(''stc'')$: $a - 1 \leq s.I(t) \wedge s.I(t) \leq b - 1$.

∗ **CASE** $s.I(t) \geq upper(I_s(t))$:

In the case where $s.I(t) > upper(I_s(t))$, then $s.I(t) > b$. By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$ (Rule (5)), we have $s.I(t) = s'.I(t) = b$. Then, $b > b$ is a contradiction.

In the case where $s.I(t) = upper(I_s(t))$, then $s.I(t) = b$. By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$ (Rule (4)), we have $s'.I(t) = s.I(t) + 1$.
By definition of $s'.I(t) \in [a, b]$, we have $s'.I(t) \leq b$:
$\Rightarrow s.I(t) + 1 \leq b$
$\Rightarrow$ $b + 1 \leq b$ is contradiction.

• **CASE** $I_s(t) = [a, \infty]$ where $a \in \mathbb{N}^*$: $\boxed{\texttt{checktc}(\Delta(id_t), \sigma(id_t)) = \texttt{true}}$

By construction <`transition_type` ⇒ `TEMP_A_INF`> ∈ gm_t. By property of the elaboration relation, we have $\Delta(id_t)(''tt'') =$ `TEMP_A_INF`; thus we can simplify the term `checktc` as follows:

$$\texttt{checktc}(\Delta(id_t), \sigma(id_t)) = (\sigma(id_t)(''stc'') \geq \sigma(id_t)(''A'') - 1)) \tag{C.39}$$

Rewriting the goal with (C.39), and simplifying the goal:
$\boxed{\sigma(id_t)(''stc'') \geq \sigma(id_t)(''A'') - 1.}$

From $s'.I(t) \in [a, \infty]$, we can deduce $a \leq s'.I(t)$. Then, let us perform case analysis on $s.I(t) \leq lower(I_s(t))$ or $s.I(t) > lower(I_s(t))$:

– **CASE** $s.I(t) \leq lower(I_s(t))$:

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $s.I(t) = \sigma(id_t)(''stc'')$.
By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$ (Rule (4)), we have $s'.I(t) = s.I(t) + 1$:
$\Rightarrow s'.I(t) \geq a$
$\Rightarrow s.I(t) + 1 \geq a$
$\Rightarrow s.I(t) \geq a - 1$
By construction, <`time_A_value` ⇒a> ∈ ipm_t, and by property of stable σ, we have $\sigma(id_t)(''A'') = a$.
Rewriting the goal with $\sigma(id_t)(''A'') = a$ and $s.I(t) = \sigma(id_t)(''stc'')$:
$s.I(t) \geq a - 1.$

– **CASE** $s.I(t) > lower(I_s(t))$:

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $\sigma(id_t)(''stc'') = lower(I_s(t)) = a$.
By construction, <`time_A_value` ⇒a> ∈ ipm_t, and by property of stable σ, we have $\sigma(id_t)(''A'') = a$.
Rewriting the goal with $\sigma(id_t)(''stc'') = a$ and $\sigma(id_t)(''A'') = a$: $a \geq a - 1.$

□

Lemma 41 (Falling Edge Equal Firable 2). *For all sitpn, d, γ, Δ, σ_e, E_c, E_p, τ, s, s', σ, σ_i, $\sigma_\downarrow$, σ' that verify the hypotheses of Definition 27, then $\forall t \in T, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, $\sigma'(id_t)(''s\_firable'') =$ `true` $\Rightarrow t \in Firable(s')$.*

Proof. Given a $t \in T$ and $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, and assuming that $\sigma'(id_t)(''s\_firable'') =$ `true`, let us show $\boxed{t \in Firable(s').}$

By construction and by definition of id_t, there exist gm_t, ipm_t, opm_t s.t. `comp`$(id_t, ''transition'', gm_t, ipm_t, opm_t) \in d.cs$.
By property of the `Inject`$_\downarrow$ relation, the $\mathcal{H}$-VHDL falling edge relation, the stabilize relation, `comp`$(id_t, ''transition'', gm_t, ipm_t, opm_t) \in d.cs$, and through the examination of the `firable` process defined in the transition design architecture, we can deduce:

$$\sigma'(id_t)(''sfa'') = \sigma(id_t)(''se'') \,.\, \sigma(id_t)(''scc'') \,.\, \texttt{checktc}(\Delta(id_t), \sigma(id_t)) = \texttt{true} \tag{C.40}$$

From (C.40), we can deduce:

$$\sigma(id_t)(\text{"se"}) = \texttt{true} \tag{C.41}$$
$$\sigma(id_t)(\text{"scc"}) = \texttt{true} \tag{C.42}$$
$$\texttt{checktc}(\Delta(id_t), \sigma(id_t)) = \texttt{true} \tag{C.43}$$

Term $\texttt{checktc}(\Delta(id_t), \sigma(id_t))$ as the same definition as in Lemma Falling edge equal firable 1. By definition of $t \in Firable(s')$, there are three points to prove:

1. $\boxed{t \in Sens(s'.M)}$
2. $\boxed{\forall c \in \mathcal{C},\ \mathbb{C}(t,c) = 1 \Rightarrow s'.cond(c) = \texttt{true} \text{ and } \mathbb{C}(t,c) = -1 \Rightarrow s'.cond(c) = \texttt{false}}$
3. $\boxed{t \notin T_i \vee s'.I(t) \in I_s(t)}$

Let us prove these three points:

1. $\boxed{t \in Sens(s'.M)}$:

   By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$, we have $s.M = s'.M$. Rewriting the goal with $s.M = s'.M$: $\boxed{t \in Sens(s.M).}$

   By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $\sigma(id_t)(\text{"se"}) = \texttt{true} \Leftrightarrow t \in Sens(s.M)$.

   From $\sigma(id_t)(\text{"se"}) = \texttt{true}$, we can deduce: $t \in Sens(s.M)$.

2. $\boxed{\forall c \in \mathcal{C},\ \mathbb{C}(t,c) = 1 \Rightarrow s'.cond(c) = \texttt{true} \text{ and } \mathbb{C}(t,c) = -1 \Rightarrow s'.cond(c) = \texttt{false}}$

   Given a $c \in \mathcal{C}$, there are two points to prove:

   (a) $\boxed{\mathbb{C}(t,c) = 1 \Rightarrow s'.cond(c) = \texttt{true}.}$

   (b) $\boxed{\mathbb{C}(t,c) = -1 \Rightarrow s'.cond(c) = \texttt{false}.}$

   Let us prove these two points:

   (a) Assuming that $\mathbb{C}(t,c) = 1$, let us show $\boxed{s'.cond(c) = \texttt{true}.}$

   By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have:

$$\sigma(id_t)(\text{"scc"}) = \prod_{c' \in conds(t)} \begin{cases} E_c(\tau, c') & \text{if } \mathbb{C}(t,c') = 1 \\ \texttt{not}(E_c(\tau, c')) & \text{if } \mathbb{C}(t,c') = -1 \end{cases} = \texttt{true} \tag{C.44}$$

   where $conds(t) = \{c_i \in \mathcal{C} \mid \mathbb{C}(t,c_i) = 1 \vee \mathbb{C}(t,c_i) = -1\}$.

From $\mathbb{C}(t,c) = 1$, we can deduce $c \in conds(t)$. By definition of the product expression, we have:

$$E_c(\tau,c)\,.\prod_{c'\in conds(t)\setminus\{c\}}\begin{cases}E_c(\tau,c') & if\ \mathbb{C}(t,c')=1\\ \texttt{not}(E_c(\tau,c')) & if\ \mathbb{C}(t,c')=-1\end{cases} = \texttt{true} \tag{C.45}$$

From (C.45), we can deduce that $E_c(\tau,c) = \texttt{true}$.

By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$ (Rule (1)), we have $s'.cond(c) = E_c(\tau,c)$.
Rewriting the goal with $s'.cond(c) = E_c(\tau,c)$ and $E_c(\tau,c) = \texttt{true}$: tautology.

(b) Assuming that $\mathbb{C}(t,c) = -1$, let us show $\boxed{s'.cond(c) = \texttt{false}.}$

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have:

$$\sigma(id_t)(\text{"}scc\text{"}) = \prod_{c'\in conds(t)}\begin{cases}E_c(\tau,c') & if\ \mathbb{C}(t,c')=1\\ \texttt{not}(E_c(\tau,c')) & if\ \mathbb{C}(t,c')=-1\end{cases} = \texttt{true} \tag{C.46}$$

where $conds(t) = \{c' \in \mathcal{C} \mid \mathbb{C}(t,c') = 1 \vee \mathbb{C}(t,c') = -1\}$.
From $\mathbb{C}(t,c) = -1$, we can deduce $c \in conds(t)$. By definition of the product expression, we have:

$$\texttt{not}\ E_c(\tau,c)\,.\prod_{c'\in conds(t)\setminus\{c\}}\begin{cases}E_c(\tau,c') & if\ \mathbb{C}(t,c')=1\\ \texttt{not}(E_c(\tau,c')) & if\ \mathbb{C}(t,c')=-1\end{cases} = \texttt{true} \tag{C.47}$$

From (C.47), we can deduce that $E_c(\tau,c) = \texttt{false}$.

By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$ (Rule (1)), we have $s'.cond(c) = E_c(\tau,c)$.
Rewriting the goal with $s'.cond(c) = E_c(\tau,c)$ and $E_c(\tau,c) = \texttt{false}$: tautology.

3. $\boxed{t \notin T_i \vee s'.I(t) \in I_s(t)}$

Reasoning on $\texttt{checktc}(\Delta(id_t), \sigma(id_t)) = \texttt{true}$, there are 3 cases:

(a) $\left(\texttt{not}\ \sigma(id_t)(\text{"}srtc\text{"})\,.\,[\ldots]\right) = \texttt{true}$[1]

(b) $\left(\sigma(id_t)(\text{"}srtc\text{"})\,.\,\Delta(id_t)(\text{"}tt\text{"}) \neq \texttt{NOT\_TEMP}\,.\,\sigma(id_t)(\text{"}A\text{"}) = 1\right) = \texttt{true}$

(c) $\left(\Delta(id_t)(\text{"}tt\text{"}) = \texttt{NOT\_TEMP}\right) = \texttt{true}$

(a) **CASE** $\left(\texttt{not}\ \sigma(id_t)(\text{"}srtc\text{"})\,.\,[\ldots]\right) = \texttt{true}$:
Then, we can deduce $\texttt{not}\ \sigma(id_t)(\text{"}srtc\text{"}) = \texttt{true}$ and $[\ldots] = \texttt{true}$. From $\texttt{not}\ \sigma(id_t)(\text{"}srtc\text{"}) = \texttt{true}$, we can deduce $\sigma(id_t)(\text{"}srtc\text{"}) = \texttt{false}$, and from $[\ldots] = \texttt{true}$, we have three other cases:

[1] See equation (C.32) for the full definition.

i. **CASE** $\big(\Delta(id_t)(''tt'') = \texttt{TEMP\_A\_B} \,.\, (\sigma(id_t)(''stc'') \geq \sigma(id_t)(''A'') - 1) \,.\, (\sigma(id_t)(''stc'') \leq \sigma(id_t)(''B'') - 1)\big) = \texttt{true}$

ii. **CASE** $(\Delta(id_t)(''tt'') = \texttt{TEMP\_A\_A} \,.\, (\sigma(id_t)(''stc'') = \sigma(id_t)(''A'') - 1)) = \texttt{true}$

iii. **CASE** $(\Delta(id_t)(''tt'') = \texttt{TEMP\_A\_INF} \,.\, (\sigma(id_t)(''stc'') \geq \sigma(id_t)(''A'') - 1)) = \texttt{true}$

Let us prove the goal is these three contexts:

i. **CASE** $\big(\Delta(id_t)(''tt'') = \texttt{TEMP\_A\_B} \,.\, (\sigma(id_t)(''stc'') \geq \sigma(id_t)(''A'') - 1) \,.\, (\sigma(id_t)(''stc'') \leq \sigma(id_t)(''B'') - 1)\big) = \texttt{true}$:
Then, converting boolean equalities into intuitionistic predicates, we have:
- $\Delta(id_t)(''tt'') = \texttt{TEMP\_A\_B}$
- $\sigma(id_t)(''stc'') \geq \sigma(id_t)(''A'') - 1$
- $\sigma(id_t)(''stc'') \leq \sigma(id_t)(''B'') - 1$

By property of the elaboration relation, and $\Delta(id_t)(''tt'') = \texttt{TEMP\_A\_B}$, there exist $a, b \in \mathbb{N}^*$ s.t. $I_s(t) = [a, b]$. Let us take such an a and b. Then, let us show $\boxed{s'.I(t) \in I_s(t).}$

Rewriting the goal with $I_s(t) = [a, b]$: $\boxed{s'.I(t) \in [a, b].}$
By construction, <`time_A_value` $\Rightarrow a>$ and <`time_B_value` $\Rightarrow b>$, and by property of stable σ, we have $\sigma(id_t)(''A'') = a$ and $\sigma(id_t)(''B'') = b$.
Rewriting the goal with $\sigma(id_t)(''A'') = a$ and $\sigma(id_t)(''B'') = b$, and by definition of $\in$: $\boxed{\sigma(id_t)(''A'') \leq s'.I(t) \leq \sigma(id_t)(''B'').}$
Now, let us perform case analysis on $s.I(t) \leq upper(I_s(t))$ or $s.I(t) > upper(I_s(t))$:
- **CASE** $s.I(t) \leq upper(I_s(t))$:

  By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $s.I(t) = \sigma(id_t)(''stc'')$.
  From $\sigma(id_t)(''se'') = \texttt{true}$, we can deduce $t \in Sens(s.M)$, and from $\sigma(id_t)(''srtc'') = \texttt{false}$, we can deduce $s.reset_t(t) = \texttt{false}$. Then, by definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$ (Rule (4)), we have $s'.I(t) = s.I(t) + 1$.
  $\Rightarrow$ $\boxed{\sigma(id_t)(''A'') \leq s.I(t) + 1 \leq \sigma(id_t)(''B'')}$ (by $s'.I(t) = s.I(t) + 1$)
  $\Rightarrow$ $\boxed{\sigma(id_t)(''A'') \leq \sigma(id_t)(''stc'') + 1 \leq \sigma(id_t)(''B'')}$ (by $s.I(t) = \sigma(id_t)(''stc'')$)
  $\Rightarrow$ $\boxed{\sigma(id_t)(''A'') - 1 \leq \sigma(id_t)(''stc'') \leq \sigma(id_t)(''B'') - 1}$
  We assumed $\sigma(id_t)(''stc'') \geq \sigma(id_t)(''A'') - 1$ and $\sigma(id_t)(''stc'') \leq \sigma(id_t)(''B'') - 1$, and thus we can deduce: $\sigma(id_t)(''A'') - 1 \leq \sigma(id_t)(''stc'') \leq \sigma(id_t)(''B'') - 1$
- **CASE** $s.I(t) > upper(I_s(t))$:

  By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $\sigma(id_t)(''stc'') = upper(I_s(t)) = b$.
  Then, from $\sigma(id_t)(''stc'') \leq \sigma(id_t)(''B'') - 1$, $\sigma(id_t)(''stc'') = upper(I_s(t)) = b$ and $\sigma(id_t)(''B'') = b$, we can deduce the following contradiction:
  $\sigma(id_t)(''B'') \leq \sigma(id_t)(''B'') - 1.$

ii. $(\Delta(id_t)(''tt'') = \texttt{TEMP\_A\_A} \,.\, (\sigma(id_t)(''stc'') = \sigma(id_t)(''A'') - 1)) = \texttt{true}$:
Then, converting boolean equalities into intuitionistic predicates, we have:

- $\Delta(id_t)(''tt'') = \texttt{TEMP\_A\_A}$
- $\sigma(id_t)(''stc'') = \sigma(id_t)(''A'') - 1$

By property of the elaboration relation, and $\Delta(id_t)(''tt'') = \texttt{TEMP\_A\_A}$, there exist $a \in \mathbb{N}^*$ s.t. $I_s(t) = [a, a]$. Let us take such an a. Then, let us show $\boxed{s'.I(t) \in I_s(t).}$

Rewriting the goal with $I_s(t) = [a, a]$: $\boxed{s'.I(t) \in [a, a].}$

By construction, $<\texttt{time\_A\_value} \Rightarrow a>$, and by property of stable σ, we have $\sigma(id_t)(''A'') = a$.

Rewriting the goal with $\sigma(id_t)(''A'') = a$, unfolding the definition of $\in$, and simplifying the goal: $\boxed{s'.I(t) = \sigma(id_t)(''A'').}$

Now, let us perform case analysis on $s.I(t) \leq upper(I_s(t))$ or $s.I(t) > upper(I_s(t))$:

- **CASE** $s.I(t) \leq upper(I_s(t))$:

  By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $s.I(t) = \sigma(id_t)(''stc'')$.
  From $\sigma(id_t)(''se'') = \texttt{true}$, we can deduce $t \in Sens(s.M)$, and from $\sigma(id_t)(''srtc'') = \texttt{false}$, we can deduce $s.reset_t(t) = \texttt{false}$. Then, by definition of $E_c, \tau \vdash s \overset{\downarrow}{\rightarrow} s'$ (Rule (4)), we have $s'.I(t) = s.I(t) + 1$.
  $\Rightarrow$ $\boxed{s.I(t) + 1 = \sigma(id_t)(''A'')}$ (by $s'.I(t) = s.I(t) + 1$)
  $\Rightarrow$ $\boxed{\sigma(id_t)(''stc'') + 1 = \sigma(id_t)(''A'')}$ (by $s.I(t) = \sigma(id_t)(''stc'')$)
  $\Rightarrow$ $\sigma(id_t)(''stc'') = \sigma(id_t)(''A'') - 1$ (assumption)

- **CASE** $s.I(t) > upper(I_s(t))$:

  By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $\sigma(id_t)(''stc'') = upper(I_s(t)) = a$.
  Then, from $\sigma(id_t)(''stc'') = \sigma(id_t)(''A'') - 1$, $\sigma(id_t)(''stc'') = upper(I_s(t)) = a$, $\sigma(id_t)(''A'') = a$, and $a \in \mathbb{N}^*$, we can derive the following contradiction:
  $\sigma(id_t)(''A'') = \sigma(id_t)(''A'') - 1.$

iii. $(\Delta(id_t)(''tt'') = \texttt{TEMP\_A\_INF} \,.\, (\sigma(id_t)(''stc'') \geq \sigma(id_t)(''A'') - 1)) = \texttt{true}$:
Then, converting boolean equalities into intuitionistic predicates, we have:

- $\Delta(id_t)(''tt'') = \texttt{TEMP\_A\_INF}$
- $\sigma(id_t)(''stc'') \geq \sigma(id_t)(''A'') - 1$

By property of the elaboration relation, and $\Delta(id_t)(''tt'') = \texttt{TEMP\_A\_INF}$, there exist $a \in \mathbb{N}^*$ s.t. $I_s(t) = [a, \infty]$. Let us take such an a. Then, let us show $\boxed{s'.I(t) \in I_s(t).}$

Rewriting the goal with $I_s(t) = [a, \infty]$: $\boxed{s'.I(t) \in [a, \infty].}$

By construction, $<\texttt{time\_A\_value} \Rightarrow a>$, and by property of stable σ, we have $\sigma(id_t)(''A'') = a$.

Rewriting the goal with $\sigma(id_t)(''A'') = a$, unfolding the definition of $\in$, and simplifying the goal: $\boxed{\sigma(id_t)(''A'') \leq s'.I(t).}$

Now, let us perform case analysis on $s.I(t) \leq lower(I_s(t))$ or $s.I(t) > lower(I_s(t))$:

- **CASE** $s.I(t) \leq lower(I_s(t))$:

  By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $s.I(t) = \sigma(id_t)(''stc'')$.

From $\sigma(id_t)(\text{"}se\text{"}) = \texttt{true}$, we can deduce $t \in Sens(s.M)$, and from $\sigma(id_t)(\text{"}srtc\text{"}) = \texttt{false}$, we can deduce $s.reset_t(t) = \texttt{false}$. Then, by definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$ (Rule (4)), we have $s'.I(t) = s.I(t) + 1$.

$\Rightarrow$ $\boxed{\sigma(id_t)(\text{"}A\text{"}) \leq s.I(t) + 1}$ (by $s'.I(t) = s.I(t) + 1$)

$\Rightarrow$ $\boxed{\sigma(id_t)(\text{"}A\text{"}) \leq \sigma(id_t)(\text{"}stc\text{"}) + 1}$ (by $s.I(t) = \sigma(id_t)(\text{"}stc\text{"})$)

$\Rightarrow$ $\sigma(id_t)(\text{"}A\text{"}) - 1 \leq \sigma(id_t)(\text{"}stc\text{"})$ (assumption)

- **CASE** $s.I(t) > lower(I_s(t))$:

  By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, we have $\sigma(id_t)(\text{"}stc\text{"}) = lower(I_s(t)) = a$.
  From $\sigma(id_t)(\text{"}se\text{"}) = \texttt{true}$, we can deduce $t \in Sens(s.M)$, and from $\sigma(id_t)(\text{"}srtc\text{"}) = \texttt{false}$, we can deduce $s.reset_t(t) = \texttt{false}$. Then, by definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$ (Rule (4)), we have $s'.I(t) = s.I(t) + 1$.

  $\Rightarrow$ $\boxed{\sigma(id_t)(\text{"}A\text{"}) \leq s.I(t) + 1}$ (by $s'.I(t) = s.I(t) + 1$)

  $\Rightarrow$ $\boxed{a \leq s.I(t) + 1}$ (by $\sigma(id_t)(\text{"}A\text{"}) = a$)

  $\Rightarrow$ $\boxed{a < s.I(t)}$

  $\Rightarrow$ $lower(I_s(t)) < s.I(t)$ (assumption)

(b) $\left(\sigma(id_t)(\text{"}srtc\text{"}) \,.\, \Delta(id_t)(\text{"}tt\text{"}) \neq \texttt{NOT\_TEMP} \,.\, \sigma(id_t)(\text{"}A\text{"}) = 1\right) = \texttt{true}$

Then, converting boolean equalities into intuitionistic predicates, we have:

- $\sigma(id_t)(\text{"}srtc\text{"}) = \texttt{true}$
- $\Delta(id_t)(\text{"}tt\text{"}) \neq \texttt{NOT\_TEMP}$
- $\sigma(id_t)(\text{"}A\text{"}) = 1$

By property of the elaboration relation, and $\Delta(id_t)(\text{"}tt\text{"}) \neq \texttt{NOT\_TEMP}$, there exist an $a \in \mathbb{N}^*$ and a $ni \in \mathbb{N}^* \sqcup \{\infty\}$ s.t. $I_s(t) = [a, ni]$. Let us take such an a and ni.

By construction, $<\texttt{time\_A\_value} \Rightarrow a> \in ipm_t$, and by property of stable σ, we have $\sigma(id_t)(\text{"}A\text{"}) = a$. Thus, we can deduce $a = 1$ and $I_s(t) = [1, ni]$.

By definition of $\gamma, E_c, \tau \vdash s \overset{\uparrow}{\approx} \sigma$, from $\sigma(id_t)(\text{"}se\text{"}) = \texttt{true}$, we can deduce $t \in Sens(s.M)$, and from $\sigma(id_t)(\text{"}srtc\text{"}) = \texttt{true}$, we can deduce $s.reset_t(t) = \texttt{true}$.

By definition of $E_c, \tau \vdash s \xrightarrow{\downarrow} s'$ (Rule (3)), $t \in Sens(s.M)$ and $s.reset_t(t) = \texttt{true}$, we have $s'.I(t) = 1$.

Now, let us show $\boxed{s'.I(t) \in I_s(t)}$.

Rewriting the goal with $s'.I(t) = 1$ and $I_s(t) = [1, ni]$: $1 \in [1, ni]$.

(c) $\left(\Delta(id_t)(\text{"}tt\text{"}) = \texttt{NOT\_TEMP}\right) = \texttt{true}$

Let us show $\boxed{t \notin T_i}$.

By property of the elaboration relation and $\Delta(id_t)(\text{"}tt\text{"}) = \texttt{NOT\_TEMP}$, we have $t \notin T_i$.

□

Lemma 42 (Falling edge equal not firable). *For all sitpn, d, γ, Δ, σ_e, E_c, E_p, τ, s, s', σ, σ_i, $\sigma_\downarrow$, σ' that verify the hypotheses of Definition 27, then $\forall t \in T, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, $t \notin Firable(s') \Leftrightarrow \sigma'(id_t)("s\_firable") = \texttt{false}$.*

Proof. Proving the above lemma is trivial by appealing to Lemma 39 and by reasoning on contrapositives. □

C.4.7 Falling edge and fired transitions

Lemma 43 (Falling Edge Equal Fired Set). *For all sitpn, d, γ, Δ, σ_e, E_c, E_p, τ, s, s', σ, σ_i, $\sigma_\downarrow$, σ' that verify the hypotheses of Definition 27, then $\forall t \in T$, $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, $\forall fset \subseteq T$, s.t. $IsFiredSet(s', fset)$, $t \in fset \Leftrightarrow \sigma'(id_t)("fired") = true$.*

Proof. Given a $t \in T$, and $id_t \in Comps(\Delta)$, and a $fset \subseteq T$ s.t. $IsFiredSet(s', fset)$, let us show $\boxed{t \in fset \Leftrightarrow \sigma'(id_t)("fired") = true.}$

By definition of $IsFiredSet(s', fset)$, we have $IsFiredSetAux(s', \emptyset, T, fset)$.
Then, we can appeal to Lemma 44 to solve the goal, but first we must prove the following *extra hypothesis* (i.e, one of the premise of Lemma Falling edge equal fired set aux):

$\boxed{\begin{array}{l}\forall t' \in T, id_{t'} \in Comps(\Delta) \text{ s.t. } \gamma(t') = id_{t'},\\ (t' \in \emptyset \Rightarrow \sigma'(id_{t'})("fired") = \texttt{true}) \wedge (\sigma'(id_{t'})("fired") = \texttt{true} \Rightarrow t' \in \emptyset \vee t' \in T).\end{array}}$

Given a $t' \in T$ and an $id_{t'} \in Comps(\Delta)$ s.t. $\gamma(t') = id_{t'}$, there are two points to prove:

1. $\boxed{t' \in \emptyset \Rightarrow \sigma'(id_{t'})("fired") = \texttt{true}}$
2. $\boxed{\sigma'(id_{t'})("fired") = \texttt{true} \Rightarrow t' \in \emptyset \vee t' \in T}$

Let us show these two points:

1. Assuming $t' \in \emptyset$, let us show $\boxed{\sigma'(id_{t'})("fired") = \texttt{true}.}$

   $t' \in \emptyset$ is a contradiction.

2. Assuming $\sigma'(id_{t'})("fired") = \texttt{true}$, let us show $\boxed{t' \in \emptyset \vee t' \in T.}$

   By definition, $t' \in T$.

□

Lemma 44 (Falling edge equal fired set aux). *For all sitpn, d, γ, Δ, σ_e, E_c, E_p, τ, s, s', σ, σ_i, $\sigma_\downarrow$, σ' that verify the hypotheses of Definition 27, then $\forall t \in T, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, $\forall fired \subseteq T$, $T_s \subseteq T$, $fset \subseteq T$, assume that:*

- $IsFiredSetAux(s', fired, T_s, fset)$
- *EH (Extra. Hypothesis):*
  $\forall t' \in T$, $id_{t'} \in Comps(\Delta)$ s.t. $\gamma(t') = id_{t'}$,
  $(t' \in fired \Rightarrow \sigma'(id_{t'})("fired") = \texttt{true}) \wedge (\sigma'(id_{t'})("fired") = \texttt{true} \Rightarrow t' \in fired \vee t' \in T_s)$.

then $t \in fset \Leftrightarrow \sigma'(id_t)(\text{"}fired\text{"}) = \texttt{true}$.

Proof. Given a $t \in T$, an $id_t \in Comps(\Delta)$, a $fired, T_s, fset \subseteq T$, and assuming $IsFiredSetAux(s', fired, T_s, fset)$ and EH, let us show $\boxed{t \in fset \Leftrightarrow \sigma'(id_t)(\text{"}fired\text{"}) = \texttt{true}.}$
Let us reason by induction on $IsFiredSetAux(s', fired, T_s, fset)$.

- **BASE CASE**: $\boxed{t \in fired \Leftrightarrow \sigma'(id_t)(\text{"}fired\text{"}) = \texttt{true}.}$

  In that case, $fired = fset$ and $T_s = \emptyset$, EH looks like this:
  $\forall t' \in T,\ id_{t'} \in Comps(\Delta)$ s.t. $\gamma(t') = id_{t'}$,
  $(t' \in fired \Rightarrow \sigma'(id_{t'})(\text{"}fired\text{"}) = \texttt{true}) \wedge (\sigma'(id_{t'})(\text{"}fired\text{"}) = \texttt{true} \Rightarrow t' \in fired \vee t' \in \emptyset)$.

  From EH, we can deduce $t \in fired \Leftrightarrow \sigma'(id_t)(\text{"}fired\text{"}) = \texttt{true}.$

- **INDUCTION CASE**: $\boxed{t \in fset \Leftrightarrow \sigma'(id_t)(\text{"}fired\text{"}) = \texttt{true}.}$

  In that case, we have:

  – $IsTopPrioritySet(T_s, tp)$
  – $ElectFired(s', fired, tp, fired')$
  – $FiredAux(s', fired', T_s \setminus tp, fset)$

  $(\forall t' \in T, id_{t'} \in Comps(\Delta)\ s.t.\ \gamma(t') = id_{t'},$
  $(t' \in fired' \Rightarrow \sigma'(id_{t'})(\text{"}fired\text{"}) = \texttt{true}) \wedge (\sigma'(id_{t'})(\text{"}fired\text{"}) = \texttt{true} \Rightarrow t' \in fired' \vee t' \in T_s \setminus tp)) \Rightarrow$
  $t \in fset \Leftrightarrow \sigma'_t(\text{"}fired\text{"}) = true.$

  Applying the induction hypothesis, then, the new goal is:

  $\forall t' \in T, id_{t'} \in Comps(\Delta)\ s.t.\ \gamma(t') = id_{t'},$
  $(t' \in fired' \Rightarrow \sigma'(id_{t'})(\text{"}fired\text{"}) = \texttt{true})$
  $\wedge(\sigma'(id_{t'})(\text{"}fired\text{"}) = \texttt{true} \Rightarrow t' \in fired' \ \vee\ t' \in T_s \setminus tp)$

  Apply Lemma Elect Fired Equal Fired to solve the goal.

□

Lemma 45 (Elect Fired Equal Fired). *For all* $sitpn, d, \gamma, \Delta, \sigma_e, E_c, E_p, \tau, s, s', \sigma, \sigma_i, \sigma_\downarrow, \sigma'$ *that verify the hypotheses of Definition 27, then* $\forall fired, fired', T_s, tp,\ fset \subseteq T$, *assume that:*

- $IsTopPrioritySet(T_s, tp)$
- $ElectFired(s', fired, tp, fired')$

- $FiredAux(s', fired', T_s \setminus tp, fset)$
- *EH (Extra. Hypothesis):*
  $\forall t' \in T, id_{t'} \in Comps(\Delta)$ *s.t.* $\gamma(t') = id_{t'}$,
  $(t' \in fired \Rightarrow \sigma'(id_{t'})(\text{"}fired\text{"}) = \texttt{true}) \wedge (\sigma'(id_{t'})(\text{"}fired\text{"}) = \texttt{true} \Rightarrow t' \in fired \vee t' \in T_s)$

then $\forall t \in T, id_t \in Comps(\Delta)$ *s.t.* $\gamma(t) = id_t$,
$\big(t \in fired' \Rightarrow \sigma'(id_t)(\text{"}fired\text{"}) = \texttt{true}\big) \wedge \big(\sigma'(id_t)(\text{"}fired\text{"}) = \texttt{true} \Rightarrow t \in fired' \vee t \in T_s \setminus tp\big)$.

Proof. Given a $t \in T$ and an $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, let us show

$\boxed{\big(t \in fired' \Rightarrow \sigma'(id_t)(\text{"}fired\text{"}) = \texttt{true}\big) \wedge \big(\sigma'(id_t)(\text{"}fired\text{"}) = \texttt{true} \Rightarrow t \in fired' \vee t \in T_s \setminus tp\big).}$

Let us reason by induction on $ElectFired(s', fired, tp, fired')$; there are three cases:

1. **BASE CASE**: $tp = \emptyset$ and $fired = fired'$.
2. **INDUCTIVE CASE**: $tp = \{t_0\} \cup tp_0$ and t_0 is elected to be fired.
3. **INDUCTIVE CASE**: $tp = \{t_0\} \cup tp_0$ and t_0 is not elected to be fired.

Let us prove the goal in these three contexts:

1. **BASE CASE**:

   $\boxed{\big(t \in fired \Rightarrow \sigma'(id_t)(\text{"}fired\text{"}) = \texttt{true}\big) \wedge \big(\sigma'(id_t)(\text{"}fired\text{"}) = \texttt{true} \Rightarrow t \in fired \vee t \in T_s\big).}$

   Apply EH to solve the goal.

2. **INDUCTIVE CASE**: $tp = \{t_0\} \cup tp_0$ and t_0 is elected to be fired.
   In that case, we have:

   - $IsTopPrioritySet(T_s, \{t_0\} \cup tp_0)$
   - $ElectFired(s', fired \cup \{t_0\}, tp_0, fired')$
   - $IsFiredSetAux(s', fired', T_s \setminus \{t_0\} \cup tp_0, fset)$
   - $t_0 \in Firable(s')$
   - $t_0 \in Sens(s'.M - \sum_{t_i \in Pr(t, fired)} pre(t_i))$ where $Pr(t, fired) = \{t' \mid t' \succ t \wedge t' \in fired\}$
   - EH: $\forall t' \in T$, $id_{t'} \in Comps(\Delta)$ s.t. $\gamma(t') = id_{t'}$,
     $(t' \in fired \Rightarrow \sigma'(id_{t'})(\text{"}f\text{"}) = \texttt{true}) \wedge (\sigma'(id_{t'})(\text{"}f\text{"}) = \texttt{true} \Rightarrow t' \in fired \vee t' \in T_s)$

   $\forall T'_s \subseteq T$,
   $IsTopPrioritySet(T'_s, tp_0) \Rightarrow$
   $IsFiredSetAux(s', fired', T'_s \setminus tp_0, fset) \Rightarrow$
   $\big(\forall t' \in T, id_{t'} \in Comps(\Delta)$ s.t. $\gamma(t') = id_{t'}$,
   $(t' \in fired \cup \{t_0\} \Rightarrow \sigma'_{t'}(\text{"}f\text{"}) = \texttt{true}) \wedge (\sigma'(id_{t'})(\text{"}f\text{"}) = \texttt{true} \Rightarrow t' \in fired \cup \{t_0\} \vee t' \in T'_s)\big) \Rightarrow$
   $\forall t \in T$, $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$,

$\big(t \in fired' \Rightarrow \sigma'(id_t)(''f'') = \texttt{true}\big) \wedge \big(\sigma'(id_t)(''f'') = \texttt{true} \Rightarrow t \in fired' \vee t \in T'_s \setminus tp_0\big)$

$\forall t \in T,\ id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$,
$\big(t \in fired' \Rightarrow \sigma'_t(''f'') = \texttt{true}\big) \wedge \big(\sigma'_t(''f'') = \texttt{true} \Rightarrow t \in fired' \vee t \in T_s \setminus \{t_0\} \cup tp_0\big)$

To solve the goal, we can apply the induction hypothesis with $T'_s = T_s \setminus \{t_0\}$; then, there are three points to prove:

(a) $IsTopPrioritySet(T_s \setminus \{t_0\}, tp_0)$

(b) $IsFiredSetAux(s', fired', (T_s \setminus \{t_0\}) \setminus tp_0, fset)$

(c) $\forall t' \in T, id_{t'} \in Comps(\Delta)$ s.t. $\gamma(t') = id_{t'}$,
$(t' \in fired \cup \{t_0\} \Rightarrow \sigma'_{t'}(''f'') = \texttt{true}) \wedge (\sigma'(id_{t'})(''f'') = \texttt{true} \Rightarrow t' \in fired \cup \{t_0\} \vee t' \in T_s \setminus \{t_0\})$

Let us prove these three points:

(a) $IsTopPrioritySet(T_s \setminus \{t_0\}, tp_0)$

Not provable yet.

(b) $IsFiredSetAux(s', fired', (T_s \setminus \{t_0\}) \setminus tp_0, fset)$.
We know that $(T_s \setminus \{t_0\}) \setminus tp_0 = T_s \setminus (\{t_0\} \cup tp_0)$, and thus
$IsFiredSetAux(s', fired', T_s \setminus (\{t_0\} \cup tp_0), fset)$ is an assumption.

(c) $\forall t' \in T, id_{t'} \in Comps(\Delta)$ s.t. $\gamma(t') = id_{t'}$,
$(t' \in fired \cup \{t_0\} \Rightarrow \sigma'(id_{t'})(''f'') = \texttt{true}) \wedge (\sigma'(id_{t'})(''f'') = \texttt{true} \Rightarrow t' \in fired \cup \{t_0\} \vee t' \in T_s \setminus \{t_0\})$

Given a $t' \in T$ and an $id_{t'} \in Comps(\Delta)$ s.t. $\gamma(t') = id_{t'}$, let us show
$(t' \in fired \cup \{t_0\} \Rightarrow \sigma'(id_{t'})(''f'') = \texttt{true})$
$\wedge (\sigma'(id_{t'})(''f'') = \texttt{true} \Rightarrow t' \in fired \cup \{t_0\} \vee t' \in T_s \setminus \{t_0\})$.
The proof is in two parts.

i. Assuming that $t' \in fired \cup \{t_0\}$, let us show $\sigma'(id_{t'})(''f'') = \texttt{true}$.
Case analysis on $t' \in fired \cup \{t_0\}$; there are two cases:
- $t' \in fired$
- $t' = t_0$

Let us prove the goal in these two contexts.
- **CASE** $t' \in fired$: Thanks to EH, we can deduce $\sigma'_{t'}(''f'') = \texttt{true}$.

- **CASE** $t' = t_0$:
  By definition of $id_{t'}$, there exist a $gm_{t'}, ipm_{t'}, opm_{t'}$ s.t. $\texttt{comp}(id_{t'}, "transition", gm_{t'}, ipm_{t'}, opm_{t'}) \in d.cs$.
  By property of the stabilize relation and $\texttt{comp}(id_{t'}, "transition", gm_{t'}, ipm_{t'}, opm_{t'}) \in d.cs$, and through the examination of the `fired_evaluation` process defined in the transition design architecture:

$$\sigma(id_{t'})("f") = \sigma(id_{t'})("sfa") \,.\, \sigma(id_{t'})("spc") \tag{C.48}$$

  Rewriting the goal with (C.48): $\boxed{\sigma(id_{t'})("sfa") \,.\, \sigma(id_{t'})("spc") = \texttt{true}.}$
  Then, there are two points to prove:
  A. $\boxed{\sigma(id_{t'})("sfa") = \texttt{true}.}$
  B. $\boxed{\sigma(id_{t'})("spc") = \texttt{true}.}$
  Let us prove these two points:
  A. $\boxed{\sigma(id_{t'})("sfa") = \texttt{true}.}$
  Appealing to Lemma 39, we can deduce $\sigma(id_{t'})("sfa") = \texttt{true}.$
  B. $\boxed{\sigma(id_{t'})("spc") = \texttt{true}.}$
  Appealing to Lemma 46, we can deduce $\sigma(id_{t'})("spc") = \texttt{true}.$

ii. Assuming that $\sigma'(id_{t'})("f") = \texttt{true}$, let us show $\boxed{t' \in fired \cup \{t_0\} \ \vee \ t' \in T_s \setminus \{t_0\}.}$
From $\sigma'(id_{t'})("f") = \texttt{true}$ and EH, we can deduce that $t' \in fired \vee t' \in T_s$.
Case analysis on $t' \in fired \vee t' \in T_s$.

- **CASE** $t' \in fired$: then, it is trivial to show $t' \in fired \cup \{t_0\}.$
- **CASE** $t' \in T_s$: We know that $t_0 \in T_s$. Therefore, either $t' \in T_s \setminus \{t_0\}$, or $t' = t_0$, and then, $t' \in fired \cup \{t_0\}.$

3. **INDUCTIVE CASE**: $tp = \{t_0\} \cup tp_0$ and t_0 is not elected to be fired.

- $IsTopPrioritySet(T_s, \{t_0\} \cup tp_0)$
- $ElectFired(s', fired, tp_0, fired')$
- $IsFiredSetAux(s', fired', T_s \setminus \{t_0\} \cup tp_0, fset)$
- $\neg\big(t_0 \in Firable(s') \wedge t_0 \in Sens(s'.M - \sum_{t_i \in Pr(t, fired)} pre(t_i))\big)$
- EH:
  $\forall t' \in T, id_{t'} \in Comps(\Delta)$ s.t. $\gamma(t') = id_{t'}$,
  $(t' \in fired \Rightarrow \sigma'(id_{t'})("f") = \texttt{true}) \wedge (\sigma'(id_{t'})("f") = \texttt{true} \Rightarrow t' \in fired \ \vee \ t' \in T_s)$

$\forall T'_s \subseteq T,$
$IsTopPrioritySet(T'_s, tp_0) \Rightarrow$
$IsFiredSetAux(s', fired', T'_s \setminus tp_0, fset) \Rightarrow$
$(\forall t' \in T, id_{t'} \in Comps(\Delta)$ s.t. $\gamma(t') = id_{t'},$
$(t' \in fired \Rightarrow \sigma'(id_{t'})("f") = \texttt{true}) \wedge (\sigma'(id_{t'})("f") = \texttt{true} \Rightarrow t' \in fired \vee t' \in T'_s)) \Rightarrow$
$\forall t \in T, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t,$
$(t \in fired' \Rightarrow \sigma'(id_t)("f") = \texttt{true}) \wedge (\sigma'(id_t)("f") = \texttt{true} \Rightarrow t \in fired' \vee t \in T'_s \setminus tp_0)$

$\forall t \in T, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t,$
$(t \in fired' \Rightarrow \sigma'(id_t)("f") = \texttt{true}) \wedge (\sigma'(id_t)("f") = \texttt{true} \Rightarrow t \in fired' \vee t \in T_s \setminus \{t_0\} \cup tp_0).$

Then, we can apply the induction hypothesis with $T'_s = T_s \setminus \{t_0\}$, then, there are three points to prove:

(a) $IsTopPrioritySet(T_s \setminus \{t_0\}, tp_0)$

(b) $IsFiredSetAux(s', fired', (T_s \setminus \{t_0\}) \setminus tp_0, fset)$

(c) $\forall t' \in T, id_{t'} \in Comps(\Delta)$ s.t. $\gamma(t') = id_{t'},$
$(t' \in fired \Rightarrow \sigma'(id_{t'})("f") = \texttt{true}) \wedge (\sigma'(id_{t'})("f") = \texttt{true} \Rightarrow t' \in fired \vee t' \in T_s \setminus \{t_0\})$

Let us prove these three points:

(a) $IsTopPrioritySet(T_s \setminus \{t_0\}, tp_0)$

Not provable yet.

(b) $IsFiredSetAux(s', fired', (T_s \setminus \{t_0\}) \setminus tp_0, fset)$
We know that $(T_s \setminus \{t_0\}) \setminus tp_0 = T_s \setminus (\{t_0\} \cup tp_0)$, and thus
$IsFiredSetAux(s', fired', T_s \setminus (\{t_0\} \cup tp_0), fset)$ is an assumption.

(c) $\forall t' \in T, id_{t'} \in Comps(\Delta)$ s.t. $\gamma(t') = id_{t'},$
$(t' \in fired \Rightarrow \sigma'(id_{t'})("f") = \texttt{true}) \wedge (\sigma'(id_{t'})("f") = \texttt{true} \Rightarrow t' \in fired \vee t' \in T_s \setminus \{t_0\})$
Given a $t' \in T$ and an $id_{t'} \in Comps(\Delta)$ s.t. $\gamma(t') = id_{t'}$, let us show

$(t' \in fired \Rightarrow \sigma'(id_{t'})("f") = \texttt{true}) \wedge (\sigma'(id_{t'})("f") = \texttt{true} \Rightarrow t' \in fired \vee t' \in T_s \setminus \{t_0\})$

The proof is in two parts:

i. Assuming that $t' \in fired$, let us show $\boxed{\sigma'(id_{t'})("f") = \texttt{true}.}$
From $t' \in fired$ and EH, $\sigma'(id_{t'})("f") = \texttt{true}.$

ii. Assuming that $\sigma'(id_{t'})("f") = \texttt{true}$, let us show $\boxed{t' \in fired \ \vee \ t' \in T_s \setminus \{t_0\}.}$
Thanks to $\sigma'(id_{t'})("f") = \texttt{true}$ and EH, we know that: $t' \in fired \vee t' \in T_s$.
Case analysis on $t' \in fired \vee t' \in T_s$; there are two cases:

- **CASE** $t' \in fired.$
- **CASE** $t' \in T_s$:
  From $IsTopPrioritySet(T_s, \{t_0\} \cup tp_0)$, we can deduce that $t_0 \in T_s$. Therefore, either $t' \in T_s \setminus \{t_0\}$ or $t' = t_0$.
  In the case where $t' = t_0$, we need to show a contradiction by proving $t' \in Firable(s')$ and $t' \in Sens(s'.M - \sum_{t_i \in Pr(t,fired)} pre(t_i))$ based on $\sigma'(id_{t'})("f") = \texttt{true}$.
  By definition of $id_{t'}$, there exist a $gm_{t'}, ipm_{t'}, opm_{t'}$ s.t. $\texttt{comp}(id_{t'}, "transition", gm_{t'}, ipm_{t'}, opm_{t'}) \in d.cs$.
  By property of the stabilize relation and $\texttt{comp}(id_{t'}, "transition", gm_{t'}, ipm_{t'}, opm_{t'}) \in d.cs$:
  $$\sigma(id_{t'})("f") = \sigma(id_{t'})("sfa") \,.\, \sigma(id_{t'})("spc") = \texttt{true} \tag{C.49}$$
  From $\sigma(id_{t'})("sfa") = \texttt{true}$, and appealing to Lemma Falling edge equal firable, we can deduce $t' \in Firable(s')$.
  From $\sigma(id_{t'})("spc") = \texttt{true}$, and appealing to Lemma Stabilize Compute Priority Combination After Falling Edge, we can deduce $t' \in Sens(s'.M - \sum_{t_i \in Pr(t,fired)} pre(t_i))$.
  Then, as $t' = t_0$, $\neg(t_0 \in Firable(s') \wedge t_0 \in Sens(s'.M - \sum_{t_i \in Pr(t,fired)} pre(t_i)))$ is a contradiction.

□

Lemma 46 (Stabilize Compute Priority Combination After Falling Edge). *For all $sitpn$, d, γ, Δ, σ_e, E_c, E_p, τ, s, s', σ, σ_i, $\sigma_\downarrow$, σ' that verify the hypotheses of Definition 27, then $\forall t \in T, id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$,*
$\forall fired, fired', T_s, tp, fset \subseteq T$ assume that:

- $IsTopPrioritySet(T_s, \{t\} \cup tp)$
- $ElectFired(s', fired, tp, fired')$
- $FiredAux(s', fired', T_s \setminus \{t\} \cup tp, fset)$
- *EH:* $\forall t' \in T, id_{t'} \in Comps(\Delta)$ s.t. $\gamma(t') = id_{t'}$,
  $(t' \in fired \Rightarrow \sigma'(id_{t'})("f") = \texttt{true}) \wedge (\sigma'(id_{t'})("f") = \texttt{true} \Rightarrow t' \in fired \ \vee \ t' \in T_s)$.
- $t \in Firable(s')$

then $t \in Sens(s'.M - \sum_{t_i \in Pr(t,fired)} pre(t_i)) \Leftrightarrow \sigma'(id_t)(''spc'') = \texttt{true}$

Proof. Given a $t \in T$ and an $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$, a $fired$, $fired'$, T_s, tp, $fset \subseteq T$ and assuming all the above hypotheses, let us show

$$t \in Sens(s'.M - \sum_{t_i \in Pr(t,fired)} pre(t_i)) \Leftrightarrow \sigma'(id_t)(''spc'') = \texttt{true}.$$

By construction and by definition of id_t, there exist gm_t, ipm_t, opm_t s.t. $\texttt{comp}(id_t, ''transition'', gm_t, ipm_t, opm_t) \in d.cs$.
By property of the stabilize relation, $\texttt{comp}(id_t, ''transition'', gm_t, ipm_t, opm_t) \in d.cs$, and through the examination of the `priority_authorization_evaluation` process defined in the transition design architecture, we can deduce:

$$\sigma'(id_t)(''spc'') = \prod_{i=0}^{\Delta(id_t)(''ian'')-1} \sigma'(id_t)(''pauths''')[i] \tag{C.50}$$

Rewriting the goal with (C.50):

$$t \in Sens(s'.M - \sum_{t_i \in Pr(t,fired)} pre(t_i)) \Leftrightarrow \prod_{i=0}^{\Delta(id_t)(''ian'')-1} \sigma'(id_t)(''pauths''')[i] = \texttt{true}.$$

Then, the proof is in two parts:

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

Let us prove both sides of the equivalence:

1. Assuming that $t \in Sens(s'.M - \sum_{t_i \in Pr(t,fired)} pre(t_i))$, let us show

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

   Let us perform case analysis on $input(t)$; there are 2 cases:

   - **CASE** $input(t) = \emptyset$:
     By construction, <`input_arcs_number` $\Rightarrow$1> $\in gm_t$ and <`priority_authorizations(0)` $\Rightarrow$ `true`> $\in ipm_t$.
     By property of the elaboration relation, we have $\Delta(id_t)(''ian'') = 1$, and by property of the stabilize relation, we have $\sigma'(id_t)(''pauths''')[0] = \texttt{true}$.
     Rewriting the goal with $\Delta(id_t)(''ian'') = 1$ and $\sigma'(id_t)(''pauths''')[0] = \texttt{true}$, and simplifying the goal: tautology.

- **CASE** $input(t) \neq \emptyset$:
  Then, let us show an equivalent goal:
  $\boxed{\forall i \in [0, \Delta(id_t)(\text{"}ian\text{"}) - 1],\ \sigma'(id_t)(\text{"}pauths\text{"})[i] = \texttt{true}.}$
  Given an $i \in [0, \Delta(id_t)(\text{"}ian\text{"}) - 1]$, let us show $\boxed{\sigma'(id_t)(\text{"}pauths\text{"})[i] = \texttt{true}.}$
  By construction, $<\texttt{input\_arcs\_number} \Rightarrow |input(t)|> \in gm_t$.
  By property of the elaboration relation, we have $\Delta(id_t)(\text{"}ian\text{"}) = |input(t)|$. Then, we can deduce $i \in [0, |input(t)| - 1]$.
  By construction, for all $i \in [0, |input(t)| - 1]$, there exist a $p \in input(t)$ and an $id_p \in Comps(\Delta)$ s.t. $\gamma(p) = id_p$, there exist a gm_p, ipm_p, opm_p s.t. $\texttt{comp}(id_p, \text{"}place\text{"}, gm_p, ipm_p, opm_p) \in d.cs$, and there exist a $j \in [0, |output(p)|]$ and an $id_{ji} \in Sigs(\Delta)$ s.t. $<\texttt{input\_arcs\_valid(i)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in ipm_t$ and $<\texttt{output\_arcs\_valid(j)} \Rightarrow \texttt{id}_{\texttt{ji}}> \in opm_t$. Let us take such a $p \in input(t)$, $id_p \in Comps(\Delta)$, gm_p, ipm_p, opm_p, $j \in [0, |output(p)|]$ and $id_{ji} \in Sigs(\Delta)$.
  Now, let us perform case analysis on the nature of the arc connecting p and t; there are 2 cases:
  - **CASE** $pre(p,t) = (\omega, \texttt{test})$ or $pre(p,t) = (\omega, \texttt{inhib})$:
    By construction, $<\texttt{priority\_authorizations(i)} \Rightarrow \texttt{true}> \in ipm_t$, and by property of the stabilize relation: $\sigma'(id_t)(\text{"}pauths\text{"})[i] = \texttt{true}.$
  - **CASE** $pre(p,t) = (\omega, \texttt{basic})$:
    Let us define $output_c(p) = \{t \in T \mid \exists \omega,\ pre(p,t) = (\omega, \texttt{basic})\}$, the set of output transitions of p that are in conflict. Then, there are two cases, one for each way to solve the conflicts between the output transitions of p:
    * **CASE** For all pair of transitions in $output_c(p)$, all conflicts are solved by mutual exclusion:
      By construction, $<\texttt{priority\_authorizations(i)} \Rightarrow \texttt{true}> \in ipm_t$, and by property of the stabilize relation: $\sigma'(id_t)(\text{"}pauths\text{"})[i] = \texttt{true}.$
    * **CASE** The priority relation is a strict total order over the set $output_c(p)$:
      By construction, there exists an $id'_{ji} \in Sigs(\Delta)$ s.t.
      $<\texttt{priority\_authorizations(i)} \Rightarrow \texttt{id}'_{\texttt{ji}}> \in ipm_t$ and
      $<\texttt{priority\_authorizations(j)} \Rightarrow \texttt{id}'_{\texttt{ji}}> \in opm_p$.
      By property of the stabilize relation, $\texttt{comp}(id_t, \text{"}transition\text{"}, gm_t, ipm_t, opm_t) \in d.cs$ and $\texttt{comp}(id_p, \text{"}place\text{"}, gm_p, ipm_p, opm_p) \in d.cs$, we can deduce:

      $$\sigma'(id_t)(\text{"}pauths\text{"})[i] = \sigma'(id'_{ji}) = \sigma'(id_p)(\text{"}pauths\text{"})[j] \tag{C.51}$$

      Rewriting the goal with (C.51): $\boxed{\sigma'(id_p)(\text{"}pauths\text{"})[j] = \texttt{true}.}$
      By property of the stabilize relation, $\texttt{comp}(id_p, \text{"}place\text{"}, gm_p, ipm_p, opm_p) \in d.cs$, and through the examination of the `priority_evaluation` process defined in the place design behavior, we can deduce:

      $$\sigma'(id_p)(\text{"}pauths\text{"})[j] = (\sigma'(id_p)(\text{"}sm\text{"}) \geq \texttt{vsots} + \sigma'(id_p)(\text{"}oaw\text{"})[j]) \tag{C.52}$$

Let us define the vsots term as follows:

$$\texttt{vsots} = \sum_{i=0}^{j-1} \begin{cases} \sigma'(id_p)("oaw")[i] \texttt{ if } \sigma'(id_p)("otf")[i]. \\ \qquad\qquad\qquad\qquad \sigma'(id_p)("oat")[i] = \texttt{basic} \\ 0 \textit{ otherwise} \end{cases} \tag{C.53}$$

Rewriting the goal with (C.52): $\boxed{\sigma'(id_p)("sm") \geq \texttt{vsots} + \sigma'(id_p)("oaw")[j]}$

By definition of $t \in Sens(s'.M - \sum_{t_i \in Pr(t,fired)} pre(t_i))$, we can deduce:

$s'.M(p) \geq \sum_{t_i \in Pr(t,fired)} pre(p,t_i) + \omega$.

Then, there are three points to prove:

(a) $\boxed{s'.M(p) = \sigma'(id_p)("sm")}$

(b) $\boxed{\omega = \sigma'(id_p)("oaw")[j]}$

(c) $\boxed{\sum_{t_i \in Pr(t,fired)} pre(p,t_i) = \texttt{vsots}}$

Let us prove these three points:

(a) $\boxed{s'.M(p) = \sigma'(id_p)("sm")}$

Appealing to Lemma 32: $s'.M(p) = \sigma'(id_p)("sm")$.

(b) $\boxed{\omega = \sigma'(id_p)("oaw")[j]}$

By construction, and as $pre(p,t) = (\omega, \texttt{basic})$, we have $<\texttt{output\_arcs\_weights(j)} \Rightarrow \omega> \in ipm_p$.

By property of the stabilize relation and $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$: $\omega = \sigma'(id_p)("oaw")[j]$.

(c) $\boxed{\sum_{t_i \in Pr(t,fired)} pre(p,t_i) = \texttt{vsots}}$

Let us replace the left and right term of the equality by their full definition:

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

Let us define $f(t_i) = \begin{cases} \omega \texttt{ if } pre(p,t_i) = (\omega, \texttt{basic}) \\ 0 \textit{ otherwise} \end{cases}$ and

$$g(i) = \begin{cases} \sigma'(id_p)(''oaw'')[i] \texttt{ if } \sigma'(id_p)(''otf'')[i]. \\ \qquad\qquad\qquad\qquad \sigma'(id_p)(''oat'')[i] = \texttt{basic} \\ 0 \textit{ otherwise} \end{cases}$$

Let us reason by induction on the right term of the goal.

BASE CASE: then, we have $i > j - 1$, and then $j = 0$.

$$\sum_{t_i \in Pr(t,fired)} \begin{cases} \omega \texttt{ if } pre(p, t_i) = (\omega, \texttt{basic}) \\ 0 \textit{ otherwise} \end{cases} = 0$$

By property of the well-definition of $sitpn$, the priority relation is a strict total order over the transitions of set $output_c(p)$. This ordering is reflected in the ordering of the indexes of the output port `priority_authorizations` for each place component instance. Thus, in the `priority_authorizations` output port of a place component instance, the element of index 0 is connected to the transition of $output_c(t)$ with the highest firing priority. We know that component id_t is connected to `priority_authorizations(0)` in the output port map of component id_p. By construction, transition t is the transition of $output_c(p)$ with the highest firing priority, i.e, $\nexists t' \in output_c(p)$ s.t. $t' \succ t$.

For all transition $t_i \in Pr(t, fired)$, either t_i is not in $output_c(p)$, and thus t_i has no effect in the value of the sum term $\sum_{t_i \in Pr(t,fired)} f(t_i)$; or, $t_i \in output_c(p)$. Then, by definition of $t_i \in Pr(t, fired)$, $t_i \succ t$, which is contradiction with $\nexists t' \in output_c(p)$ s.t. $t' \succ t$.

INDUCTIVE CASE: then, $0 \leq j - 1$, and thus $j > 0$.

For all $Pr' \subseteq T$, $g(0) + \sum_{t_i \in Pr'} f(t_i) = g(0) + \sum_{i=1}^{j-1} g(i)$

$$\sum_{t_i \in Pr(t,fired)} f(t_i) = g(0) + \sum_{i=1}^{j-1} g(i).$$

By definition of $g(0)$:

$$\sum_{t_i \in Pr(t,fired)} f(t_i) = \begin{cases} \sigma'(id_p)(''oaw'')[0] \texttt{ if } \sigma'(id_p)(''otf'')[0]. \\ \qquad\qquad\qquad\qquad \sigma'(id_p)(''oat'')[0] = \texttt{basic} \\ 0 \textit{ otherwise} \end{cases} + \sum_{i=1}^{j-1} g(i).$$

Case analysis on the value of $\sigma'(id_p)("otf")[0] \,.\, \sigma'(id_p)("oat")[0] = \texttt{basic}$:

In the case where $\big(\sigma'(id_p)("otf")[0] \,.\, \sigma'(id_p)("oat")[0] = \texttt{basic}\big) = \texttt{false}$, then $g(0) = 0$, and we can use the induction hypothesis with $Pr' = Pr(t, fired)$ to prove the goal.

In the case where $\big(\sigma'(id_p)("otf")[0] \,.\, \sigma'(id_p)("oat")[0] = \texttt{basic}\big) = \texttt{true}$, then $g(0) = \sigma'(id_p)("oaw")[0]$:

$$\sum_{t_i \in Pr(t,fired)} f(t_i) = \sigma'(id_p)("oaw")[0] + \sum_{i=1}^{j-1} g(i).$$

By construction, and knowing that $j > 0$ and that the priority relation is a strict total order over the set $output_c(p)$, there exist a $t_0 \in output_c(p)$, an $id_{t_0} \in Comps(\Delta)$, gm_{t_0}, ipm_{t_0} opm_{t_0}, and an $id_{ft_0} \in Sigs(\Delta)$ such that:

- $\gamma(t_0) = id_{t_0}$
- $t_0 \succ t$
- $\texttt{comp}(id_{t_0}, "transition", gm_{t_0}, ipm_{t_0}, opm_{t_0}) \in d.cs$
- $<\texttt{fired} \Rightarrow \texttt{id}_{\texttt{ft}_0}> \in opm_{t_0}$
- $<\texttt{output\_transitions\_fired}(0) \Rightarrow \texttt{id}_{\texttt{ft}_0}> \in ipm_p$

By property of the stabilize relation, $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$ and $\texttt{comp}(id_{t_0}, "transition", gm_{t_0}, ipm_{t_0}, opm_{t_0}) \in d.cs$:

$$\sigma'(id_{t_0})("f") = \sigma'(id_{ft_0}) = \sigma'(id_p)("otf")[0] = \texttt{true} \tag{C.54}$$

From EH and $\sigma'(id_{t_0})("f") = \texttt{true}$, we have either $t_0 \in fired$ or $t_0 \in T_s$.

❑ In the case where $t_0 \in fired$, then, by definition of $\sum$:

$$f(t_0) + \sum_{t_i \in Pr(t,fired) \setminus \{t_0\}} f(t_i) = \sigma'(id_p)("oaw")[0] + \sum_{i=1}^{j-1} g(i).$$

By definition of $t_0 \in output_c(p)$, there exists $\omega \in \mathbb{N}^*$ s.t. $pre(p, t_0) = (\omega, \texttt{basic})$. Thus, we have $f(t_0) = \omega$
By construction, $<\texttt{output\_arcs\_weights}(0) \Rightarrow \omega>$, and by property of the stabilize relation, we have $\sigma'(id_p)("oaw")[0] = \omega$. Thus, we can deduce that $g(0) = \omega$, and then we can rewrite the goal in order to apply the induction hypothesis with $Pr' = Pr(t, fired) \setminus \{t_0\}$.

❑ In the case where $t_0 \in T_s$:
As t is a top-priority transition in set T_s, there exists no transition $t' \in T_s$ s.t. $t' \succ t$.
Contradicts $t_0 \succ t$.

2. Assuming that $\prod_{i=0}^{\Delta(id_t)("ian")-1} \sigma'(id_t)("pauths")[i] = \texttt{true}$, let us show

$$\boxed{t \in Sens(s'.M - \sum_{t_i \in Pr(t,fired)} pre(t_i)).}$$

By definition of $t \in Sens(s'.M - \sum_{t_i \in Pr(t,fired)} pre(t_i))$:

$$\boxed{\begin{array}{l}\forall p \in P, \omega \in \mathbb{N}^*, \\ \big((pre(p,t) = (\omega, \texttt{basic}) \vee pre(p,t) = (\omega, \texttt{test})) \Rightarrow s'.M(p) - \sum_{t_i \in Pr(t,fired)} pre(p,t_i) \geq \\ \omega\big) \\ \wedge \big(pre(p,t) = (\omega, \texttt{inhib}) \Rightarrow s'.M(p) - \sum_{t_i \in Pr(t,fired)} pre(p,t_i) < \omega\big)\end{array}}$$

Given a $p \in P$ and an $\omega \in \mathbb{N}^*$, let us show

$$\boxed{\begin{array}{l}\big((pre(p,t) = (\omega, \texttt{basic}) \vee pre(p,t) = (\omega, \texttt{test})) \Rightarrow s'.M(p) - \sum_{t_i \in Pr(t,fired)} pre(p,t_i) \geq \\ \omega\big) \\ \wedge \big(pre(p,t) = (\omega, \texttt{inhib}) \Rightarrow s'.M(p) - \sum_{t_i \in Pr(t,fired)} pre(p,t_i) < \omega\big)\end{array}}$$

By construction, there exists an $id_p \in Comps(\Delta)$ s.t. $\gamma(p) = id_p$. By construction and by definition of id_p, there exist gm_p, ipm_p, opm_p s.t. $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$.

There are three different cases:

(a) Assuming that $pre(p,t) = (\omega, \texttt{test})$, let us show $\boxed{s'.M(p) - \sum_{t_i \in Pr(t,fired)} pre(p,t_i) \geq \omega.}$

Then, assuming that the priority relation is well-defined, there exists no transition t_i connected by a `basic` arc to p that verifies $t_i \succ t$. This is because t is connected to p by a `test` arc; thus, t is not in conflict with the other output transitions of p; thus, there is no relation of priority between t and the output of p.

Then, we can deduce that $\sum_{t_i \in Pr(t,fired)} pre(p,t_i) = 0$.

Then, the new goal is $s'.M(p) \geq \omega$.

Knowing that $t \in Firable(s')$, thus, $t \in Sens(s'.M)$, thus, we have $s'.M(p) \geq \omega$.

(b) Assuming that $pre(p,t) = (\omega, \texttt{inhib})$, let us show $\boxed{s'.M(p) - \sum_{t_i \in Pr(t,fired)} pre(p,t_i) < \omega.}$

Use the same strategy as above.

(c) Assuming that $pre(p,t) = (\omega, \texttt{basic})$, let us show $\boxed{s'.M(p) - \sum_{t_i \in Pr(t,fired)} pre(p,t_i) \geq \omega.}$

Then, there are two cases:

i. **CASE** For all pair of transitions in $output_c(p)$, all conflicts are solved by mutual exclusion.
Then, assuming that the priority relation is well-defined, it must not be defined over the set $output_c(t)$, and we know that $t \in output_c(p)$ since $pre(p,t) = (\omega, \texttt{basic})$.
Then, there exists no transition t_i connected to p by a `basic` arc that verifies $t_i \succ t$.
Then, we can deduce $\sum_{t_i \in Pr(t,fired)} pre(p,t_i) = 0$.
Then, the new goal is $s'.M(p) \geq \omega$.
We know $t \in Firable(s')$, thus, $t \in Sens(s'.M)$, thus, $s'.M(p) \geq \omega$.

ii. **CASE** The priority relation is a strict total order over the set $output_c(p)$.
By construction, there exists $id_t \in Comps(\Delta)$ s.t. $\gamma(t) = id_t$. By construction and by definition of id_t, there exist gm_t, ipm_t, opm_t s.t. $\texttt{comp}(id_t, "transition", gm_t, ipm_t, opm_t) \in d.cs$.
By construction, there exist $j \in [0, |input(t)| - 1]$, $k \in [0, |output(t)| - 1]$, and $id_{kj} \in Sigs(\Delta)$ s.t. $<\texttt{priority\_authorizations(j)} \Rightarrow \texttt{id}_{\texttt{kj}}> \in ipm_t$ and $<\texttt{priority\_authorizations(k)} \Rightarrow \texttt{id}_{\texttt{kj}}> \in opm_p$. Let us take such an j, k and id_{kj}.
From $\prod_{i=0}^{\Delta(id_t)("ian")-1} \sigma'(id_t)("pauths")[i] = \texttt{true}$, we can deduce that for all $i \in [0, \Delta(id_t)\ ("ian") - 1]$, $\sigma'(id_t)("pauths")[i] = \texttt{true}$.
By construction, $<\texttt{input\_arcs\_number} \Rightarrow |input(t)|> \in gm_t$, and by property of the elaboration relation, we have $\Delta(id_t)("ian") = |input(t)|$. Then, from $j \in [0, |input(t)| - 1]$, we can deduce $j \in [0, \Delta(id_t)("ian") - 1]$. And, from $\forall i \in [0, \Delta(id_t)("ian") - 1]$, $\sigma'(id_t)\ ("pauths")[i] = \texttt{true}$, we can deduce $\sigma'(id_t)("pauths")[j] = \texttt{true}$.
By property of the stabilize relation, $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$ and $\texttt{comp}(id_t, "transition", gm_t, ipm_t, opm_t) \in d.cs$:

$$\sigma'(id_p)("pauths")[k] = \sigma'(id_{kj})\sigma'(id_t)("pauths")[j] = \texttt{true} \tag{C.55}$$

By property of the stabilize relation and $\texttt{comp}(id_p, "place", gm_p, ipm_p, opm_p) \in d.cs$:

$$\sigma'(id_p)("pauths")[k] = (\sigma'(id_p)("sm") \geq \texttt{vsots} + \sigma'(id_p)("oaw")[k]) \tag{C.56}$$

Let us define the `vsots` term as follows:

$$\texttt{vsots} = \sum_{i=0}^{k-1} \begin{cases} \sigma'(id_p)("oaw")[i] \texttt{ if } \sigma'(id_p)("otf")[i]. \\ \qquad\qquad\qquad\qquad \sigma'(id_p)("oat")[i] = \texttt{basic} \\ 0 \ otherwise \end{cases} \tag{C.57}$$

From (C.55) and (C.56), we can deduce that $\sigma'(id_p)("sm") \geq \texttt{vsots} + \sigma'(id_p)("oaw")[k]$.
Then, there are three points to prove:

A. $\boxed{s'.M(p) = \sigma'(id_p)("sm")}$

B. $\boxed{\omega = \sigma'(id_p)("oaw")[k]}$

C. $\boxed{\sum_{t_i \in Pr(t, fired)} pre(p, t_i) = \texttt{vsots}}$

See 1 for the remainder of the proof.

□

Lemma 47 (Falling Edge Equal Not Fired)**.** *For all sitpn, d, γ, Δ, σ_e, E_c, E_p, τ, s, s', σ, σ_i, $\sigma_\downarrow$, σ' that verify the hypotheses of Definition 27, then $\forall t, id_t$ s.t. $\gamma(t) = id_t$, $t \notin Fired(s') \Leftrightarrow \sigma'_t(''fired'') =$* `false`*.*

Proof. Proving the above lemma is trivial by appealing to Lemma Falling edge equal fired and by reasoning on contrapositives. □

Bibliography

[1] David Andreu, David Guiraud, and Guillaume Souquet. "A Distributed Architecture for Activating the Peripheral Nervous System". In: *Journal of Neural Engineering* 6.2 (Apr. 1, 2009), p. 026001. ISSN: 1741-2560, 1741-2552. DOI: 10.1088/1741-2560/6/2/026001. URL: https://iopscience.iop.org/article/10.1088/1741-2560/6/2/026001 (visited on 06/08/2020).

[2] Peter J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann, Oct. 7, 2010. 933 pp. ISBN: 978-0-08-056885-0. Google Books: XbZr8DurZYEC.

[3] 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.

[4] Yves Bertot and P. Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Berlin ; New York: Springer, 2004. 469 pp. ISBN: 978-3-540-20854-9.

[5] David C. Black et al. *SystemC: From the Ground Up, Second Edition*. Springer Science & Business Media, Dec. 18, 2009. 291 pp. ISBN: 978-0-387-69958-5. Google Books: Op2a6yiO9jwC.

[6] 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. URL: https://link.springer.com/chapter/10.1007/11813040_31 (visited on 05/25/2020).

[7] 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. URL: http://link.springer.com/10.1007/BF01383873 (visited on 09/18/2019).

[8] 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. URL: https://doi.org/10.1145/3385412.3385965 (visited on 05/05/2021).

[9] Timothy Bourke et al. "A Formally Verified Compiler for Lustre". In: (), p. 17.

[10] 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.

[11] Peter T. Breuer, Luis Sánchez Fernández, and Carlos Delgado Kloos. "A Functional Semantics for Unit-Delay VHDL". In: *Formal Semantics for VHDL*. Ed. by Carlos Delgado Kloos and Peter T. Breuer. Red. by Jonathan Allen. Vol. 307. Boston, MA: Springer US, 1995, pp. 43–70. ISBN: 978-1-4613-5941-8 978-1-4615-2237-9. DOI: 10.1007/978-1-4615-2237-9_3. URL: http://link.springer.com/10.1007/978-1-4615-2237-9_3 (visited on 09/09/2019).

[12] Peter T. Breuer, Luis Sánchez Fernández, and Carlos Delgado Kloos. "A Simple Denotational Semantics, Proof Theory and a Validation Condition Generator for Unit-Delay VHDL". In: *Formal Methods in System Design* 7.1 (Aug. 1, 1995), pp. 27–51. ISSN: 1572-8102. DOI: 10.1007/BF01383872. URL: https://doi.org/10.1007/BF01383872 (visited on 02/20/2020).

[13] Egon Börger, Uwe Glässer, and Wolfgang Muller. "A Formal Definition of an Abstract VHDL'93 Simulator by EA-Machines". In: *Formal Semantics for VHDL*. Ed. by Carlos Delgado Kloos and Peter T. Breuer. Red. by Jonathan Allen. Vol. 307. Boston, MA: Springer US, 1995, pp. 107–139. ISBN: 978-1-4613-5941-8 978-1-4615-2237-9. DOI: 10.1007/978-1-4615-2237-9_5. URL: http://link.springer.com/10.1007/978-1-4615-2237-9_5 (visited on 09/12/2019).

[14] 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.

[15] Jose R. Celaya, Alan A. Desrochers, and Robert J. Graves. "Modeling and Analysis of Multi-Agent Systems Using Petri Nets". In: *2007 IEEE International Conference on Systems, Man and Cybernetics*. 2007 IEEE International Conference on Systems, Man and Cybernetics. Oct. 2007, pp. 1439–1444. DOI: 10.1109/ICSMC.2007.4413960.

[16] 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. URL: https://doi.org/10.1145/1707801.1706312 (visited on 05/22/2020).

[17] Alain Colmerauer. "An Introduction to Prolog III". In: *Computational Logic* (1990), pp. 37–79. DOI: 10.1007/978-3-642-76274-1_2. URL: https://link.springer.com/chapter/10.1007/978-3-642-76274-1_2 (visited on 03/31/2020).

[18] 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.

[19] Maulik A. Dave. "Compiler Verification: A Bibliography". In: *ACM SIGSOFT Software Engineering Notes* 28.6 (Nov. 1, 2003), p. 2. ISSN: 01635948. DOI: 10.1145/966221.966235. URL: http://portal.acm.org/citation.cfm?doid=966221.966235 (visited on 05/22/2020).

[20] René David and Hassane Alla. "Petri Nets for Modeling of Dynamic Systems: A Survey". In: *Automatica* 30.2 (Feb. 1, 1994), pp. 175–202. ISSN: 0005-1098. DOI: 10.1016/0005-1098(94)90024-8. URL: https://www.sciencedirect.com/science/article/pii/0005109894900248 (visited on 06/17/2021).

[21] Frank Dederichs, Claus Dendorfer, and Rainer Weber. "Focus: A Formal Design Method for Distributed Systems". In: *Parallel Computer Architectures* (1993), pp. 190–202. DOI: 10.1007/978-3-662-21577-7_14. URL: https://link.springer.com/chapter/10.1007/978-3-662-21577-7_14 (visited on 03/31/2020).

[22] Michel Diaz. *Les Réseaux de Petri: Modèles Fondamentaux*. Hermès science publications, 2001.

[23] David Déharbe and Dominique Borrione. "Semantics of a Verification-Oriented Subset of VHDL". In: *Correct Hardware Design and Verification Methods*. Advanced Research Working Conference on Correct Hardware Design and Verification Methods. Springer, Berlin, Heidelberg, Oct. 2, 1995, pp. 293–310. DOI: 10.1007/3-540-60385-9_18. URL: https://link.springer.com/chapter/10.1007/3-540-60385-9_18 (visited on 04/01/2020).

[24] Gert Döhmen and Ronald Herrmann. "A Deterministic Finite-State Model for VHDL". In: *Formal Semantics for VHDL*. Ed. by Carlos Delgado Kloos and Peter T. Breuer. The Kluwer International Series in Engineering and Computer Science. Boston, MA: Springer US, 1995, pp. 170–204. ISBN: 978-1-4615-2237-9. DOI: 10.1007/978-1-4615-2237-9_7. URL: https://doi.org/10.1007/978-1-4615-2237-9_7 (visited on 03/02/2020).

[25] 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. Google Books: Ze60AwAAQBAJ.

[26] 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.

[27] Max Fuchs and Michael Mendler. "A Functional Semantics for Delta-Delay VHDL Based on Focus". In: *Formal Semantics for VHDL*. Ed. by Carlos Delgado Kloos and Peter T. Breuer. Red. by Jonathan Allen. Vol. 307. Boston, MA: Springer US, 1995, pp. 9–42. ISBN: 978-1-4613-5941-8 978-1-4615-2237-9. DOI: 10.1007/978-1-4615-2237-9_2. URL: http://link.springer.com/10.1007/978-1-4615-2237-9_2 (visited on 09/03/2019).

[28] K. G. W. Goossens. "Reasoning about VHDL Using Operational and Observational Semantics". In: *Correct Hardware Design and Verification Methods*. Advanced Research Working Conference on Correct Hardware Design and Verification Methods. Springer, Berlin, Heidelberg, Oct. 2, 1995, pp. 311–327. DOI: 10.1007/3-540-60385-9_19. URL: https://link.springer.com/chapter/10.1007/3-540-60385-9_19 (visited on 04/01/2020).

[29] David Guiraud et al. "An Implantable Neuroprosthesis for Standing and Walking in Paraplegia: 5-Year Patient Follow-Up". In: *Journal of Neural Engineering* 3.4 (Sept. 2006), pp. 268–275. ISSN: 1741-2552. DOI: 10.1088/1741-2560/3/4/003. URL: https://doi.org/10.1088/1741-2560/3/4/003 (visited on 06/11/2021).

[30] 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.

[31] Graham Hutton. "Introduction to HOL: A Theorem Proving Environment for Higher Order Logic by Mike Gordon and Tom Melham (Eds.), Cambridge University Press, 1993, ISBN 0-521-44189-7". In: *Journal of Functional Programming* 4.4 (Oct. 1994), pp. 557–559. ISSN: 1469-7653, 0956-7968. DOI: 10.1017/S0956796800001180. URL: https://www.cambridge.org/core/journals/journal-of-functional-programming/article/introduction-to-hol-a-theorem-proving-environment-for-higher-order-logic-by-gordon-mike-and-melham-tom-eds-cambridge-university-press-1993-isbn-0521441897/682CAD7058D7014549AE3F9580D0220B (visited on 04/22/2020).

[32] IEEE Computer Society et al. *IEEE Standard VHDL Language Reference Manual*. New York, N.Y.: Institute of Electrical and Electronics Engineers, 2000. ISBN: 978-0-7381-1948-9 978-0-7381-1949-6. URL: http://ieeexplore.ieee.org/lpdocs/epic03/standards.htm (visited on 09/16/2019).

[33] *IEEE/IEC 62142-2005 - IEC/IEEE International Standard - Verilog(R) Register Transfer Level Synthesis*. URL: https://standards.ieee.org/standard/62142-2005.html (visited on 06/30/2021).

[34] Mark P. Jones. *The Implementation of the Gofer Functional Programming System*. Yale University, Department of Computer Science, May 1994, p. 52.

[35] 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. URL: https://tel.archives-ouvertes.fr/tel-01766458 (visited on 02/10/2020).

[36] 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. URL: https://doi.org/10.1007/s10817-009-9155-4 (visited on 01/21/2020).

[37] David Long and Zane Scott. *A Primer for Model-Based Systems Engineering*. Lulu.com, 2011. 126 pp. ISBN: 978-1-105-58810-5. Google Books: pCaoAwAAQBAJ.

[38] 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. URL: https://doi.org/10.1145/3437992.3439916 (visited on 05/04/2021).

[39] 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.

[40] 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. URL: https://tel.archives-ouvertes.fr/tel-01704776 (visited on 02/10/2020).

[41] Gordon E. Moore. "Cramming More Components onto Integrated Circuits, Reprinted from Electronics, Volume 38, Number 8, April 19, 1965, Pp.114 Ff." In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (Sept. 2006), pp. 33–35. ISSN: 1098-4232. DOI: 10.1109/N-SSC.2006.4785860.

[42] T. Murata. "Petri Nets: Properties, Analysis and Applications". In: *Proceedings of the IEEE* 77.4 (Apr. 1989), pp. 541–580. ISSN: 1558-2256. DOI: 10.1109/5.24143.

[43] Serafín Olcoz. "A Formal Model of VHDL Using Coloured Petri Nets". In: *Formal Semantics for VHDL*. Ed. by Carlos Delgado Kloos and Peter T. Breuer. The Kluwer International Series in Engineering and Computer Science. Boston, MA: Springer US, 1995, pp. 140–169. ISBN: 978-1-4615-2237-9. DOI: 10.1007/978-1-4615-2237-9_6. URL: https://doi.org/10.1007/978-1-4615-2237-9_6 (visited on 03/02/2020).

[44] S. Owre et al. "A Tutorial on Using PVS for Hardware Verification". In: *Theorem Provers in Circuit Design*. International Conference on Theorem Provers in Circuit Design. Springer, Berlin, Heidelberg, Sept. 26, 1994, pp. 258–279. DOI: 10.1007/3-540-59047-1_53. URL: https://link.springer.com/chapter/10.1007/3-540-59047-1_53 (visited on 03/31/2020).

[45] S.L. Pandey, K. Umamageswaran, and P.A. Wilsey. "VHDL Semantics and Validating Transformations". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 18.7 (July 1999), pp. 936–955. ISSN: 1937-4151. DOI: 10.1109/43.771177.

[46] Volnei A. Pedroni. *Circuit Design with VHDL, Third Edition*. MIT Press, Apr. 14, 2020. 609 pp. ISBN: 978-0-262-35392-2. Google Books: fVzbDwAAQBAJ.

[47] Carl Adam Petri. "Kommunikation mit Automaten". In: *http://edoc.sub.uni-hamburg.de/informatik/v petri.pdf* (1962). URL: https://edoc.sub.uni-hamburg.de//informatik/volltexte/2011/160/ (visited on 06/10/2021).

[48] Ralf Reetz and Thomas Kropf. "A Flow Graph Semantics of VHDL: A Basis for Hardware Verification with VHDL". In: *Formal Semantics for VHDL*. Ed. by Carlos Delgado Kloos and Peter T. Breuer. The Kluwer International Series in Engineering and Computer Science. Boston, MA: Springer US, 1995, pp. 205–238. ISBN: 978-1-4615-2237-9. DOI: 10.1007/978-1-4615-2237-9_8. URL: https://doi.org/10.1007/978-1-4615-2237-9_8 (visited on 03/02/2020).

[49] 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. URL: https://link.springer.com/chapter/10.1007/3-540-45620-1_5 (visited on 06/08/2020).

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

[51] Krishnaprasad Thirunarayan and Robert L. Ewing. "Structural Operational Semantics for a Portable Subset of Behavioral VHDL-93". In: *Formal Methods in System Design* 18.1 (Jan. 1, 2001), pp. 69–88. ISSN: 1572-8102. DOI: 10.1023/A:1008786720393. URL: https://doi.org/10.1023/A:1008786720393 (visited on 03/02/2020).

[52] 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. URL: http://link.springer.com/10.1007/978-1-4615-2237-9_4 (visited on 09/12/2019).

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

[54] 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.

[55] 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. URL: http://www.sciencedirect.com/science/article/pii/S0164121214000727 (visited on 01/16/2020).

[56] 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. URL: https://doi.org/10.1007/s11704-015-4364-y (visited on 01/21/2020).