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]

UNIVERSITÉ
DE MONTPELLIER

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

For/Dedicated to/To my…

Chapter 1

Proving semantic preservation in HILECOP

In this chapter, I want to talk about/draw the attention to:

- the properties of comp. instances itfaces deduced from the transformation (in part "a detailled proof")
- To ease the reading, place p and PCI id_p, meaning that $\gamma(p) = id_p$.
- PCI=place component instance, TCI=transition component instance.

In this chapter, we present our semantic preservation theorem along with its 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 used in the proof, and some hints regarding the proof strategy. The full proof is available to the reader in Appendix **??**. The theorems and lemmas presented in this chapter will be refering to the lemmas of Appendix **??**. The structure of this chapter is the following one: in Section 1.1, we present our review of the literature pertaining to the proof of semantic preservation theorems for transformation functions; in Section, we detail our state similarity relation, i.e, the semantic bond between an SITPN and its $\mathcal{H}$-VHDL translation; in Section, we draw out our semantic preservation theorem; in Section, we detail a particularly tricky point of the proof related to the computation of fired transitions, and we show how it has led to a bug detection in HILECOP's code; in Section, we present some points of the mechanization of the proof verification with the Coq proof assistant.

1.1 Proofs of semantic preservation in the literature

In this section, we present the review of the literature pertaining to 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. 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 in three categories. Each category covers a specific case of transformation function; the three categories are:

- Compilers for generic programming languages
- Compilers for hardware description languages
- Model-to-model and model-to-text transformations

In the introduction of his article about CompCert [11], 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 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, or bisimulation, diagram. Figure shows the different kind of simulation diagrams possibly relating two programs. 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.

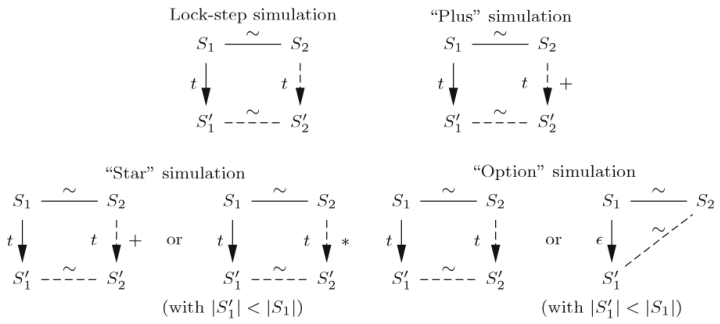

FIGURE 1.1: Simulation diagrams between source and target programs

Anyway, in the case where the source program execution succeeds, the theorem of semantic preservation takes this general form:

Consider a source program P_1 compiled into a target program P_2, a starting state S_1 for program P_1 and a starting state S_2 for program P_2 such that S_1 and S_2 are similar states w.r.t. the exhibited state comparison relation. If the execution of P_1 leads from state S_1 to state S_1', then there exists a 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 exhibited state comparison relation.

Compiler verification tasks 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 aim of the thesis is to perform compiler *verification* over the HILECOP methodology. Some of the works, cited afterwards, are more interested in compiler or transformation validation techniques than in verification. They are presented here for the sake of coverage.

Now that we have clarified the meaning of semantic preservation for GPL compilers, we state that this definition of semantic preservation holds also for 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?
- How are described the runtime state for the source and target representations?
- How is expressed the state comparison relation?
- How is stated the semantic preservation theorem?
- What is the employed proof strategy?

1.1.1 Compilers for generic programming languages

Taking the CompCert compiler as an example, the compilation pass from Clight programs to Cminor programs is described in [2, 11]. 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 is the same for all intermediate language involved in the CompCert compiler. The memory model consists in block references; each block has a lower and an upper bound. To access a 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 sof the source program to the memory states 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: the authors prove a simulation lemma for the Clight expressions, then for the 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 [11, 14], or compilers for functional languages [7, 15], compiler verfication proceeds as follows:

1. establish a relation between the memory models of the source and target languages, and between the global execution environments

2. prove simulation 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 [11]). As a consequence, the complexity of the relation for memory model comparison increases.

1.1.2 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 in 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 time-related semantics of the considered languages.

In [3, 5], the source languages are a subset of the BlueSpec specification language for hardware synthesis , and the target is an RTL representation of the circuit. The execution states of the source and target are based on registers. In [3], the execution state also hold 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 states that the registers hold the same values after the execution of source program and the resulting RTL circuit after one clock cycle.

In [4], the source language is a subset of Lustre and the target language is 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 refer to instances of other nodes. In the compilation process, each Lustre node is translated into an Obc class. An Obc class hold 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. Also, as we are dealing with hardwares, the judgments treat synchronous statements and combinational ones. On the side of the Obc language, the semantics define a function *step* that computes the execution 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 output streams o from input streams i, then the iterative execution of the *step* function for the corresponding Obc class builds every step of output streams o given the values of input streams i. The proof is done by induction over the clock step count, and by induction over the evaluation derivation of the nody instruction body.

In [12], 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 resembles the one of VHDL; the 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 by means of 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 [17], 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 (refering 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 [10], the authors verify a methodology to design hardware models with SystemC models. SystemC models describe hardware with modules; a module is a C++ class with ports, data members and methods. The methodology describes a transformation from SystemC into 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 modules 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 module is divided into a signal store, mapping signal to value, and a variable store, mapping variable to value. The execution state of an AsmL class is only composed of a variable store. The theorem of semantic preservation states that, after translation, a SystemC model has the same *observational* behavior than its corresponding AsmL class. What is compared between a SystemC model and its corresponding AsmL class through their observational behavior is the activity of the processes of the first one and the activity of the methods of the second one. Processes and methods must be active at the same delta cycles. Therefore, what is compared here are not the values that the execution states hold, but rather the activity of the source and target programs.

1.1.3 Model transformations

Regarding model transformations, a lot of works consider semantic preservation as the preservation of structural properties in the transformed model [1, 6, 13].

Still, there are many cases where the source model and th 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 simulation relation (see Figure 1.1).

In [8] and [16], the authors are interested in giving a translational semantics to a given model having itself a reference execution semantics. In [8], the source models are called xSpem models; they describe a set of activities exchanging resources and an holding an internal state. The target models are PNs. Both xSpem models and PNs have a state transition semantics. The state comparison is performed by checking the correspondence between each current status of the activities describe in an xSpem model and the marking of the PN. Then, the authors prove a bisimulation theorem, illustrated in Figure 1.2.

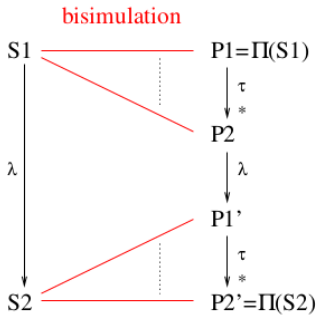

FIGURE 1.2: Bisimulation diagram relating an xSpem model execution and a Petri net execution

In Figure 1.2, one the right side of the diagram, i.e, the Petri net side, one can see that a Petri net possibly performs many internal actions (represented the arrow $\xrightarrow{\tau}{}^{*}$) before and after executing the computation step that is interest for the proof (i.e, action λ). Referring to the diagrams of Figure 1.1, this is a case of "star" simulation. The proof is performed by reasoning by induction on the structure of the xSpem model, and then by reasoning of the state transition semantics of xSpem models and PNs.

In [16], 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. 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 [9], 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 trully 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 straight-forward. The PN model has a 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 Petri net.

Discussions on transformations and proof strategies

In this thesis, we are interested in the verification of a semantic preservation property for a given transformation by proving a bisimulation theorem. To achieve this kind of proof task, the proceedings are quite similar, at least in the three cases of transformation presented above (i.e, GPLs compilation, HDLs compilation and model transformations). Even though the source and target languages or models are different from one case of transformation to the other, however, bisimulation theorems carry the same structure. The state comparison relation and the choice of the bisimulation diagram are the two angular stones of the process.

One can notice that when verifying the transformation of HDL programs, the bisimulation 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 this time-related computational step. This differs from the expression of bisimulation 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 bisimulation theorems are systematically done by induction over the semantics relation of the source languages. The semantics relation 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 enable compositional reasoning, i.e: to split the difficulty of proving the bisimulation theorem into simpler lemmas about the execution of simpler programs or simple model structures.

1.2 The state similarity relation

Before stating the 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 semantical matching means between an SITPN state and an $\mathcal{H}$-VHDL state?
2. when, in the course of the execution of an SITPN and an $\mathcal{H}$-VHDL design, does this semantical matching must hold?

We must relate the elements that constitute the execution state of an SITPN to the elements that constitute the execution state of an $\mathcal{H}$-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. 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 instances and signals on the $\mathcal{H}$-VHDL side. Thus, the state similarity relation expressing a semantical match between an SITPN state and an $\mathcal{H}$-VHDL design is defined as follows:

Add ref.

Definition 1 (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. $\forall p \in P, id_p \in Comps(\Delta)\ s.t.\ \gamma(p) = id_p,\ s.M(p) = \sigma(id_p)(\text{"}s\_marking\text{"})$.
2. $\forall t \in T_i, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t,$
   $\big(upper(I_s(t)) = \infty \wedge s.I(t) \leq lower(I_s(t)) \Rightarrow s.I(t) = \sigma(id_t)(\text{"}s\_time\_counter\text{"})\big)$
   $\wedge\big(upper(I_s(t)) = \infty \wedge s.I(t) > lower(I_s(t)) \Rightarrow \sigma(id_t)(\text{"}s\_time\_counter\text{"}) = lower(I_s(t))\big)$
   $\wedge\big(upper(I_s(t)) \neq \infty \wedge s.I(t) > upper(I_s(t)) \Rightarrow \sigma(id_t)(\text{"}s\_time\_counter\text{"}) = upper(I_s(t))\big)$
   $\wedge\big(upper(I_s(t)) \neq \infty \wedge s.I(t) \leq upper(I_s(t)) \Rightarrow s.I(t) = \sigma(id_t)(\text{"}s\_time\_counter\text{"})\big)$.
3. $\forall t \in T_i, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t, s.reset_t(t) = \sigma(id_t)(\text{"}s\_reinit\_time\_counter\text{"})$.
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 to the value of the *s_marking* signal inside the internal state of the place component instance id_p. 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. 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, a time counter will increment its value, depending on certain conditions, until it has reached 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 indefinitly. 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 have 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 will possibly be diverging (i.e, the time counter possibly continues its incrementation). 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 have 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 this 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 and correction (see Section 1.4), the state similarity relation of Definition 1.2 does not hold at the end of a clock cycle. The equality between reset orders (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 matching after the rising edge of the clock signal (i.e, in the middle of the clock cycle), and one that checks the semantic matching 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 2 (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 2 is similar to Definition 1 in all points, except for the value of conditions. A condition of an SITPN is implemented by an primary input port in the resulting $\mathcal{H}$-VHDL design. In $\mathcal{H}$-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.

The state similarity relation draws out a correspondence between the values hold by an SITPN state and the values of the signals declared in an $\mathcal{H}$-VHDL design state. However, to complete the proof of semantic preservation, we sometimes have to relate the value of signals to the value of expressions or predicates involved in the SITPN semantics. For instance, consider a given SITPN state s and a given $\mathcal{H}$-VHDL design state σ, and consider a transition t and its corresponding transition component instance 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 similiraty 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 3 (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 2) and*

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

Definition 3 extends Definition 2 with the correspondence of the sensitization of transitions and the value of signal *s_enabled*, and the computation of the boolean product of condition values and the value of signal *s_condition_combination*.

The state similarity relation after a falling edge is defined as follows:

Definition 4 (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. $\forall p \in P, id_p \in Comps(\Delta)\ s.t.\ \gamma(p) = id_p,\ s.M(p) = \sigma(id_p)(\text{"}s\_marking\text{"})$.
2. $\forall t \in T_i, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t,$
   $\big(upper(I_s(t)) = \infty \wedge s.I(t) \leq lower(I_s(t)) \Rightarrow s.I(t) = \sigma(id_t)(\text{"}s\_time\_counter\text{"})\big)$
   $\wedge \big(upper(I_s(t)) = \infty \wedge s.I(t) > lower(I_s(t)) \Rightarrow \sigma(id_t)(\text{"}s\_time\_counter\text{"}) = lower(I_s(t))\big)$
   $\wedge \big(upper(I_s(t)) \neq \infty \wedge s.I(t) > upper(I_s(t)) \Rightarrow \sigma(id_t)(\text{"}s\_time\_counter\text{"}) = upper(I_s(t))\big)$
   $\wedge \big(upper(I_s(t)) \neq \infty \wedge s.I(t) \leq upper(I_s(t)) \Rightarrow s.I(t) = \sigma(id_t)(\text{"}s\_time\_counter\text{"})\big)$.
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 4 is similar to Definition 1 except for the equality between reset orders and the value of signal *s_reinit_time_counter*. The extended version of the post falling edge state similarity relation is defined as follows:

Definition 5 (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 4) and*

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

Definition 5 extends Definition 4 by drawing out a correspondence between:

- the firability of transitions and the value of the signal $s\_firable$
- the firing status of transitions (i.e, transitions are fired or not) and the value of the output port $fired$
- the sum of tokens consumed by the firing process and the value of the signal $s\_output\_token\_sum$
- the sum of tokens produced by the firing process and the value of the signal $s\_input\_token\_sum$

1.3 Behavior Preservation Theorem

In this section, we will lay out the major theorems and lemmas stating that the HILECOP transformation function is semantic preserving. We will also present the written proofs for these theorems and lemmas.

1.3.1 Proof notations

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

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

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

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

During a proof, we constantly refer to the names of the constants and signals declared in the $\mathcal{H}$-VHDL place and transition designs. Some constants and signals have very long names, and therefore we use aliases to refer to them in the following proofs. Table **??** gives the full correspondence between constants and signals, and their aliases.

1.3.2 Preliminary definitions

We define here some relations that are necessary to formalize the 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 mimick 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 6 states that the execution environment for a given SITPN and the simulation environment for a given $\mathcal{H}$-VHDL design are similar.

Definition 6 (Similar environments)**.** *For a given $sitpn \in SITPN$, a $\mathcal{H}$-VHDL design $d \in design$, a design store $\mathcal{D} \in entity\text{-}id \nrightarrow design$, an elaborated version $\Delta \in ElDesign(d, \mathcal{D})$ 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 6 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 6, 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. The relation presented in Definition 7 permits to compare such traces.

Definition 7 (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:*

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

In Definition 7, 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 and then by a falling edge. 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 a 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 8.

Definition 8 (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.

1.3.3 The behavior preservation theorem

Theorem 1 states that the HILECOP transformation is semantic preserving when the input model is a well-defined SITPN. As a complementary task, we could show that if the transformation function returns a couple design and binder, and not an error, then the input SITPN is well-defined.

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 Elaboration, Initialization and Simulation, let us take an elaborated design Δ, two design states $\sigma_e, \sigma_0 \in \Sigma(\Delta)$, and a simulation trace $\theta_\sigma \in$ such that:

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

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

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

Rewriting the goal with (1.1):

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

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

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

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 Full bisimulation.

□

To prove Theorem 1, we must first prove that for all $\mathcal{H}$-VHDL design returned by the transformation function, there exists an elaborated version of it (Theorem Elaboration); then, we must prove that we can always build a simulation trace respecting the $\mathcal{H}$-VHDL simulation relation over τ simulation cycles (Theorem Initialization and Simulation). Finally, we can establish that the behaviors are similar by comparing the respective SITPN execution and $\mathcal{H}$-VHDL simulation traces. In this thesis, we are focusing on the proof that the execution/simulation traces are similar. For now, we choose to disregard the proof of theorems Elaboration, Initialization and Simulation stating the existence of an elaborated design and of a simulation trace for all $\mathcal{H}$-VHDL design returned by the HILECOP transformation function.

Theorem 2 (Elaboration). *For all $sitpn \in SITPN, d \in design, \gamma \in WM(sitpn, d)$ 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 (Initialization). *For all $sitpn \in SITPN, d \in design, \gamma \in WM(sitpn, d), \Delta \in ElDesign(d, \mathcal{D}_{\mathcal{H}})$, $\sigma_e \in \Sigma(\Delta)$ 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 (Simulation). *For all* $sitpn \in SITPN, d \in design, \gamma \in WM(sitpn, d), \Delta \in ElDesign(d, \mathcal{D}_{\mathcal{H}})$, $\sigma_e, \sigma_0 \in \Sigma(\Delta)$ *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 $E_p \in (\mathbb{N} \times \{\uparrow, \downarrow\}) \to Ins(\Delta) \to value$, *and simulation cycle count* $\tau \in \mathbb{N}$, *there exists s 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*:* $\mathcal{D}_{\mathcal{H}}, E_p, \Delta, \tau, \sigma_0 \vdash \text{d.cs} \to \theta_\sigma$

1.3.4 The bisimulation theorem

Here, we present the bisimulation theorem. The bisimulation theorem states that if an SITPN and its corresponding $\mathcal{H}$-VHDL design are executed/simulated over τ execution/simulation cycles, then the produced traces are semantically similar, i.e they verify the full execution trace similarity relation of Definition 8. In this thesis, we proved this particular theorem, and as said before, we left the proofs of Theorems Elaboration, Initialization and Simulation for later. We chose to focus our work on the bisimulation theorem, because it directly adresses the semantic preservation property of HILECOP's transformation function.

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.*

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

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

Proof. Given all the above variables and 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 **??**, 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:

  $E_p, \Delta, \tau, \sigma_0 \vdash d.cs \xrightarrow{\uparrow,\downarrow} \sigma, \sigma'$ 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 **??**, 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 **??**, we have $\gamma, E_c, \tau \vdash s_0 \overset{\uparrow}{\approx} \sigma$.

     By definition of $\gamma, E_c, \tau \vdash s_0 \overset{\uparrow}{\approx} \sigma$, we can show $\gamma, E_c, \tau \vdash s_0 \overset{\uparrow}{\sim} \sigma$.

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

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

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

  4. $\boxed{\gamma \vdash \theta \overset{\uparrow}{\sim} \theta''.}$

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

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

□

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 1); this is done by appealing to Lemma **??**. 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 **??**. The detailled proofs for Lemmas **??** and **??** are given in Sections **??** and **??**.

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 $\gamma \vdash \theta_s \overset{\updownarrow}{\sim} \theta_\sigma$.

Proof. Given all the above variables and assuming the above hypotheses, let us show $\boxed{\gamma \vdash \theta_s \overset{\updownarrow}{\sim} \theta_\sigma}$. 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:

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

  and $\theta_s = s' :: s'' :: \theta$ and $\theta_\sigma = \sigma' :: \sigma'' :: \theta'$.

  Then, the new goal is: $\boxed{\gamma \vdash (s' :: s'' :: \theta) \overset{\updownarrow}{\sim} (\sigma' :: \sigma'' :: \theta')}$.

  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 Falling edge, 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 Falling edge and Rising edge, we have $\gamma, E_c, \tau \vdash s' \overset{\downarrow}{\approx} \sigma'$.

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

3. $\boxed{\gamma \vdash \theta \overset{\uparrow}{\sim} \theta'}$

   We can apply the induction hypothesis with $s = s''$, $\sigma = \sigma''$, $\theta_s = \theta$ and $\theta_\sigma = \theta'$. Then, what is left to prove is: $\boxed{\gamma \vdash s'' \overset{\downarrow}{\approx} \sigma''}$

   Appealing to Lemmas Falling edge and Rising edge, 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 1.3. 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 Rising edge. First, we assume that the starting SITPN state and the starting $\mathcal{H}$-VHDL design state verify the full post falling edge state similarity relation at the beginning of the clock cycle (i.e, $s \overset{\downarrow}{\approx} \sigma$ in Figure 1.3). Then, Lemma Rising edge 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 Falling edge. 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 1.3). Then, Lemma Rising edge 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.

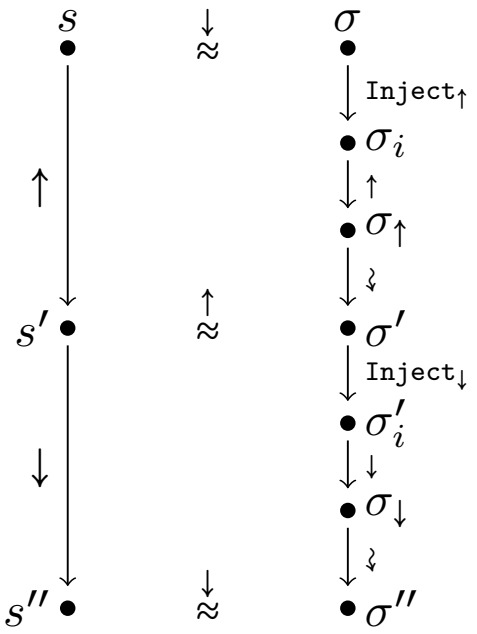

FIGURE 1.3: Bisimulation diagram over one clock cycle for a source SITPN and a target $\mathcal{H}$-VHDL design.

Here, we present Lemma Rising edge and Lemma Falling edge, along with their proofs. In the two lemmas, we added an extra hypothesis about the starting state of the $\mathcal{H}$-VHDL design: $\mathcal{D}_{\mathcal{H}}, \Delta, \sigma \vdash \text{d.cs} \xrightarrow{comb} \sigma$. The 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 1.4 for more details about combinational equations).

To prove Lemmas Rising edge and Falling edge, 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 **??**. 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 detailled in Section 1.4.

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. $\forall t \in T_i, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t,$
   $\big(upper(I_s(t)) = \infty \wedge s'.I(t) \leq lower(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(\text{"s\_time\_counter"})\big)$
   $\wedge\big(upper(I_s(t)) = \infty \wedge s'.I(t) > lower(I_s(t)) \Rightarrow \sigma'(id_t)(\text{"s\_time\_counter"}) = lower(I_s(t))\big)$
   $\wedge\big(upper(I_s(t)) \neq \infty \wedge s'.I(t) > upper(I_s(t)) \Rightarrow \sigma'(id_t)(\text{"s\_time\_counter"}) = upper(I_s(t))\big)$
   $\wedge\big(upper(I_s(t)) \neq \infty \wedge s'.I(t) \leq upper(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(\text{"s\_time\_counter"})\big)$.
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. $\forall f \in \mathcal{F}, id_f \in Outs(\Delta)\ s.t.\ \gamma(f) = id_f,\ s'.ex(f) = \sigma'(id_f)$.
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 Lemma **??** to solve 1.

– Apply Lemma **??** lemma to solve 2.

– Apply Lemma **??** to solve 3.

– Apply Lemma **??** to solve 4.

– Apply Lemma **??** to solve 5.

– Apply Lemma **??** to solve 6.

– Apply Lemma **??** to solve 7.

– Apply 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. $\forall p \in P, id_p \in Comps(\Delta)\ s.t.\ \gamma(p) = id_p,\ s'.M(p) = \sigma'(id_p)(\text{"}s\_marking\text{"})$.
2. $\forall t \in T_i, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t,$
   $(upper(I_s(t)) = \infty \wedge s'.I(t) \leq lower(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(\text{"}s\_time\_counter\text{"}))$
   $\wedge (upper(I_s(t)) = \infty \wedge s'.I(t) > lower(I_s(t)) \Rightarrow \sigma'(id_t)(\text{"}s\_time\_counter\text{"}) = lower(I_s(t)))$
   $\wedge (upper(I_s(t)) \neq \infty \wedge s'.I(t) > upper(I_s(t)) \Rightarrow \sigma'(id_t)(\text{"}s\_time\_counter\text{"}) = upper(I_s(t)))$
   $\wedge (upper(I_s(t)) \neq \infty \wedge s'.I(t) \leq upper(I_s(t)) \Rightarrow s'.I(t) = \sigma'(id_t)(\text{"}s\_time\_counter\text{"}))$.
3. $\forall c \in \mathcal{C}, id_c \in Ins(\Delta)\ s.t.\ \gamma(c) = id_c,\ s'.cond(c) = \sigma'(id_c)$.
4. $\forall a \in \mathcal{A}, id_a \in Outs(\Delta)\ s.t.\ \gamma(a) = id_a,\ s'.ex(a) = \sigma'(id_a)$.
5. $\forall f \in \mathcal{F}, id_f \in Outs(\Delta)\ s.t.\ \gamma(f) = id_f,\ s'.ex(f) = \sigma'(id_f)$.
6. $\forall t \in T, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t, t \in Firable(s') \Leftrightarrow \sigma'(id_t)(\text{"}s\_firable\text{"}) = \texttt{true}$.
7. $\forall t \in T, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t, t \notin Firable(s') \Leftrightarrow \sigma'(id_t)(\text{"}s\_firable\text{"}) = \texttt{false}$.
8. $\forall t \in T, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t, t \in Fired(s') \Leftrightarrow \sigma'(id_t)(\text{"}fired\text{"}) = \texttt{true}$.
9. $\forall t \in T, id_t \in Comps(\Delta)\ s.t.\ \gamma(t) = id_t, t \notin Fired(s') \Leftrightarrow \sigma'(id_t)(\text{"}fired\text{"}) = \texttt{false}$.
10. $\forall p \in P, id_p \in Comps(\Delta)\ s.t.\ \gamma(p) = id_p, \sum_{t \in Fired(s')} pre(p,t) = \sigma'(id_p)(\text{"}s\_output\_token\_sum\text{"})$.
11. $\forall p \in P, id_p \in Comps(\Delta)\ s.t.\ \gamma(p) = id_p, \sum_{t \in Fired(s')} post(t,p) = \sigma'(id_p)(\text{"}s\_input\_token\_sum\text{"})$.

Each point is proved by a separate lemma:

– Apply Lemma **??** to solve 1.

– Apply Lemma **??** to solve 2.

– Apply Lemma **??** to solve 3.

– Apply Lemma **??** to solve 4.

– Apply Lemma **??** to solve 5.

– Apply Lemma **??** to solve 6.

– Apply Lemma **??** to solve 7.

– Apply Lemma Falling edge equal fired to solve 8.

– Apply Lemma **??** to solve 9.

– Apply Lemma **??** to solve 10.

– Apply Lemma **??** to solve 11.

□

1.4 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 Lemma Falling edge equal fired, involved in the proof of Lemma Falling edge, to illustrate our demonstration technics. The proof of Lemma Falling edge equal fired has been one tricky part of the proof, and therefore, it is worth to be mentioned. Also, it has led to a bug detection. We give a full account on this bug detection, and on how we manage to correct it, at the end of the section.

1.4.1 An accompanied journey along the proof

The proof of Lemma 4 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 end of the falling edge phase, or 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 9 (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* $\Delta, \sigma_{\downarrow} \vdash \mathrm{d.cs} \xrightarrow{\rightsquigarrow} \sigma'$
- *State* σ *is a stable design state:* $\mathcal{D}_{\mathcal{H}}, \Delta, \sigma \vdash \mathrm{d.cs} \xrightarrow{comb} \sigma$

The hypotheses of Definition 9 are used in all the lemmas expressing some properties about the falling edge phase. Therefore, Definition 9 enables the conciser expression of these lemmas. Then, we can express Lemma Falling edge equal fired:

Lemma 4 (Falling edge equal fired)**.** *For all* $sitpn, d, \gamma, \Delta, \sigma_e, E_c, E_p, \tau, s, s', \sigma, \sigma_i, \sigma_{\downarrow}, \sigma'$ *that verify the hypotheses of Def. 9, 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 transition component instance id_t in the output $\mathcal{H}$-VHDL design d. Transition t and transition component instance id_t are bound together through the γ binder returned by the transformation function. This means that the component instance 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:

$$t \in Fired(s') \Leftrightarrow \sigma'(id_t)(\text{"}fired\text{"}) = \texttt{true}.$$

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 **??**) 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 **??**) and sensitized by the *residual* marking at the considered SITPN state. Figure 1.4 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.

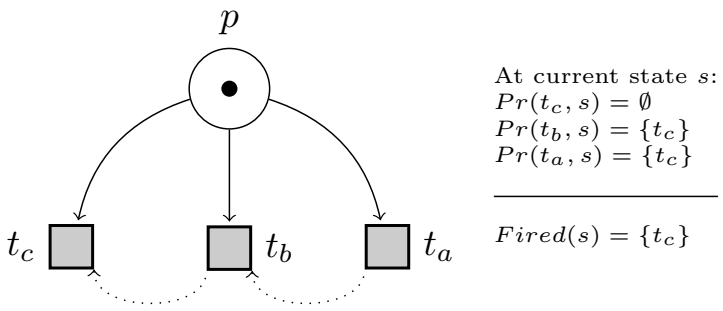

FIGURE 1.4: The set of fired transitions for an example SITPN at a given SITPN state s; on the right side, the dotted arrows indicates the priority relation between the three transitions (t_c is the top-priority transition); on the left side, each transition is associated to its Pr set which are necessary 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 1.4). The condition of the priority relation being a strict order over the set of transitions is part of the definition of a well-defined SITPN (see Definition **??**). By definition, top-priority transitions have an empty Pr set. 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 1.4, 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 transition component instances. 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 transition component instance 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 1.5 shows a part of the transition design architecture describing how the `fired` port is connected to the other internal signals.

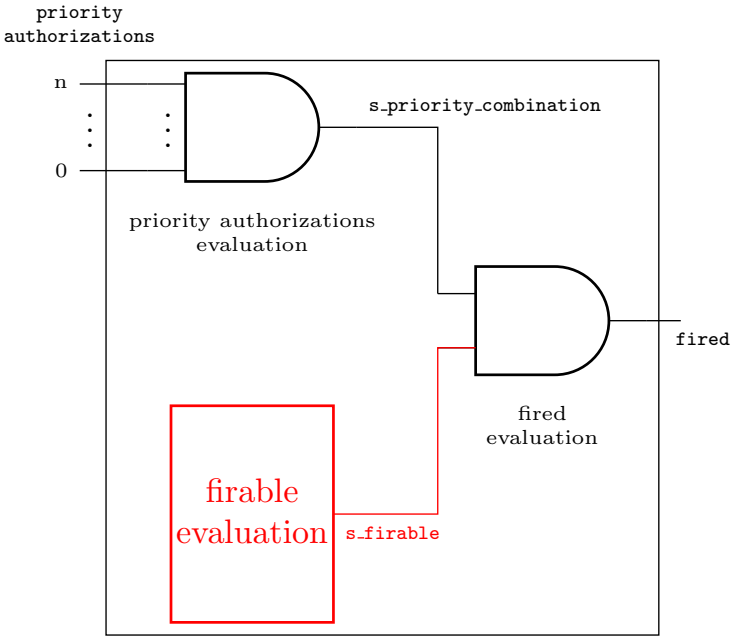

FIGURE 1.5: Wiring of the `fired` output port in the transition design architecture; on the left side is the input interface of the transition design; on the right side is the output interface of the transition design, with the `fired` port; in red are the parts of the architecture that depend on synchronous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

In Figure 1.5, the labels underneath the *and* logic ports and inside the block denote the names of the processes defined in the transition design architecture as VHDL code. As a matter of fact, Figure 1.5 is a 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. Considering a transition component instance id_t in a top-level design d; a state σ denotes the current stable state of d (remember that combinational equation hold when the signal values are stable); state $\sigma(id_t)$ denotes the internal state of the transition component instance id_t at state σ; thus, the `fired` port equation at σ is:

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

Equation (1.2) 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 1 (Signals and combinational equations)**.** *In the proceeding of the proof, a lot of combinational equations are established (e.g, Equation (1.2)). These equations link 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 combinational equation is always the result of a signal assignment statement happening inside the statement body of a process. For instance, in the transition design, the `fired_evaluation` process, presented in Listing 1.1, assigns the*

[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.

`fired` output port. Reasoning on the `fired_evaluation` process statement body and on the $\mathcal{H}$-VHDL semantics rules permits us to deduce Equation (1.2).

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

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

Listing 1.2 presents the `priority_authorizations_evaluation` *process, responsible for the assignment of the* `s_priority_combination` *in the transition design.*

```
priority_authorization_evaluation : process(priority_authorizations)
   variable v_priority_combination : std_logic;
begin
   v_priority_combination := '1';

   for i in 0 to input_arcs_number − 1 loop
     v_priority_combination := v_priority_combination and priority_authorizations(i);
   end loop;

   s_priority_combination ⇐ v_priority_combination; -- Assignment of the result
end process priority_authorization_evaluation;
```

LISTING 1.2: The `priority_authorizations_evaluation` process in the transition design's architecture. The local variable `v_priority_combination` accumulates the product of the `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 (1.3) gives the combinational equation deduced from the execution of the `priority_authorizations_evaluation` *process for a given transition component instance 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.*

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

In Equation (1.3), "spc" is an alias[2] *for the* `s_priority_combination` *signal. The* `for` *loop of the* `priority_authorization_evaluation` *process has been converted into a product expression where the index i follows the bounds of the loop. The* `priority_authorizations` *signal is an input port of type* `array`*, thus we use the bracketed notation $a[i]$ to access the element of index i in array a. Also, we know that* `input_arcs_number` *identifies a generic constant of the transition design, thus, we can retrieve its value in the elaborated design $\Delta(id_t)$.*

In the proofs laid out in Appendix **??** *and in this chapter, we do not detail how the execution of processes' statement body permit to deduce combinational equations. We find that the proofs are easier to follow without entering in so much details. We let aside the task of proving that these equations hold until the time*

[2] We will be using a lot of aliases for the names of signals in the proofs to come. Table **??** gives the correspondence between signals and constants names and their aliases.

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). Add ref.

Now that we know which combinational equation is attached to the value of the output port `fired` for a given transition component instance, 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 **??**, 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; then, 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:
$\forall i \in [0, \Delta(id_t)("ian") - 1],\ \sigma'(id_t)("pauths")[i] = \texttt{true}.$
For a given $i \in [0, \Delta(id_t)("ian") - 1]$, let us show that $\sigma'(id_t)("pauths")[i] = \texttt{true}.$ As shown in Figure 1.5, 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 transition component instance 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 place component instance id_p. Here, id_p is the place component instance bound to place p through the γ binder, i.e. we have $\gamma(p) = id_p$. Figure 1.6 shows the connection of the `priority_authorizations` port between the component instances id_p and id_t.

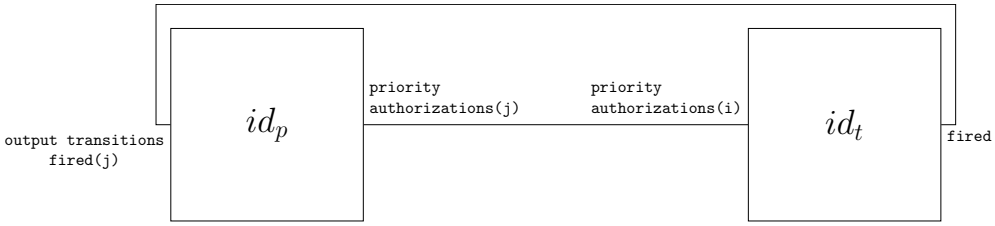

FIGURE 1.6: Connection of the j-th element of the `priority_authorizations` output port of a place component instance id_p to the i-th element of the `priority_authorizations` input port of a transition component instance id_t; also the `fired` output port of id_t is connected to the j-th element of the `output_transitions_fired` input port of id_p.

Thus, we know that the value of the i-th element of the `priority_authorizations` input port of id_t is bound to the value of the j-th element of the `priority_authorizations` output port of id_p. Therefore, we must now show that $\boxed{\sigma'(id_p)(\text{"}pauths\text{"})[j] = \texttt{true}.}$ 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 1.7 illustrates the wiring of the `priority_authorizations` output port in a place design.

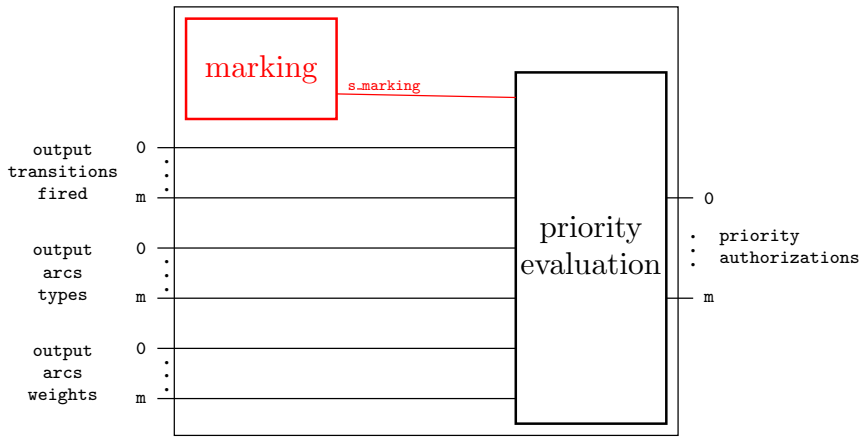

FIGURE 1.7: Wiring of the `priority_authorizations` output port in the architecture of the place design; the input port interface is on the left side and the output port interface is on the right side; the synchronous parts are in red and the combinational ones are in black.

Figure 1.7 shows that the value of the elements of the `priority_authorizations` output port is computed by the `priority_evaluation` process. This process reads the value of the `s_marking` signal, assigned by the synchronous process `marking`. It also reads the value of the input ports `output_transitions_fired`, `output_arcs_types` and `output_arcs_weights`. In Figure 1.7, the ports of the input and output interface are composite ports (i.e., of the array type) with an upper bound index equal to `m`. The number `m` is equal to the expression `output_arcs_number` − 1, where `output_arcs_number` is a generic constant of the place design. The value of the `output_arcs_number` constant is set at the generation of the generic map of a place component instance id_p, and is equal to the number of output transitions of place p. Listing 1.3 presents the code of the `priority_evaluation` process.

```
1  priority_evaluation : process (output_transitions_fired, s_marking, output_arcs_types,
       output_arcs_weights)
2    variable v_saved_output_token_sum : local_weight_t;
3  begin
4    v_saved_output_token_sum := 0;
5
6    for k in 0 to output_arcs_number − 1 loop
7
8      priority_authorizations(k) ⇐ (s_marking − v_saved_output_token_sum >=
           output_arcs_weights(k));
9
10     if (output_transitions_fired(k) = '1') and (output_arcs_types(k) = arc_t(BASIC)) then
11       v_saved_output_token_sum := v_saved_output_token_sum + output_arcs_weights(k);
12     end if;
13
14   end loop;
15 end process priority_evaluation;
```

LISTING 1.3: The `priority_evaluation` process in the place design's architecture.

In the statement body of the `priority_evaluation` process, each 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 transition component instance 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':

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

Then, rewriting the goal with Equation (1.4), the new goal is:

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

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:

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

Here, the term `vsots` corresponds to the value of the local variable `v_saved_output_token_sum` at the moment of the assignment in the `for` loop. By looking at the code of Listing 1.3 (Lines 10 to 12), we can deduce the value of the `vsots`:

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

The `vsots` term is equal to the sum of the output arc weights for all TCIs with an index comprised between 0 and $j-1$ that are `fired` and linked to the place by a `basic` input arc. The order of the indexes from 0 to `output_arcs_number` -1 reflects the priority order of the output transitions. Therefore, the indexes from 0 to $j-1$ are linked to transitions with a higher firing priority than the transition connected to the index j. Figure 1.8 takes back the SITPN of Figure 1.4 and illustrates 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).

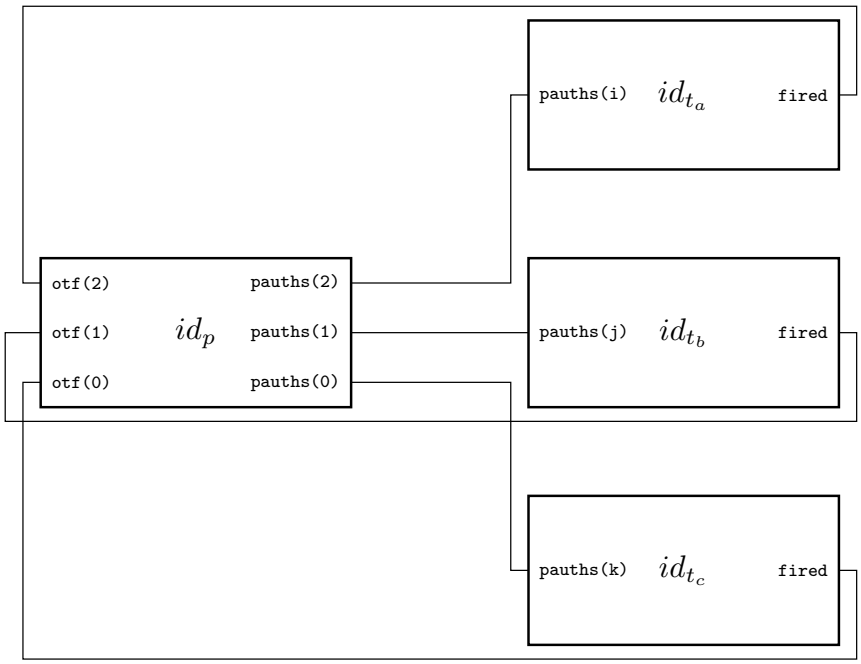

FIGURE 1.8: Connection between the `priority_authorizations` output port of PCI id_p and the `priority_authorizations` input port of TCIs id_{t_a}, id_{t_b} and id_{t_c}, and between the `output_transitions_fired` input port of id_p and the `fired` ports of id_{t_a}, id_{t_b} and id_{t_c}. `pauths` stands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

In Figure 1.8, the indexes in the interface of id_p respect the priority order of the output transitions. The index increases as the priority level of the connected TCI decreases. Thus, id_{t_c} is connected to index 0 as transition t_c is the top-priority transition in the output transitions of p.

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 **??**, 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:

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

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 1.8), 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 1.

Algorithm 1: 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 1 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 set T_s have been elected for firing or have been discarded. At Line 4, function `GetTopPriorityTransitions` returns the set tp 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 $T_s \subseteq T$. 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 1 through the definition of the *IsFiredSet* relation given in Definition 10. Definition 11 states that a given transition is fired in relation to the *IsFiredSet* relation.

Definition 10 (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 11 (Fired). *A transition $t \in T$ is said to be fired at the SITPN state $s = <M, I, reset_t, ex, cond>$, iff there exists a subset $fset \subseteq T$ such that $IsFiredSet(s, fset)$ and $t \in fset$.*

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

The definition of the *IsFiredSet* relation depends on the definition of the *IsFiredSetAux* relation given in Definition 12. 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 **??**.

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

IsFiredSetAuxNil
$$\frac{}{IsFiredSetAux(s, fired, \emptyset, fired)}$$

IsFiredSetAuxCons
$$\frac{IsTopPrioritySet(T_s, tp) \quad ElectFired(s, fired, tp, fired') \quad IsFiredSetAux(s, fired', T_s \setminus tp, fset)}{IsFiredSetAux(s, fired, T_s, fset)}$$

The *IsFiredSetAux* relation depends on the definitions of the *IsTopPrioritySet* and the *ElectFired* relations given in Definition 13 and 13. The *IsTopPrioritySet* is a relational implementation of the `GetTopPriorityTransitions` function appearing at Line 4 of Algorithm 1. The *ElectFired* relation is a relational implementation of the `ElectFired` function appearing at Line 5 of Algorithm 1.

Definition 13 (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 14 (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
$$\frac{\exists t' \in T_a \cup T_b \text{ s.t. } t' \succ t \quad IsTopPrioritySetAux(T_a, \{t\} \cup T_b, tp, tp')}{IsTopPrioritySetAux(\{t\} \cup T_a, T_b, tp, tp')}$$

Definition 15 (ElectFired). *The ElectFired relation is defined by the following rules:*

ElectFiredEmpty
$$\frac{}{ElectFired(s, fired, \emptyset, fired)}$$

ELECTFIRED⊥

$$\neg\Big(t \in Firable(s) \wedge t \in Sens(s.M - \sum_{t_i \in Pr(t,fired)} pre(t_i))\Big)$$

$$\frac{ElectFired(s, fired, tp, fired')}{ElectFired(s, fired, \{t\} \cup tp, fired')} \; Pr(t, fired) = \{t' \mid t' \succ t \wedge t' \in fired\}$$

ELECTFIRED⊤

$$t \in Firable(s) \qquad t \in Sens(s.M - \sum_{t_i \in Pr(t,fired)} pre(t_i))$$

$$\frac{ElectFired(s, \{t\} \cup fired, tp, fired')}{ElectFired(s, fired, \{t\} \cup tp, fired')} \; Pr(t, fired) = \{t' \mid t' \succ t \wedge t' \in fired\}$$

1.4.2 A report on a bug detection

In the previous section, we showed the equivalence between fired transitions and `fired` port values at the end of the falling edge phase. In a previous definition of the SITPN state, preceding the bug detection, the set of fired transitions was a member of the SITPN state record. For a given $sitpn \in SITPN$, we defined an SITPN state s by the record $s = <Fired, M, I, cond, ex>$ where $Fired$ was the set of fired transitions. The $Fired$ set was involved in the computation of time counter values during the falling edge phase. Thus, we needed the proof that the equivalence between the set of fired transitions and the value of the `fired` ports was effective at the end of the rising 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. The consequence was a divergence between the value of time counters and the value of the `s_time_counter` signals. Figure 1.9 shows a case of divergence between time counters and `s_time_counter` signals values in the course of an execution.

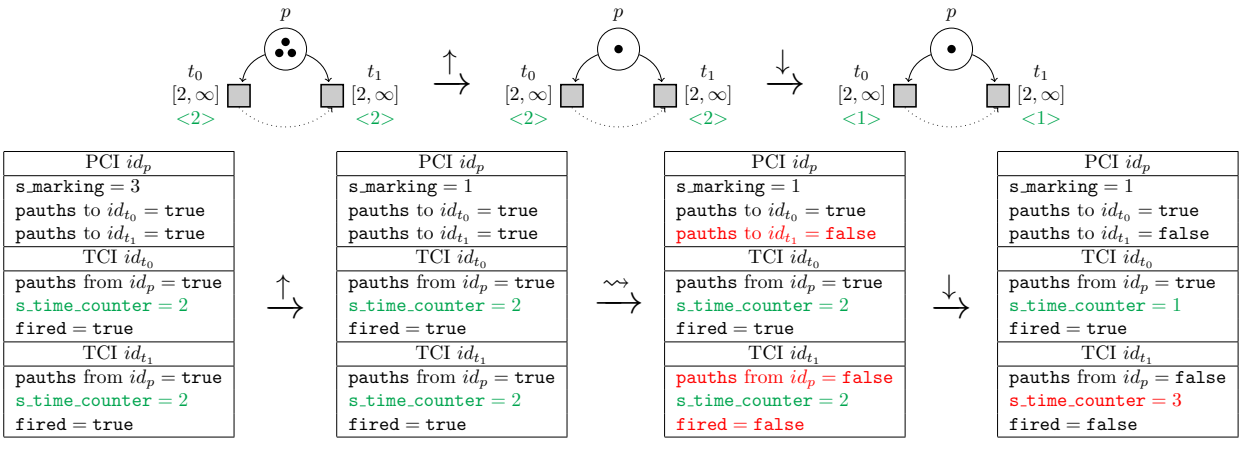

FIGURE 1.9: Bug detection: divergence between the value of time counters and the value of the `s_time_counter` signals due to the loss of the firing status information during the stabilization phase; the value of time counters and of the `s_time_counter` signals are in green; the value of diverging signals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

In Figure 1.9, during the stabilization phase coming right after the rising edge of the clock, the value of the `fired` port of TCI id_{t_1} passes to `false`. After the update of the `s_marking` signal value during the rising edge phase, PCI id_p computes new priority authorizations for its output TCIs. As the marking is only sufficient to fire transition t_0 but not transition t_1, PCI id_p indicates to TCI id_{t_1} that it no longer has the authorization to fire. Consequently, through the connection of `priority_authorizations` ports, the value of the `fired` port of id_{t_1} is set to `false`. Following the rules of the SITPN semantics, on the next falling edge, the value of time counters must be reset for transition t_0 and t_1, because both were fired at the previous rising edge. As a part of the behavior of a TCI, the `time_counter` process, executed at the falling edge of the clock, resets the value of the `s_time_counter` signal given that the value of the `fired` port is true. Thus, as the value of the `fired` port of TCI id_{t_1} is false at the falling edge, the `time_counter` process increments the value of the `s_time_counter` signal instead of resetting its value. The consequence is a divergence between the value of the time counter of transition t_1 and the value of the `s_time_counter` signal in TCI id_{t_i}.

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. Thus, 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 **??**). 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 **??** and **??** about the equivalence of time counters).

1.5 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 **??** lemma. However, the effort to achieve this thirty per cent of the verification amounts to three months of work. In this section, we give metrics to measure the gap between the size of the "paper" proof (see Appendix **??**) and the size of the computer-checked proof written in Coq. We point out some of the reasons that may explain the gap, and comment some employed techniques to reduce the size of proof scripts. As a remainder, the full code including specifications and proof scripts is available at

add ref to git repo

.

Listing 1.4 presents the Coq implementation of Theorem 5 along with the sequence of tactics constituting its proof. We also declared the Behavior preservation theorem, and the Elaboration, Initialization, Simulation theorems as axioms in the `Soundness.v` file under the `soundness` folder of the Git repository.

```
1  Theorem sitpn2hvhdl_full_bisim :
2    forall τ sitpn decpr id_ent id_arch E_c θ_s d E_p 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 1.4: Coq implementation of the Full bisimulation theorem and the mechanized version of its proof.

The proof laid out in Listing 1.4 follows the structure of the informal proof of Theorem 5. First, we perform case analysis on the structure of the τ variable through the `destruct` tactic. Then, the `intros` $*$ introduces all universally-bound variables in the proof context. Then, at Lines 25 and 26, we use a variant of the `inversion` tactic (i.e. `inversion_clear`) to unfold the definition of the SITPN full execution relation and the $\mathcal{H}$-VHDL full simulation relations. The number passed as an argument to the `inversion_clear` tactic refers to the index of the premise in the arrow-separated list of premises constituting the declaration of the theorem. At Line 31, we perform pattern matching on the proof context and on the conclusion to be proved. This permits to identify the hypothesis associated to the $\mathcal{H}$-VHDL simulation relation; we name it `Hsimloop`. This hypothesis has been

introduced in the context of the proof as a side effect of the inversion `tactic` used at Line 26. Then, we introduce in the proof context new hypotheses based on the definition of the `Hsimloop` hypothesis (i.e. the definition of the $\mathcal{H}$-VHDL simulation relation) by invoking `inversion_clear` tactic on `Hsimloop`. Then, the `constructor` tactic builds sub-goals to be proved based on the definition of the full trace similarity relation (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 1.4 is representative of our strategy to keep our mechanized proofs robust to change. The robustness criterion is important for multiple reasons. First, in the proceeding of the proof, we can always realize that some case is missing in the expression of the transformation function or discover that the semantics of the SITPNs or the $\mathcal{H}$-VHDL language is incomplete or incorrect. Therefore, we want to structure our proofs in a way that will lower the impact of correcting the transformation function or completing the semantics. Second, we know that the SITPN structure and the $\mathcal{H}$-VHDL code of the place and transition designs will be evolving in the future. Therefore, we want to be able to adapt our proofs with a minimum effort. To reach robustness to change, we follow the indications laid out in [7]. Mainly, we make an important use of the pattern matching constructs, such as `lazymatch` or `match`, to seek hypotheses in the current proof context. Also, we build hint databases and rely as much as possible on the use of the `auto` and `eauto` to solve the conclusions.

Automation

To shorten the size of proofs, we develop user-defined tactics using the Coq `Ltac` language. The tactic that most contributed to the reduction of the size of the proof scripts is the `minv` tactic (see `StateAndErrorMonadTactics.v` under the `common` folder). The `minv` tactic automate the proof of certain lemmas regarding the properties of the HILECOP transformation function in the context of the state-and-error monad. Our Coq implementation of the HILECOP transformation function implements the state-and-error monad. This monad simulates imperative language traits into functional languages. All functions involve in the HILECOP transformation function carry a compile-time state, defined as the Coq type `CompileTimeState`. Each function either return a value, modify the compile-time state or do both. To give an example of the use of the `minv` tactic, Listing 1.5 shows the implementation of the `generate_place_comp_inst` function involved in HILECOP transformation function. The `generate_place_comp_inst` function generates a $\mathcal{H}$-VHDL PCI statement from a place p passed as a parameter. As a side effect, the `generate_place_comp_inst` function adds the PCI statement to the behavior of the top-level design currently built in the compile-time state.

```
1 Definition generate_place_comp_inst (p : P sitpn) : CompileTimeState unit :=
2
3   do id        ← get_nextid;
4   do _         ← bind_place p id;
5   do pcomp     ← get_pcomp p;
6   do pcomp_inst ← HComponent_to_comp_inst id place_entid pcomp;
7   add_cs pcomp_inst.
```

LISTING 1.5: Coq implementation of the `generate_place_comp_inst` function; the function takes an SITPN place p as a parameter, and modifies the compile-time state without returning a value (i.e. the function return type is `unit`)

In its definition body, function `generate_place_comp_inst` sequentially calls to functions that sometimes modify the compile-time state (e.g. the `bind_place` function adds a binding between p and id in the generated γ binder, i.e. $\gamma(p) = id$ after the call to `bind_place`), or sometimes simply return a value without modifying the state (e.g. `get_pcomp` returns an intermediate structure representing the place component instance associated to place p in the compile-time state). During the mechanization of the proof, we often need to prove that some properties hold between the input compile-time state and the output compile-time after the call to a certain function. For example, after calling the `generate_place_comp_inst` function on a given place p and for a given input state s, let us say that a new compile-time state s' is returned. We want to show that the part of the γ binder pertaining to the binding of transitions to TCI identifiers has not changed between state s and state s'[3]. To perform the proof, we need to show that each function call composing the sequence of the `generate_place_comp_inst` function returns a compile-time state verifying the wanted property. Proving simple property like verifying that part of the compile-time states are equal through the multiple invocation of functions is highly automatable. We adapt the tactic `monadInv` defined in the CompCert project [11] to automate proof for such properties. The result is the tactic `minv` massively used in the proofs pertaining to state invariants[4].

Gap between informal and formal proof

There is a huge gap of size between the informal proof of the Full bisimulation theorem given in this Chapter and in Appendix **??** and the machine-checked formal proof. Right now, the Coq proof wins the size competition. The most significant distance between the size of the informal and the formal proof comes from the two following points: the statement of the combinational equations defining the value of $\mathcal{H}$-VHDL signals and the statement of properties about the HILECOP transformation function. Stating that a combinational equation holds for a given signal in the context of an informal proof is a one-line sentence. The same goes when invoking the properties of the PCIs and TCIs populating the top-level design behavior based on the definition of the transformation function. However, proving these statements represents a tremendous mechanization effort within the Coq proof assistant. To give an example, we begin the proof of Lemma **??** by taking a place p and a PCI identifier id_p linked through the γ binder returned by the transformation function. Then, we state the existence of a PCI statement, identified by id_p and with an associated generic map, input port map and output port map, in the behavior of the top-level design returned by the transformation function. To do so, we use the following the sentence:

"Let us take a $p \in P$ and an $id_p \in Comps(\Delta)$ such that $\gamma(p) = id_p$. By construction, there exist 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 shortcut 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.

[3] Remember that the γ binder is part of the compile-time state record type.

[4] State invariance lemmas are to be found in the `GenerateInfosInvs.v`, `GenerateArchitectureIns.v`, `GeneratePortsInvs.v` and `GenerateHVhdlInvs.v` under the `sitpn2hvhdl` folder of the Git repository.

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.

Bibliography

[1] Karima Berramla, El Abbassia Deba, and Mohammed Senouci. "Formal Validation of Model Transformation with Coq Proof Assistant". In: *2015 First International Conference on New Technologies of Information and Communication (NTIC)*. 2015 First International Conference on New Technologies of Information and Communication (NTIC). Nov. 2015, pp. 1–6. DOI: 10.1109/NTIC.2015.7368755.

[2] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. "Formal Verification of a C Compiler Front-End". In: *FM 2006: Formal Methods*. International Symposium on Formal Methods. Springer, Berlin, Heidelberg, Aug. 21, 2006, pp. 460–475. DOI: 10.1007/11813040_31. URL: https://link.springer.com/chapter/10.1007/11813040_31 (visited on 05/25/2020).

[3] 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).

[4] Timothy Bourke et al. "A Formally Verified Compiler for Lustre". In: (), p. 17.

[5] 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.

[6] 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.

[7] 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).

[8] 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.

[9] 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.

[10] 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.

[11] 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).

[12] 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).

[13] 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.

[14] 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).

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

[16] 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).

[17] 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).