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
CIS	**C**omponent **I**nstantiation **S**tatement
PCI	**P**lace **C**omponent **I**nstance
TCI	**T**ransition **C**omponent **I**nstance
GPL	**G**eneric **P**rogramming **L**anguage
HDL	**H**ardware **D**escription **L**anguage
LRM	**L**anguage **R**eference **M**anual
DSL	**D**omain **S**pecific **L**anguage
MDE	**M**odel-Driven **E**ngineering

For/Dedicated to/To my…

Chapter 1

Preliminary Notions

In this chapter, we introduce the mathematical formalisms and notations used throughout this thesis to express and formalize our ideas. Also, in the last section, we provide the basics to understand the Coq proof assistant, which is the system we use to write our programs and to mechanize our proofs. This chapter is inspired by the book *The Formal Semantics of Programming Languages: an Introduction*, the courses of the Cambridge of formal semantics, the documentation of the Coq proof assistant, and the Coq programming with dependent-types by Adam Chlipala.

add r

1.1 Set theory and mathematical notations

1.1.1 Intuitionistic first order logic

The intuitionistic first-order logic [17] constitutes our framework for the expression and the interpretation of logical formulas. The language to express logical formulas is the same between classical and intuitionistic first-order logic. A logical formula is either:

- $\bot$ (bottom), the always false formula, or $\top$ (top), the always true formula
- a predicate (i.e. an atomic formula). A predicate P possibly takes n parameters as inputs and is interpreted to either true, represented by the $\top$ symbol, or false, represented by the $\bot$ symbol. We write $P(x_0, \ldots, x_n)$ to denote an n-ary predicate P applied to the inputs $x_0, \ldots, x_n$.
- the composition of two subformulas with one of the following binary operators: the conjunction $\wedge$, the disjunction $\vee$, the implication $\Rightarrow$, the double implication $\Leftrightarrow$
- the composition of one subformula with the negation operator $\neg$
- a subformula prefixed by the universal quantifier $\forall$ or the existantial quantifier $\exists$. For instance, the formula $\forall x.P(x)$ denotes the atomic formula $P(x)$ where the parameter x is a universally quantified variable of the formula. As a shorthand notation, we write $\forall x, y, z. \ldots$ to denote $\forall x, \forall y, \forall z. \ldots$. The same stands for the existantial quantifier $\exists$.

The difference between the classical first-order logic and the intuitionistic one relies in the absence of the *law of the excluded middle* in the latter logic. In classical logic, the *law of the* *excluded middle* considers that a formula can always be either interpreted as true or false, i.e. a formula is always *decidable* regardless the valuation of its inputs. In intuitionistic logic, such an assumption is discarded. Thus, a formula of the intuitionistic logic can be *undecidable*, i.e. given a valuation of its input, one can not always state that the formula is true or false. This is implication is that one as to exhibit a *explicitly-built* proof to show that a given formula is true. One can not rely on the law of excluded middle to perform a proof by contradiction. As so, the intuitionistic logic is said to be *constructive* in the sense that one as to *construct* a concrete proof object to show that a formula is true. The intuitionistic first-order logic is built in the Coq proof assistant. Thus, a logic formula expressed with the Coq proof assistant is an intuitionistic formula by default. For this reason, we choose the intuitionistic first-order logic as our mathematical framework for the expression of logic formulas.

In what follows, we often give the definition of a predicate by relating its semantics to the semantics of an underlying formula. For instance, we define the predicate $IsEven(n)$, for all $n \in \mathbb{N}$, as follows: $IsEven(n) \equiv \exists k \in \mathbb{N}\ s.t.\ 2k = n$.

1.1.2 Set theory

In this thesis, we use set theory as the base formalism for all our mathematical definitions and proofs. In set theory, a set represents a group of elements called the members of the set. For every set X, we write $x \in X$ to denote that the element x is a member of set X. We note $X \subseteq Y$ the fact that a set X is a subset of set Y such that for all x if $x \in X$ then $x \in Y$. From here, there are multiple ways to define a set.

Extensional definition

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

Intensional definition

The intensional definition of a set specifies a given property verified by all the members of the set. For instance, here is the intensional definition of the set of even numbers: $\{n \in \mathbb{N} \mid \exists k \in \mathbb{N}\ s.t.\ n = 2k\}$ or $\{n \in \mathbb{N} \mid IsEven(n)\}$.

Set operators

Set theory also defines some operators to compose sets together. Given two sets A and B, the following sets are formed:

- $A \cup B$ denotes the set formed by the union of the members of A and the members of B, i.e. $A \cup B = \{x \mid x \in A \vee x \in B\}$.
- $A \cap B$ denotes the set formed by the intersection of set A and B, i.e. $A \cap B = \{x \mid x \in A \wedge x \in B\}$.

- $A \setminus B$ denotes the set formed by the elements of set A that are not elements of set B (the difference between set A and B), i.e. $A \setminus B = \{x \mid x \in A \land x \notin B\}$.
- $A \times B$ denotes the cartesian product between the elements of set A and set B, i.e. the set of all ordered pairs defined by $\{(x, y) \mid x \in A \land y \in B\}$. We generalize the definition to build the set of n-tuples $A_0 \times A_1 \times \cdots \times A_n$ defined by $\{(x_0, (x_1, \ldots (\ldots, x_n) \mid x_0 \in A_0, x_1 \in A_1, \ldots, x_n \in A_n\}$. It is sometimes useful to give a name to the elements of a tuple without refering to their index. In such a case, a tuple is called a record where each element, called a field, has been given an explicit name. This formalism is useful to represent rather complex data structures. For instance, say that we want to represent the set of humans by a triplet composed of the size, weight, and eye color. We can define this set as the set of triplet $\mathbb{R} \times \mathbb{R} \times \{green, blue, brown\}$. If we want to give a concrete name to the elements of the triplet, we can equivalently define such a triplet as a record, written $<size, weight, eye>$, where $size \in \mathbb{R}$, $weight \in \mathbb{R}$ and $eye \in \{green, blue, brown\}$.
- $A \sqcup B$ denotes the set formed by the disjoint union of set A and set B. The disjoint union is obtained by adjoining an index i to the elements of A and an index j to the elements of B such that $i \neq j$. Then, the two sets of couples are joined together to build the disjoint union of A and B. For instance, consider that $A = \{a, b, c\}$ and $B = \{a, b\}$. To obtain the disjoint union $A \sqcup B$, we create the two sets $A_i = \{(i, a), (i, b), (i, c)\}$ and $B_i = \{(j, a), (j, b)\}$, and then join the sets together s.t. $A \sqcup B = \{(i, a), (i, b), (i, c), (j, a), (j, b)\}$. When the two sets A and B are disjoint, i.e. $A \cap B = \emptyset$, then $A \sqcup B$ is isomorphic to $A \cup B$. To stress the fact that we are building a set from the union of two disjoint set, we prefer to use the disjoint union operator. For instance, we write $\mathbb{N} \sqcup \{\infty\}$, instead of $\mathbb{N} \cup \{infty\}$, to denote the set of values ranging from the set of natural numbers with the addition of the infinite value.
- $\mathcal{P}(A)$ denotes the powerset of A defined by all the possible subsets formed with the elements of set A, i.e. $\mathcal{P}(A) = \{X \mid X \subseteq Y\}$.

Relations and functions

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

A partial function f from set X to set Y is a binary relation from X to Y verifying that $\forall x \in X, y, y' \in Y,\ (x, y) \in f \land (x, y') \in f \Rightarrow y = y'$, i.e. x appears at most once as the first element of a pair in f. We note $f \in X \nrightarrow Y$ to denote a partial function from X to Y. The set of the first elements of the pairs defined in f is called the *domain* of f. We write it $\texttt{dom}(f) = \{x \mid \exists y\ s.t.(x, y) \in f\}$.

A total function a, or application, from X to Y is a partial function verifying that all the elements of X appear as the first element of a pair in a, i.e. for all $x \in X$, there exists $y \in Y$ such that $(x, y) \in a$. We note $a \in X \to Y$ to denote an application from X to Y.

Chapter 2

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

The main purpose of this chapter is to present the target language of the HILECOP transformation, i.e. the VHDL language. The formalization and the implementation of the VHDL language syntax and semantics is mandatory to reason about the programs generated by the HILECOP model-to-text transformation. Thus, we want the reader to clearly understand the structure and the semantics of the language to be able to fully grab the proof of semantic preservation presented in Chapter **??**. Specifically, we present here the $\mathcal{H}$-VHDL language, a synthesizable subset of the VHDL language. This subset permits to encode the programs generated by the HILECOP transformation. We devise a formal semantics for $\mathcal{H}$-VHDL which is a simplification of the simulation semantics of the VHDL language. The formalization of the $\mathcal{H}$-VHDL semantics and its implementation is one contribution of this thesis to the many formalization of the VHDL language found in the literature. The chapter is structured as follows. In Section 2.1, we give an informal presentation of the VHDL language syntax and semantics. In Section 2.2, we present the state of the art pertaining to the formalization of the VHDL language semantics. In Section 2.3, 2.4, 2.5 and 2.6, we give the full formalization of the $\mathcal{H}$-VHDL language, a subset of the VHDL language. Section 2.7 illustrates the formal $\mathcal{H}$-VHDL semantics with an example. Finally, Section 2.8 outlines the implementation of the $\mathcal{H}$-VHDL syntax and semantics with the Coq proof assistant.

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

2.1 Presentation of the VHDL language

The intent here is to give an overview of the VHDL language, its purpose, its main syntactal constructs, and an informal description of its semantics as presented in the Language Reference Manual (LRM) [13]. The VHDL language offers a lot of possibility in terms of hardware (and even software) description. Here, we are not trying to be exhaustive in our presentation of the language. We will only maintain our description of the VHDL concepts in the scope that is of interest to us. The readers that are interested in learning more about the VHDL language can refer to [13], [1] and [21].

2.1.1 Main concepts

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

A top-level VHDL program is called a *design*. A VHDL design is composed of two descriptive parts. The first part is called the entity and describes the interfaces of a circuit, namely: the input and output ports, and the generic constants. Listing 2.1 is an excerpt of the `transition` design's entity that defines the generic constants, the input and output port interfaces of the design. The generic clause of the entity holds the declaration of the generic constants. The purpose of generic constants is either to represent some dimensions of the design (e.g. the size of ports, internal signals...) or to represent constant values used throughout the design. In Listing 2.1, one can see that the `conditions_number` generic constant gives a dimension to the type of the `input_conditions` input port, which is an array of Boolean values with indexes ranging from 0 to `conditions_number-1` (that is the meaning of `std_logic_vector (conditions_number-1 downto 0)`. The port clause holds the declaration of input and output ports of the design. The `in` keyword indicates the declaration of an input port and the `out` indicates the declaration of an output port.

```vhdl
entity transition is
  generic (
    transition_type : transition_t := NOT_TEMPORAL;
    input_arcs_number : natural := 1;
    conditions_number : natural := 1;
    maximal_time_counter : natural := 1
  );
  port (
    clock : in std_logic;
    reset_n : in std_logic;
    input_conditions : in std_logic_vector(conditions_number−1 downto 0);
    time_A_value : in natural range 0 to maximal_time_counter;
    time_B_value : in natural range 0 to maximal_time_counter;
    input_arcs_valid : in std_logic_vector(input_arcs_number−1 downto 0);
    reinit_time : in std_logic_vector(input_arcs_number−1 downto 0);
    priority_authorizations : in std_logic_vector(input_arcs_number−1 downto 0);
    fired : out std_logic
  );
```

```
end transition;
```

LISTING 2.1: The entity part of the transition design in concrete VHDL syntax.

Figure 2.1 is a visual representation of the interfaces of the `transition` design.

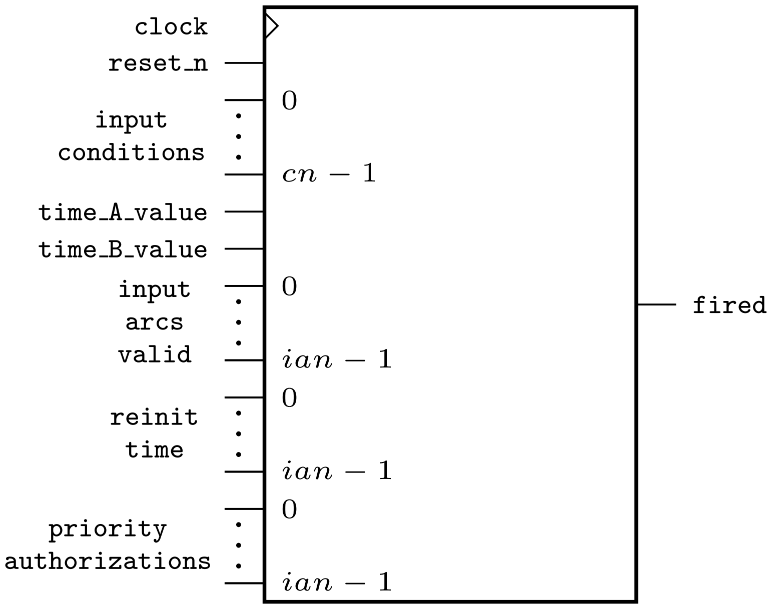

FIGURE 2.1: A representation of the `transition` design entity. On the left side, the input port interface of the transition design; *cn* stands for *conditions_number* and *ian* stands for *input_arcs_number*, i.e. two of the generic constants declared in the generic clause of the `transition` design entity; the numbers at the right of the input pins represent the pin indexes. On the right side, the output port interface of the `transition` design.

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

Behavior specification with processes

The first way to specify the behavior of a design is through the description of processes. Processes are concurrent statements that describes the wiring or the operations performed on the signals of a given design. These operations are described by sequential statements in the body of processes. A process declares a sensitivy list that corresponds to the signals read in the process statement body; also, it possibly declares internal variables. Listing 2.2 gives an excerpt of the `transition` design architecture containing the declarative part of the architecture (i.e. the declaration of internal signals) and three of the processes describing the `transition` design behavior, namely: the `condition_evaluation` process, the `firable` process and the `fired_evaluation` process. In Listing 2.2, Lines 2 to 8 correspond to the declaration of the internal signals of the `transition` design. Line 11 starts the declaration of the `condition_evaluation` process. The sensitivity list of the `condition_evaluation` process holds one signal, the `input_conditions` input port. The value of the `input_conditions` input port is read is the process body. Then, as a design rule, it must be declared in the sensitivity list. The process defines a local variable `v_internal_condition` at Line 12. At Line 13, the `begin` keyword starts the declaration of the process body, i.e. the block of sequential statements performing operations on the signals of the `transition` design.

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

```

end transition_architecture;
```

LISTING 2.2: An excerpt of the architecture part of the transition design in concrete VHDL syntax.

In the statement body of a process, the designer can use control flow statements common to most of the generic programming languages (if statement, for loops. . .), and also statements that are specific to the VHDL language. The most representative statement, and the one of interest to us, is the *signal assignment* statement. The signal assignment statement relate a given signal identifier to a source expression. For instance, at Line 20 of Listing 2.2, the signal assignment statement, represented with the $\Leftarrow$ operator, assigns the value of the internal variable `v_internal_condition` to the `s_condition_evaluation` signal. The `v_internal_variable` that itself holds the Boolean product between the subelements of the `input_conditions` input port performed in the `for` loop of Lines 16 to 18.

When considering a VHDL design in the point of view of hardware synthesis, a signal assignment statement specifies a wiring between a target signal identifier and other source signals. Figure 2.2 gives a synthesis-oriented view of the processes described in Listing 2.2.

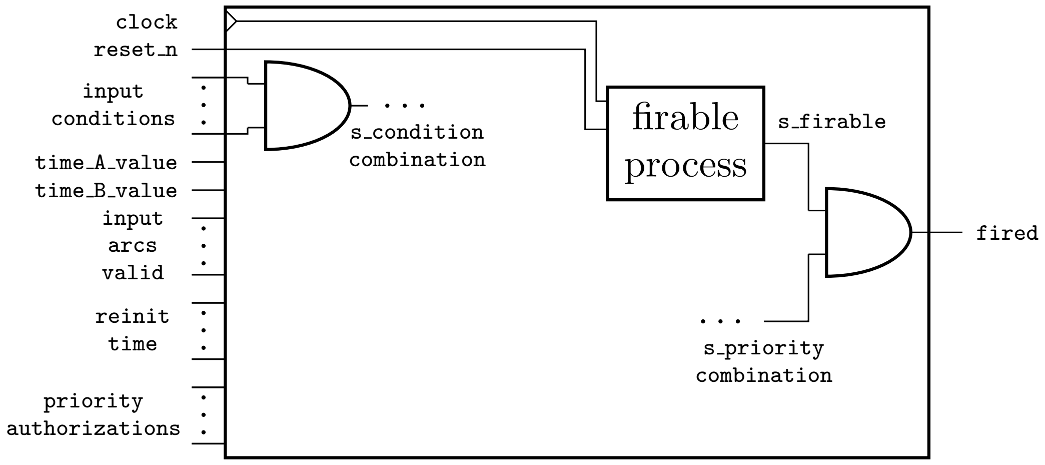

FIGURE 2.2: A representation of a part of the transition design architecture comprising three processes. On the left side, the `condition_evaluation` process connecting the `input_conditions` input port to the `s_condition_combination` internal signal; the `firable` process in the middle; on the right side, the `fired_evaluation` process connecting the `s_firable` and the `s_priority_combination` signals to the `fired` output port.

In Figure 2.2, the `condition_evaluation` process is represented as an `and` port performing the product over the elements of the `input_conditions` input port. The `fired_evaluation` process is a simple `and` gate connecting the `fired` output port to the `s_firable` and `s_priority_combination` internal signals. The `fired_evaluation` and the `condition_evaluation` processes are combinational processes. They describe the value of an output signal based on the value of input signals. For instance, the value of the `s_condition_combination` signal is a function of the value of the `input_conditions` input port, i.e `s_condition_combination`=

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

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

To complete the presentation of the statements to be found in the body of processes, the VHDL language is also equipped with `timing` constructs, i.e. statements that explicitly specify an amount of time in a given time unit. The signal assignement statement possibly specifies a time clause indicating when the assignment must be performed. For instance, the signal assignment statement specifying that the value of signal `b` must be assigned to signal `a` in 3 milliseconds takes the form: $a \Leftarrow b$ `in 3 ms`. When no time clause is specified for a signal assignment statement, we talk about a δ-delay signal assignment, i.e. the application of the signal assignment is related to some δ interval corresponding the time of propagation through a wire. When a time clause is specified, we talk about an unit-delay signal assignment. δ-delay signal assignments are synthetizable, meaning they have an equivalent implementation on a physical device, whereas, unit-delay signal assignments can not be synthetized. Unit-delay signal assignments do not appear neither in the VHDL designs generated by HILECOP transformation nor in the declaration of the place and transition designs. We are only mentioning their existence because they play a part in the simulation algorithm described in Section 2.1.2.

Behavior specification with design instances

The second way to specify the behavior of a design is to use other designs, or rather instances of other designs, as subcomponents. In that case, the design is said to be composite as it embeds instances of other designs in its own behavior. Also, a design at the highest level of embedding, i.e. that is not instantiated as a part of another design's behavior, is called a *top-level* design. The design instantiation, or component instantiation, statement permits to instantiate a design in an embedding architecture. When instantiating a design with a design instantiation statement, the designer provides the component instance with an identifier. Then, the design instance must be dimensioned; this is performed through a generic map that associates the generic constants of the design being instantiated to a static value. Finally, the designer specifies how the component instance is connected to the other elements of the architecture. A port map associates the input ports and output ports of the component instance to expressions or to the signals of the embedding architecture. Listing 2.3 shows an example of instantiation of the HILECOP's `transition` design. This instance is involved in the definition of the behavior of an embedding design called `toplevel`.

```
architecture toplevel_architecture of toplevel is
begin
```

```
  ...
  id_t : entity transition
  generic map (
    transition_type ⇒ NOT_TEMPORAL,
    input_arcs_number ⇒ 1,
    conditions_number ⇒ 1,
    maximal_time_counter ⇒ 1
  )
  port map (
    clock ⇒ clock,
    reset_n ⇒ reset_n,
    time_A_value ⇒ 0,
    time_B_value ⇒ 0,
    input_conditions(0) ⇒ id_0,
    input_arcs_valid(0) ⇒ id_1,
    priority_authorizations(0) ⇒ '1',
    reinit_time(0) ⇒ id_2,
    fired ⇒ id_3
  );
  ...
end toplevel_architecture;
```

LISTING 2.3: An example of design instantiation statement in the architecture of the `toplevel` design. Here, the design being instantiated is the transition design.

In Listing 2.3, the `transition` component instance (TCI) has the identifier id_t. Following the `entity` keyword is the name of the design being instantiated; here, the `transition` design. Then, the generic map associates the generic constants of the transition design (i.e. the left side of the arrow, also called the *formal* part) to static values (i.e. the right side of the arrow called the *actual* part). This permits the dimensiniong the component instance. For example, remember that the `input_arcs_number` generic constant value determines the number of elements in the composite input ports `input_arcs_valid`, `priority_authorizations` and `reinit_time` (cf. Figure 2.1). The port map associates the input ports of the `transition` design to expressions. For instance, the `time_A_value` input port is connected to the constant value 0, and the `input_conditions` input port is connected to the internal signal id_0 at index 0. The port map also associates the output ports with signal identifiers. Contrary to the association of input ports, output ports can not be associated to expressions. An output port association describes a direct wiring. In the port map described in Listing 2.3, the association `fired` ⇒ id_3 expresses that the `fired` output port is connected to the signal id_3, where signal id_3 is defined in the embedding design. Figure 2.3 illustrates the `transition` component instance id_t and the wiring of its input and output port interfaces inside the `toplevel` design.

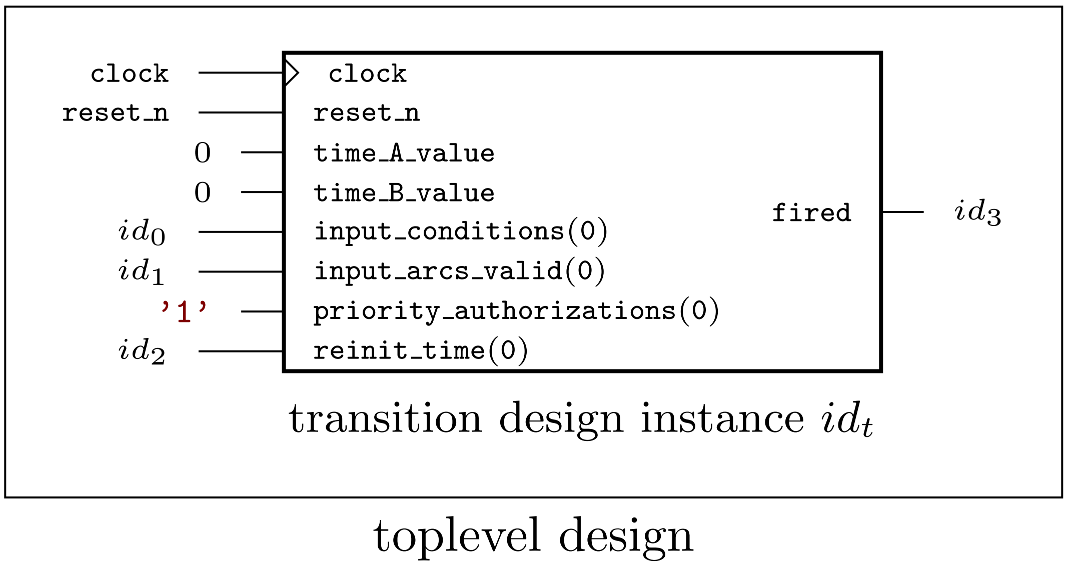

FIGURE 2.3: Visual representation of a design instantiation statement. Here, the figure represents the transition design instance described in Listing 2.3.

2.1.2 Informal semantics of the VHDL language

There are two approaches to the description of circuits with the VHDL language. The first aims at the simulation of the described circuits, and the second aims at the synthesis of described circuits on physical supports. These two approaches arise from the practice and the use of the VHDL language by electronicians. Even though, in practice, there are two ways to consider a VHDL design, i.e. a synthesis-oriented way and a simulation-oriented way, the LRM does not define a synthesis-oriented semantics for the VHDL language. A synthesis-oriented semantics gives an interpretation to a design by describing an equivalent in a lower level formalism, closer to the physical circuit. For instance, the Verilog language gives a synthesis-oriented semantics to its programs by defining an equivalent RTL level description [14]. The LRM gives an informal semantics to VHDL designs through the definition of a simulation algorithm [13, p.167]. The purpose of simulation is to compute the evolution of the values of signals during a certain time interval. Through the simulation process, the designer is able to control the behavior of the modeled circuits and to detect flaws in the evolution of the signal values.

Former to the simulation, the LRM defines an elaboration phase. The elaboration phase operates syntactic and semantic controls over the design code. It also describes code transformations over the design's behavioral part to obtain a simulation-ready execution model. More specifically, the elaboration phase builds the simulation environment and the default simulation state associated with the design under simulation. The simulation environment is built based on the declarative parts of the top-level design; it maps the signals to their types. In the default simulation state, each signal is associated with a current value (i.e. the default value of the signal's type) and with a driver. A driver maps time points to values and the association between a given time point and a signal value is called a transaction. The need for drivers the values associated with a given singal is explained by the presence of unit-delay signal assignments. A unit-delay signal assignment specify a time clause indicating when a giving assignment must be performed, e.g. $a \Leftarrow b$ `in 3ms` (signal `a` takes the value of signal `b` in 3 milliseconds). Thus, when a unit-delay signal assignment is executed in the course of a simulation, its effect is to change the driver of the target signal by posting a new transaction. For instance, let T_c by the current simulation time, the execution of statement $a \Leftarrow$ `true in 2ns` sets a new transaction in the driver of signal a. The new transaction associates the value `true` to the time point $T_c +$ `2ns`. Note that without unit-delay signal assignments, i.e. without a specified time clause, drivers are not needed as all assignments take effect at the current simulation time. Moreover, the elaboration checks the well-formedness of the design by performing static type-checking on the behavioral part of the design. It also checks that the connection between signals respect certain rules, for instance, that there are no multiply-driven signals, i.e. signals that are written to by multiple processes. Finally, the elaboration operates some transformations over the VHDL code, and thus builds the *execution* model. To summarize, all concurrent statements of the behavioral part are transformed until the top-level design behavior is only composed of processes.

After the elaboration, the top-level design, or rather its corresponding execution model, is ready to be simulated. Two entities are involved in the simulation: the *sea* of processes obtained after the elaboration of the top-level design's behavior, and a *kernel* process. The kernel process orchestrates the simulation; it handles the time of the simulation, i.e. it holds a variable describing the current time of the simulation, and controls the execution of processes. Figure 2.4, which is an excerpt from [5], describes the structure of the VHDL simulation algorithm.

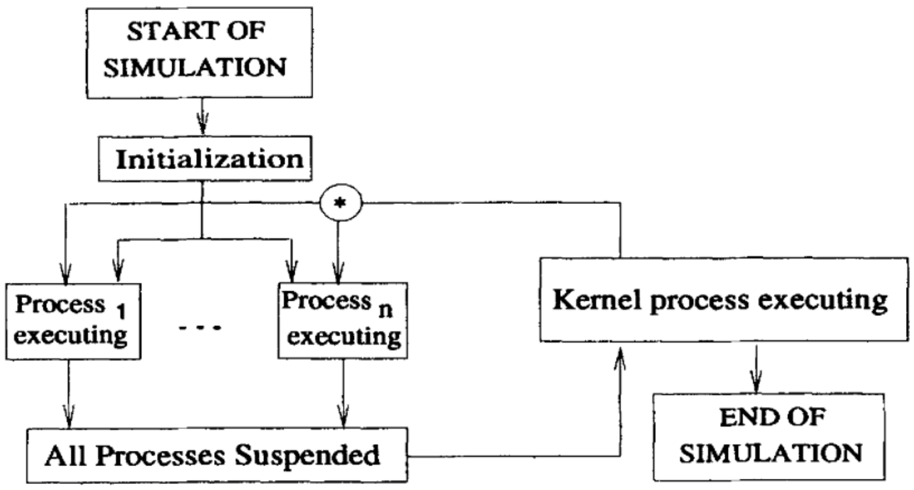

FIGURE 2.4: The VHDL simulation loop. Excerpt from [5].

The simulation starts with an initialization phase. During the initialization phase all processes are run exactly once. Then, the simulation cycles are structured as follows. All processes execute their statement body concurrently. New transactions are posted in the drivers of signals for every interpreted signal assignment statement. The execution goes on until all processes have executed their statement body and then have reached a suspension state. When, all processes are suspended, the kernel process takes over. Figure 2.5 shows the activity diagram associated with the kernel process.

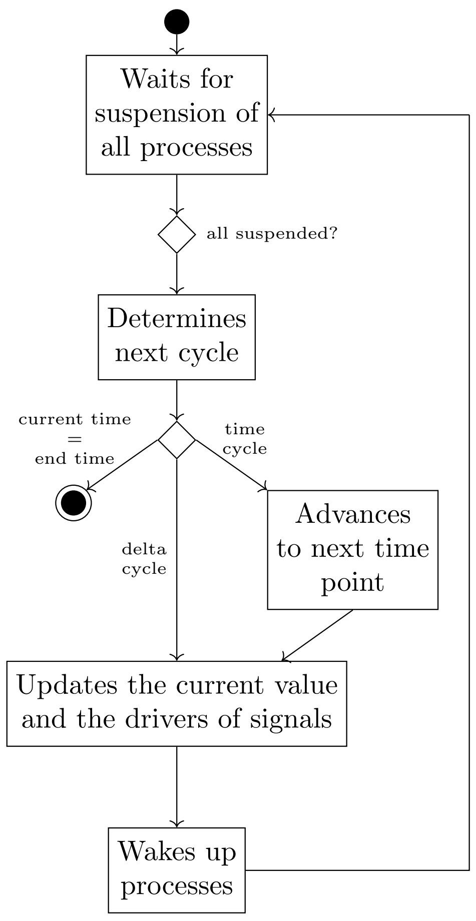
Waits for suspension of all processes
all suspended?
Determines next cycle
current time = end time
time cycle
delta cycle
Advances to next time point
Updates the current value and the drivers of signals
Wakes up processes

FIGURE 2.5: The activity diagram of the kernel process. Square boxes represent activities, diamond nodes are decision nodes. The black circle at the top represents the starting point of the activities; the other black circle in the middle of the diagram represents the end of all activities.

As shown in Figure 2.5, after the suspension of all processes, the kernel process will then determine the kind of simulation cycle that will be performed next. There are two kinds of cycles: delta cycles or time cycles. If the value of a signal changes at the current time point, i.e. its driver holds a transaction at the current time point with a new value, then a delta cycle must be performed. Then, the simulation time does not change. The kernel process updates the current value of signals and their drivers, and wakes up the processes sensitive to the signals that obtained new values. The repetition of multiple delta cycles corresponds to the stabilization of signal values, i.e. the propagation of values through the wires, that takes effect in a negligeable δ time. If all signal values are stable at the current time point, then a time cycle must be performed. The kernel process looks up the drivers for the next time point where the value of a given signal will change. Then, the kernel process advances the simulation time to this next time point before updating the signal values and resuming the execution of processes. The simulation goes on like this, alternating between delta and time cycles, until the current time value reaches the time specified for the end of the simulation.

2.2 Choosing a formal semantics for VHDL

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

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

2.2.1 Specifying our needs: HILECOP and VHDL

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

The need for coverage

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

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

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

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

Regarding component instantiation statements, the VHDL LRM describes a way to transform these statements into equivalent process statements and block constructs [13, p. 141] which are a part of the elaboration of the design. However, we want to preserve the hierarchical structure provided by the component instantiation statements arguing that it will be easier to compare the state of a given SITPN model with a VHDL design state with an explicit hierarchical structure. Indeed, there exists a mapping between places and transitions of an SITPN and their mirror (generated by the transformation) place and transition component instances (PCIs and TCIs). This one-to-one correspondence might turn out to be handy to perform the proof of behavior preservation. Obviously, the semantics must cover the evaluation of process statements which are the core concurrent statements of VHDL programs.

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

The need for a synchronous execution

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

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

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

Other considerations

Considering the kind of proof that needs to be established, we would rather consider an operational semantics for VHDL. The reason is that, in the CompCert project [16], which is one of our major inpsiration source, the whole C compiler toolchain is verified by reasoning over the operational semantics of the source and target languages. A last consideration pertains to whether or not the VHDL semantics must explicitly handle errors. As the SITPN semantics does not include the production of error values, the handling of errors by the VHDL semantics is not a mandatory aspect.

Qualifying criterions

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

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

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

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

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

2.2.2 Looking for an existing formal semantics

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

Denotational semantics

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

In [10], the authors give a denotational semantics to the VHDL language within the Focus [7] framework, a method for the development of distributed systems. Signal values and their evolution through time are represented as streams of values. Statements are denoted as stream-processing functions. Processes are stream-processing functions that takes input signal streams (signals of the sensitivity list) and yields transaction traces (i.e, waveforms) over output signals (i.e, signal that are written by the process). Transaction traces are merged together as the result of the concurrent execution of processes. The authors only consider 0-delay signal assignments in their semantics, stating that it is sufficient to "consider time at a logical level to model both synchronous and asynchronous designs". However, it necesitates some transformations on a design that has a synchronous execution to express it only with 0-delay signal assignments. Therefore, this semantics does not express synchronism of execution in an explicit manner. Moreover, the component instantiation statements are not dealt with, and no mention is made of the elaboration phase.

In [4], the authors give a denotational, yet relational, semantics for VHDL. A state of a VHDL design is represented by a function binding signals to values; a worldline is a time-ordered list of states. Statements (including processes) are denoted in the semantics by a relation that binds an input couple, composed of a time point and a worldline, to an output couple of the same type. Multiple input and output couples possibly satisfy the relation denoting a particular statement; thus, the semantics is undeterministic. The semantics tries to abstract from the formalization of the simulation cycle as it is done in the LRM. The authors want to establish a semantics that is abstract enough to be able to compare all other works of formalization with the authors semantics. The authors also give an axiomatic semantics (i.e, in the Hoare logic style) which is proved to be sound and complete with the first denotational semantics. A Prolog [6] implementation of the axiomatic semantics is given. Regarding our needs, the semantics only deals with unit-delay signal assignments. However, this semantics enables the representation of a δ-delay signal assignment with a unit-delay signal assignment adorned with a "`after 0 ns`" time clause. The hierarchical structure of designs is not preserved, and, although expressible, the semantics does not explicitly express a synchronous simulation cycle.

The denotational semantics expressed in [20] uses interval temporal logic as an underlying model. Leveraging this underlying model, the authors are interested in proving some properties over VHDL designs to help compilers to optimize the code, for instance, by using rewrite rules proved to be valid against the model. Some of the proofs laid out by the authors are embedded in PVS [19]. The expression of the dynamic model uses many concepts described in the LRM, like drivers, port association, driving and effective values for signals. The semantics deals with both unit-delay and δ-delay signal assignments. The semantics works on fully-elaborated designs, therefore, it does not deal with component instantiation statements. Moreover, interval temporal logic is useful to reason on the VHDL designs in the presence of delays, however, it looses its interest for designs presenting only 0-delay assignments.

In [2], the author states that "denotational semantics is more adequate for mathematical reasoning". The author formalizes the VHDL semantics to prove the equivalence between VHDL programs (for instance, a specification and an implementation). What is of major interest regarding our needs is that the author has expressed a simulation cycle for synchronous designs. Therefore, a distinction is made between combinational and synchronous processes in the abstract syntax. Moreover, this work formalizes the elaboration part of a VHDL design former to the simulation; also, the elaboration keeps the hierarchical setting of the VHDL design, that is component instantiation statements are not replaced by processes. Due to the time abstraction, the semantics only deals with 0-delay. It is explained by the fact that the reference time-unit is the clock period (i.e, the only known time-step), and the advancing of time, happening during the simulation cycle as described in the LRM, is captured within the setting of the simulation cycle. Also, the semantics takes primary inputs into account (i.e, input ports of the top-level design); to preserve a synchronous behavior for the simulated design, the hypothesis is made that the values of the primary inputs are stable between two clock events.

Operational semantics

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

In [3], a formal description of a *functional* semantics for VHDL is laid out based on stream-processing functions. The semantics is expressed with the functional programming language Gofer [15], thus enabling the computation of execution traces, that is, the computation of the streams representing the values taken by signals over time. As in the former work of the same author [4], only unit-delay signal assignments are dealt with, however, this time the author describes a deterministic operational semantics. Regarding our needs, this work is neither interested in preserving the hierarchical structure of VHDL designs, and no mention is made regarding how a design is elaborated, nor in expressing an explicit synchronous simulation cycle.

In [5], the authors formalize the simulation loop of the LRM using Evolving Algebra machines (EA-machines). All important constructs of the VHDL language are represented as records; processes are represented as concurrent agents running pseudo-codes, and the simulation control flow is passed to and fro between the kernel process (i.e, the simulation orchestrator) and the rest of the processes that execute the design behavior. This semantics implements closely the simulation loop as described in the LRM. Therefore, it is very rich and deals with most of the VHDL constructs, including the two time paradigms of the language (i.e. δ time and unit time). Moreover, the semantics works on fully-elaborated designs, therefore, component instantiation statements are omitted. However, a synchronous execution is fully expressible even if not explicitly embedded in the expression of the simulation loop.

In [24], the author presents a natural semantics for VHDL. The simulation loop is expressed by inference rules, and the execution of processes is based on the events over signals of their corresponding sensitivity lists. The execution of statements computes transaction traces, that is, the projected waveforms for signals over the future of the simulation. The semantics deals both with unit and delta delays. Regarding our needs, this semantics covers does not entirely cover the subset of VHDL we are interested in. Component instantiation statements are not dealt with. A synchronous execution is expressible within the semantics, although it would be hidden in the inference rule formalizing the generic simulation loop. Also, the semantics does not provided its simulation loop with a simulation horizon (a maximum number of simulation cycle to be computed). The simulation ends when signal values evolve no more. The question of the influence of the environment, measured through the values of the primary inputs of a design, is not discussed.

In [11], the author presents an operational semantics for VHDL in the small-step style. The semantics follows closely the simulation cycle described in the LRM; however it is very concise and clear. The covered VHDL subset comprises both unit and delta-delay signal assignments. There is a interesting discussion about the non-determinism of VHDL, since it is a concurrent programming language: it entails that non-determinism is only existent at the processes level, that is, internal sequential statement of processes can be executed in an undeterministic manner (refered to by the author as A actions, that is, *internal* actions). But at every delta or time step (refered to as δ and T actions) of the execution, the design state can be computed in a deterministic manner, since all processes have reached a suspension point at the end of their inner body. The author is interested in comparing the behaviors of two VHDL designs by proving that some relation of equivalence holds between the two. He describes two strategies to compare VHDL programs. The first one is bisimulation; it is based on the comparison of the sequence of actions (either A, δ or T actions) performed by the two programs. The second one is observational equivalence; it is based on the observation of the value of the output signals of two VHDL programs (the observees), that receive values in their input signals from another VHDL program (the observer). The observer stimulates the entries of the observees and reaches a success state based on its observations of the value of the outputs. Regarding our needs, this semantics permits the description of our synchronous simulation cycle. However, like most of the semantics presented here, the component instantiation statement is not supported as it stands, but it is rather transformed into the equivalent processes statements. Small-step semantics is not needed in our case because we are only interested in the values of signals at the delta and time steps (for us, time steps correspond to clock events). We are not interested in capturing the design states in the middle of the execution of a process body. We are more interested in "weak bisimulation", therefore forsaking the internal actions performed by a VHDL design. In [23], the authors extend the work of [11], especially by handling shared variables, in the presence of which a VHDL program can have a concrete underterministic behavior. The authors are also interested in the equivalence between two VHDL programs, and they are interested in determining a unique meaning property for VHDL programs. The unique meaning property states that the execution of a VHDL design in the presence of shared variables is unique. This work is interesting as it points out the fact that VHDL is not only subject to benign undeterminism. However, we are not interested in dealing with constructs so advanced as shared variables, therefore, this work is not really relevant to us.

Translational semantics

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

In [22], the author expresses the formal semantics of VHDL by translating a VHDL design into a corresponding *flowgraph*. All VHDL constructs, ranging from sequential statements to concurrent processes, are expressed with individual flowgraphs that are then composed together through their interfaces. The simulation cycle of VHDL is also encoded by means of connected flow graphs: one for the "execution part" of the semantics, that is, all processes run until suspension, and one for the update part (i.e, the kernel process in the semantics of [5]). Flowgraphs come with a large amount of tools for analysis, and this translational semantics is involved in the setting of a framework to reason about VHDL programs using multiple technics (automatic theorem proving, model checking...). All these technics lean on the flowgraph formalism.

In [9], the author introduces a translational semantics for VHDL based on deterministic finite-state automatons. Again, the reason for using such automatons lies in the existence of many analysis tools. Moreover, forcing the generation of deterministic automatons improves the time execution of model-checking technics. The translation is performed on an elaborated VHDL design; a data space stores the values of signals and variables, and automatons represent the control-flow of VHDL statements. Each VHDL statement is associated to a specific automaton; sequence of statements are achieved by automaton composition. The simulation kernel is also represented by a specific automaton. Processes are composed together with respect to synchronization states, i.e. states that permit to pass the control from one process to another, therefore achieving determinism in the control flow of the overall automaton.

In [18], the author presents a translation from VHDL to Coloured Petri Nets (CPNs) thus giving a formal semantics to VHDL constructs. The author approach to VHDL semantics is a strict translation of the "event-based" VHDL simulator by means of Petri nets. The author translates VHDL execution models (sea of processes) into CPNs, and also translates the kernel process into a CPN. The kernel process has previously been expressed as a VHDL process so that the translation into CPN is similar to the translation of other processes. Signals are not represented in the subnets, instead, three shared variables depict the signal states: one variable for the driving, one for the effective and one for the current value of a given signal (see [13, p.167] for the details on the values associated with signals during the simulation). Colour domains of places in the subnets represent the different types of VHDL domains. Variables are represented by tokens. Values in drivers are represented by sequences of transactions (equivalent to waveforms); the author defines a set of functions that are convenient to handle sequences of transactions. Sequential statements are partitioned into two kinds: control flow (if, loop, case...) and notation (operations on signals and variables) nets. Processes subnets are made by the fusing of each sequential statements in the process body. There is a special *Resume* place that can be set by the kernel process to resume the activity of a process. Concurrency is not discussed here, as the Petri net models are inherently concurrent models. The kernel process is a broad CPN having some of its places interfaced with the process subnets. The decoloration of the Petri net enables the analysis of the model and the detection of dead-locks.

In [8], the author gives a formal semantics to VHDL by transforming a VHDL design into an abstract machine, i.e defined by a set of inputs, outputs, states and transition function over states and outputs. The author is interested in the verification of properties over VHDL designs (temporal properties) or to prove equivalence between designs (bisimulation). To operate this transformation, only a subset of VHDL is considered, otherwise a finite-state representation is not reachable. The covered VHDL subset consists of objects with finite types, and no quantitative timing constructs (no after clause in signal assignments). The transformation generates a decision diagram (i.e. a control flow graph) and a state space for each process defined in the design's behavior. The decision diagram encodes the transition function over states and outputs. Process statements are composed with a special composition operator to obtain a global abstract machine. Moreover, the article lays out a method to transform a block statement into an abstract machine. The initiative is to be noticed as there are only a few papers, dealing with the formalization of the VHDL semantics, that are interested in such hierarchical constructs as block or component instantiation statements. The article concludes with an expression of the space of complexity entailed by the transformation of a VHDL design into an abstract machine.

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

Table 2.1 summarizes the analysis of the VHDL semantics encountered during our literature review. Table 2.1 compares the different VHDL semantics in relation to our qualifying criterions (see Section 2.2.1).

		Fuchs and Mendler [10]	Breuer et al. [4]	Pandey et al. [20]	Borrione and Salem [2]	Breuer et al. [3]	Börger et al. [5]	Van Tassel [24]	Goossens [11]	Reetz and Kropf [22]	Döhmen and Herrmann [9]	Olcoz [18]	Déharbe and Borrione [8]
Semantics Description	Kind	D	D, A	D	D	O	O	O	O	T	T	T	T
	Purpose	AR, ATP	AR	AR	AR	SS	SS	SS, ITP	SS, MC	ATP, MC, ITP	MC, ITP	MC	MC
Qualifying Criterions	Component Instantiation	T	T	T	N	T	T	T	T	T	T	T	N
	Synchronism	NE	NE	NE	Ex	E	E	E	E	E	E	E	NE
	Elaboration	×	×	×	✓	×	×	✓	×	×	×	×	✓
Extra. Informations.	Impl. Technology	Focus [7]	Prolog [6]	PVS [19]	?	Gofer [15]	?	HOL [12]	?	HOL [12]	?	?	?
	Particular Model or Data Types	Stream Processing	No	Interval Temporal Logic	No	Stream Processing	Evolving Algebra Machines	Natural Semantics (big-step)	Structural Semantics (small-step)	Flow Graphs	Finite-State Automatons	Colored Petri Nets	Abstract Machines and Decision Diagrams

TABLE 2.1: A comparative summary on VHDL formal semantics.

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

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

In Table 2.1, cells are colored in green when the cell's content foster the adoption of the semantics, in yellow when the content does not go towards the adoption of the semantics but is not disqualifying, and red when the content is a disqualifying criterion. With regards to the semantics adoption, cells are labelled in light grey when their content is neutral. Now comparing the entries of Table 2.1 with the expression of our needs, we can discard the semantics with a cell labelled in red, that is most of the denotational semantics; moreover, all translational semantics are let aside for the reasons cited before. The candidate semantics are the operational semantics, plus the denotational semantics by Borrione and Salem [2], the only semantics that formalizes an explicitly synchronous simulation cycle. The semantics that is the most likely to be adopted is the Borrione and Salem's semantics. However, we prefer an operational setting for our semantics because it is more fit to our task. To lower down the complexity of proofs, we really need a semantics that builds the synchronism into its simulation cycle, therefore putting aside all the intricacies of the full-blown VHDL simulation cycle. Moreover, the big-step style for an operational semantics is more relevant to us; as stated before, we are not interested in the intermediary states of computation that a small-step style semantics considers. Based on these observations, we have decided to formalize our own VHDL semantics inspired from the semantics of Borrione and Salem's [2] and Van Tassel's [24]. The following sections are dedicated to the presentation of the syntax and semantics of a subset of VHDL called $\mathcal{H}$-VHDL. $\mathcal{H}$-VHDL embeds the subset of VHDL that we are interested in when considering the VHDL designs generated by the HILECOP transformation.

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

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

2.3.1 Design declaration

Similarly to [24], we define the *design* construct in the $\mathcal{H}$-VHDL's abstract syntax which has no equivalent in the concrete syntax of VHDL.

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

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

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

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

2.3.2 Concurrent statements

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

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

Process statement

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

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

Component instantiation statement

The VHDL LRM defines two kinds of component instantiation statement (CIS): the instantiation of a component instance [13, p.139] and the instantiation of a design entity [13, p.141]. The component instantiation statement used in the $\mathcal{H}$-VHDL abstract syntax corresponds to the instantiation of a design entity.

$$\begin{array}{lcl}
\text{cistmt} & ::= & \texttt{comp}\ (\text{id}_c\,,\ \text{id}_e\,,\ \text{gmap}\,,\ \text{ipmap}\,,\ \text{opmap}) \\
\text{gmap} & ::= & \text{assoc}_g^* \\
\text{ipmap} & ::= & \text{assoc}_{ip}^* \\
\text{opmap} & ::= & \text{assoc}_{op}^* \\
\text{assoc}_g & ::= & (\text{id}\,,\text{e}) \\
\text{assoc}_{ip} & ::= & (\text{name}\,,\text{e}) \\
\text{assoc}_{op} & ::= & (\text{id}\,,(\text{name} \mid \texttt{open})) \mid (\text{id}(\text{e})\,,\text{name})
\end{array}$$

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

2.3.3 Sequential statements

$$\begin{array}{lcl}
\text{ss} & ::= & \text{name} \Leftarrow \text{e} \mid \text{name} := \text{e} \mid \texttt{if}\ (\text{e})\ \text{ss}\ [\text{ss}] \mid \texttt{for}\ (\text{id}\,,\text{e}\,,\text{e})\ \text{ss} \\
& & \mid \texttt{falling}\ \text{ss} \mid \texttt{rising}\ \text{ss} \mid \texttt{rst}\ \text{ss}\ \text{ss}' \mid \text{ss};\ \text{ss} \mid \texttt{null}
\end{array}$$

The ss entry defines the sequential statements that compose the body of processes. The signal assignment statement is represented with the $\Leftarrow$ operator; the variable assignment statement with the := operator. Also, we devise three control flow statements that have no equivalent in the VHDL syntax: the `falling` block statement, the `rising` block statement and the `rst` block (or reset) block statement. The `falling` statement (resp. `rising` ss) declares a block of sequential statements to be executed only at the falling edge (resp. rising edge) of the clock signal (see Section 2.6.5). Also, the `rst` statement declares two blocks, the first one must be executed during the initialization phase of the simulation; otherwise, the second one is executed (see Section 2.6.4). These invented constructs are equivalent to specific if-else statements that are commonly used in the body of a synchronous process (see Section **??** for an example of transcription of a specific if-else statement into one of these constructs).

2.3.4 Expressions, names and types

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

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

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

2.4 Preliminary definitions

2.4.1 Semantic domains

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

TABLE 2.2: The *type* and *value* semantic types.

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

In Table 2.2, the *type* type is in any way similar to the τ entry of the $\mathcal{H}$-VHDL abstract syntax. However, all constraint bounds, that were taking the form of expressions in the constrained natural and the array type indications, have been evaluated to natural numbers. `NATMAX` denotes the maximum value for a natural number. The `NATMAX` value depends on the implementation of the VHDL language; `NATMAX` must at least be equal to $2^{31} - 1$. Note that the *array* value contains at least one value as an array's index range contains at least one index.

Notation 1 (Partial functions)**.** *In the following sections, when the context is free from any ambiguity, we adopt the following notations:*

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

2.4.2 Elaborated design and design state

Now, let us define the elaborated design structure. The elaborated design structure is built during the elaboration phase (see Section 2.5). Then, the elaborated design will act as a runtime environment in the expression of the simulation rules. Let *ElDesign* be the set of the elaborated designs. An elaborated design is a composite environment built out of multiple sub-environments. Each sub-environment is a table, represented as a function, mapping identifiers of a certain category of constructs (e.g, input port identifiers) to their declaration information (e.g, type indication for input ports). We represent an elaborated design as a record where the fields are the sub-environments. An elaborated design is defined as follows:

Definition 1 (Elaborated Design)**.** *An elaborated design $\Delta \in ElDesign$ is a record $<Gens, Ins, Outs, Sigs, Ps, Comps>$ where:*

– *$Gens \in generic\text{-}id \to (type \times value)$ is the function yielding the type and the value of generic constants.*

– *$Ins \in input\text{-}id \to type$ is the function yielding the type of input ports.*

– *$Outs \in output\text{-}id \to type$ is the function yielding the type of output ports.*

– *$Sigs \in declared\text{-}signal\text{-}id \to type$ is the function yelding the type of declared signals.*

– *$Ps \in process\text{-}id \to (variable\text{-}id \to (type \times value))$ is the function associating process identifiers to their local environment.*

– *$Comps \in component\text{-}id \to ElDesign$ is the function mapping component instance identifiers to their own elaborated design version*

We assume that there are no overlapping between the identifiers of the sub-environments (i.e, an identifier belongs to at most one sub-environment), and also between the identifiers of the sub-environments and the identifiers of local environments. When there is no ambiguity, we write $\Delta(x)$ to denote the value returned for identifier x, where x is looked up in the appropriate field of Δ. We write $x \in \Delta$ to state that identifier x is defined in one of Δ's fields. We note

$\Delta(x) \leftarrow v$ the overriding of the value associated to identifier x with value v in the appropriate field of Δ, $\Delta \cup (x, v)$ to note the addition of the mapping from identifier x to value v in the appropriate field of Δ, that assuming $x \notin \Delta$. We write $x \in \mathcal{F}(\Delta)$, where $\mathcal{F}$ is a field of Δ, when more precision is needed regarding the lookup of identifier x in the record Δ.

Let Σ be the set of design states. A design state of $\sigma \in \Sigma$ is defined as follows:

Definition 2 (Design state)**.** *A design state $\sigma \in \Sigma$ is a record $<\mathcal{S}, \mathcal{C}, \mathcal{E}>$ where:*

- *$\mathcal{S} \in$ signal-id $\rightarrow$ value, is the function yielding the current values of the design's signals (ports and declared signals).*
- *$\mathcal{C} \in$ component-id $\rightarrow \Sigma$, is the function yielding the current state of component instances.*
- *$\mathcal{E} \subseteq$ signal-id $\sqcup$ component-id, is the set of signal and component instance identifiers that generated an event at the current design state.*

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

Notation 2 (No events design state)**.** *The function $NoEv \in \Sigma \rightarrow \Sigma$ returns a design state similar to the one passed in parameter but with an empty set of events. I.e, for all design state $\sigma \in \Sigma$ s.t. $\sigma = <\mathcal{S}, \mathcal{C}, \mathcal{E}>$, $NoEv(\sigma) = <\mathcal{S}, \mathcal{C}, \emptyset>$.*

2.5 Elaboration rules

The goal of the elaboration phase is to build an elaborated design Δ along with a *default* state σ_e, out of a $\mathcal{H}$-VHDL design d and for a given design store $\mathcal{D}$. The elaboration relation performs type-checking operations over the declarative and behavioral parts of the design. Even though the elaboration of a design is described in the LRM, the formalization of this phase has been performed in few works only [2, 8, 24], and never in a setting that covers both syntactical well-formedness and type-checking of the designs. We are interested in the formalization of the elaboration phase because we are interested in the *well-formedness* of the programs generated by the HILECOP transformation. Here, the term well-formedness refers to a syntactically valid design, w.r.t. the syntactic rules of the VHDL language, and to a well-typed design, w.r.t. the typing rules defined in the LRM. Formalizing the elaboration phase is also a way to define how the runtime environment and the runtime state of the simulation are built. For now, we haven't tackle down the proof that the $\mathcal{H}$-VHDL designs generated by HILECOP are elaborable, i.e. syntactically well-formed and well-typed. As explained in Chapter **??**, this task is foreseen in our work perspectives. In our own formalization of the elaboration phase, and contrary to what is prescribed by the LRM [13, p. 166], we are not dealing with the transformation of the component instantiation statements into block statements. We prefer to preserve the hierarchical structure of the design (i.e. its composite structure) during its elaboration. We argue that dealing with component instantiation statements instead of block statements does not complexify the semantics of the $\mathcal{H}$-VHDL simulation rules.

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

2.5.1 Design elaboration

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

Definition 3 (Design store). *A design store $\mathcal{D} \in$ entity-id $\nrightarrow$* design *is a partial function mapping design identifiers (i.e. the entity identifier of designs) to their corresponding representation in abstract syntax. As a prerequisite to the elaboration of HILECOP-generated designs (i.e, resulting from the transformation of a SITPN into an $\mathcal{H}$-VHDL design), a particular design store $\mathcal{D}_{\mathcal{H}}$ is defined. Design store $\mathcal{D}_{\mathcal{H}}$ binds the `transition` and `place` identifiers to the definition of the `place` and `transition` designs in $\mathcal{H}$-VHDL abstract syntax:*

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

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

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

DESIGNELAB

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

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

2.5.2 Generic clause elaboration

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

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

Premises

- $etype_g$ transforms a type indication, specifically attached to a generic constant declaration, into a *type* instance and checks its well-formedness (see Section 2.5.5).
- The e relation links an expression e to its value v in a given context (see Section 2.6.9). The context of evaluation for an expression is composed of a given elaborated design, a given design state, and given local environment. We omit symbols at the left of the thesis when they refer to empty structures. For instance, $\vdash e \xrightarrow{e} v$ is a notation for $\Delta_\emptyset, \sigma_\emptyset, \Lambda_\emptyset \vdash \mathrm{e} \xrightarrow{e} v$.
- SE_l states that an expression is *locally* static (see Section 2.5.9).
- $v \in_c T$ and $\mathcal{M}(\mathrm{id}_g) \in_c T$ checks that the default value and the value of yielded by the dimensioning function belongs to the type of the declared generic constant (see Section 2.5.8).

Side conditions

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

GENELABDIMEN

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

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

GenElabDefault

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

GenElabComp

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

2.5.3 Port clause elaboration

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

Premises

- The *etype* relation associates a type indication to its corresponding semantic type and checks its well-formedness (see Section 2.5.5).
- The *defaultv* relation associates a given semantic type to its implicit *default* value.

InPortElab

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

OutPortElab

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

PortElabComp

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

2.5.4 Architecture declarative part elaboration

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

SigElab

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

SigElabComp

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

2.5.5 Type indication elaboration

The *etype* relation checks the well-formedness of a type indication τ, and transforms it into a semantic *type* (as defined in Table 2.2). A type indication τ is well-formed in the context Δ if τ denotes the `boolean` keyword or the `natural` or `array` keywords with a *well-formed* constraint, and a well-formed element type in the `array` case.

ETypeBool

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

ETypeNat

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

ETypeArray

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

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

- its bounds are globally static expressions [13, p.36] conforming to the $nat(0, \mathtt{NATMAX})$ type after evaluation.
- its lower bound value is inferior or equal to its upper bound value.

Remark 1 (Type of constraints)**.** *As the VHDL language reference stays unclear about the type of range and index constraints [13, p.33], we add the restriction that range and index constraints must have bounds of the* $nat(0, \mathtt{NATMAX})$ *type, i.e. the interval of natural numbers representable with the VHDL language.*

Premises

- The $\in_c$ relation states that a given value conforms to a given type (see Section 2.5.5).
- The SE_g relation states that an expression is *globally* static (see Section 2.5.9).

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

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

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

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

- its bounds are locally static expressions [13, p.36] conforming to the $nat(0, \mathtt{NATMAX})$ type after evaluation.
- its lower bound value is inferior or equal to its upper bound value.

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

2.5.6 Behavior elaboration

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

Elaboration of concurrent statements

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

CsParElab

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

CsNullElab

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

Process statement elaboration

To elaborate a process statement, the *ebeh* relation associates the process identifier with a local environment in the *Ps* sub-environment of Δ. The *ebeh* builds the local environment from the process's local variable declaration list (see the *evars* relation). The *ebeh* relation also checks that the sequential statements composing the body of the process are well-typed (see the $valid_{ss}$ relation in Section 2.5.11).

Premises

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

Side conditions

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

PsElab

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

Process declarative part elaboration

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

VarElab

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

VarElabComp

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

Component instantiation statement elaboration

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

Premises

- The *emapg* relation builds a function $\mathcal{M} \in \textit{generic-id} \nrightarrow \textit{value}$ out of a generic map (see definition below).
- $\texttt{valid}_{ipm}$ (resp. $\texttt{valid}_{opm}$) states that an input port map (resp. output port map) is valid, i.e well-formed and well-typed (see Section 2.5.10).

Side conditions

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

CompElab

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

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

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

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

$$\textsc{AssocGElab}\quad \frac{SE_l(\mathrm{e}) \qquad \vdash \mathrm{e} \xrightarrow{e} v}{\mathcal{M} \vdash\ (\mathrm{id}_g, \mathrm{e}) \xrightarrow{emapg} \mathcal{M} \cup (\mathrm{id}_g, v)}\ \mathrm{id}_g \notin \mathcal{M} \qquad \textsc{GMElab}\quad \frac{\mathcal{M} \vdash \mathrm{assoc}_g \xrightarrow{emapg} \mathcal{M}' \qquad \mathcal{M}' \vdash \mathrm{gmap} \xrightarrow{emapg} \mathcal{M}''}{\mathcal{M} \vdash \mathrm{assoc}_g,\ \mathrm{gmap} \xrightarrow{emapg} \mathcal{M}''}$$

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

2.5.7 Implicit default value

According to the VHDL LRM, at the declaration of a port, a signal or a variable, these items must receive an implicit default value depending on their types [13, p.61, 64, 173]. The $defaultv$ relation determines the default value for a given type.

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

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

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

2.5.8 Typing relation

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

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

2.5.9 Static expressions

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

Locally static expressions

An expression is *locally* static if:

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

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

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

Globally static expressions

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

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

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

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

2.5.10 Valid port map

Valid input port map

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

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

Premises

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

VALIDIPM

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

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

Side conditions

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

LISTIPMSIMPLE

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

Premises

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

Side conditions

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

ListIPMPartial

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

ListIPMCons

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

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

$$\begin{aligned} \mathtt{check}_{pm}(Ports, \mathcal{L}) \equiv \forall \mathrm{id}_f \in \mathtt{dom}(Ports),\ \mathrm{id}_f \in \mathcal{L} \lor (&Ports(\mathrm{id}_f) = \mathtt{array}(T,n,m) \land \\ &\forall i \in [n,m],\ (\mathrm{id}_f, i) \in \mathcal{L}) \end{aligned}$$

Valid output port map

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

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

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

Premises

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

VALIDOPM

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

Side conditions

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

LISTOPMSIMPLETOSIMPLE

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

Side conditions

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

LISTOPMSIMPLETOPARTIAL

$$\frac{SE_l(\text{e}_i) \qquad \vdash \text{e}_i \xrightarrow{e} v_i \qquad v_i \in_c nat(n, m)}{\Delta, \Delta_c, \mathcal{L}, \mathcal{L}_{ids} \vdash (\text{id}_f, \text{id}_a(\text{e}_i)) \xrightarrow{list_{opm}} \mathcal{L} \cup \{id_f\}, \mathcal{L}_{ids} \cup \{(id_a, v_i)\}} \quad \begin{array}{l} \text{id}_f \notin \mathcal{L}, \text{id}_a, (\text{id}_a, v_i) \notin \mathcal{L}_{ids} \\ \text{id}_f \in Outs(\Delta_c) \\ \text{id}_a \in Sigs(\Delta) \cup Outs(\Delta) \\ \Delta_c(\text{id}_f) = T \\ \Delta(\text{id}_a) = array(T, n, m) \end{array}$$

LISTOPMSIMPLETOOPEN

$$\frac{}{\Delta, \Delta_c, \mathcal{L}, \mathcal{L}_{ids} \vdash (\text{id}_f, \mathtt{open}) \xrightarrow{list_{opm}} \mathcal{L} \cup \{id_f\}, \mathcal{L}_{ids}} \quad \begin{array}{l} \text{id}_f \notin \mathcal{L} \\ \text{id}_f \in Outs(\Delta_c) \end{array}$$

Remark 3 (Unconnected output port.). *We forbid the case where an indexed formal part corresponding to the subelement of a composite output port is unconnected, i.e* $(\mathrm{id}_f(e_i), \texttt{open})$*, as it could lead to the case where some subelements of a composite output port are connected while others are not (error case in [13, p.7]).*

ListOPMPartialToSimple

$$\frac{SE_l(\mathrm{e}_i) \quad \vdash \mathrm{e}_i \xrightarrow{e} v_i \quad v_i \in_c nat(n,m)}{\Delta, \Delta_c, \mathcal{L}, \mathcal{L}_{ids} \vdash (\mathrm{id}_f(\mathrm{e}_i), \mathrm{id}_a) \xrightarrow{list_{opm}} \mathcal{L} \cup \{(id_f, v_i)\}, \mathcal{L}_{ids} \cup \{id_a\}} \quad \begin{array}{l} \mathrm{id}_f, (\mathrm{id}_f, v_i) \notin \mathcal{L}, \mathrm{id}_a \notin \mathcal{L}_{ids} \\ \mathrm{id}_f \in Outs(\Delta_c) \\ \mathrm{id}_a \in Sigs(\Delta) \cup Outs(\Delta) \\ \Delta_c(\mathrm{id}_f) = array(T,n,m) \\ \Delta(\mathrm{id}_a) = T \end{array}$$

ListOPMPartialToPartial

$$\frac{\begin{array}{lll} SE_l(\mathrm{e'}_i) & \vdash \mathrm{e'}_i \xrightarrow{e} v'_i & v'_i \in_c nat(n',m') \\ SE_l(\mathrm{e}_i) & \vdash \mathrm{e}_i \xrightarrow{e} v_i & v_i \in_c nat(n,m) \end{array}}{\Delta, \Delta_c, \mathcal{L}, \mathcal{L}_{ids} \vdash (\mathrm{id}_f(\mathrm{e}_i), \mathrm{id}_a(\mathrm{e'}_i)) \xrightarrow{list_{opm}} \begin{array}{l} \mathcal{L} \cup \{(id_f, v_i)\}, \\ \mathcal{L}_{ids} \cup \{(id_a, v'_i)\} \end{array}} \quad \begin{array}{l} \mathrm{id}_f, (\mathrm{id}_f, v_i) \notin \mathcal{L}, \mathrm{id}_a, (\mathrm{id}_a, v'_i) \notin \mathcal{L}_{ids} \\ \mathrm{id}_f \in Outs(\Delta_c) \\ \mathrm{id}_a \in Sigs(\Delta) \cup Outs(\Delta) \\ \Delta_c(\mathrm{id}_f) = array(T,n,m) \\ \Delta(\mathrm{id}_a) = array(T,n',m') \end{array}$$

ListOPMCons

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

2.5.11 Valid sequential statements

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

Well-typed signal assignment

Premises

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

WTSig

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

WTIdxSig

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

Well-typed variable assignment

WTVar

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

WTIdxVar

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

Well-typed if statements

WTIf

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

WTIfElse

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

Well-typed loop statement

WTLoop

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

Well-typed rising and falling edge blocks

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

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

Well-typed rst blocks

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

Well-typed null statement

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

2.6 Simulation rules

In this section, we formalize a specific simulation algorithm for the $\mathcal{H}$-VHDL designs. This algorithm is much simpler than the one presented in the LRM. This is mostly due to the fact that $\mathcal{H}$-VHDL is a subset of VHDL that aims at the description of synthesizable and synchronous designs. Synthesizable designs mean that the only kind of signal assignment used to describe the design behaviors are δ-delay signal assignments. Leaving apart the synchronous side, we only need a simulation algorithm that performs *delta cycles* (see Section 2.1.2) to simulate such synthesizable designs. However, $\mathcal{H}$-VHDL designs are also synchronous designs. As such, an $\mathcal{H}$-VHDL design is equipped with a clock input port. The value of the clock input port changes from 0 to 1 and inversely at constant rate, i.e. the clock rate. One can see the changing of the value of the clock input port as the result of the execution of a unit-delay signal assignment where the time clause is equal to half the clock period. Listing 2.4 illustrates how a $\mathcal{H}$-VHDL design `tl` can be embedded in another top-level design with a process regulating the value of a clock signal by using a unit-delay signal assignment. Listing 2.4 presents the behavioral part the embedding top-level design.

```
architecture toplevel_arch of toplevel is
begin

  clkp : process (clock)
  begin
    clock ⇐ not clock after τ -- where τ is half a clock period
  end process clkp;

  $id_{tl}$ : entity tl
  generic map (...)
  port map (clock ⇒ clock, ...);

end toplevel_arch;
```

LISTING 2.4: An architecture to simulate a synchronous design. The architecture `toplevel_arch` is composed of the `clkp` process, that simulates a clock signal, and of an instance of the design `tl` named id_{tl}, i.e. the design under simulation.

In Listing 2.4, the `clkp` process assigns the clock signal with its inverse value after τ unit of time where τ corresponds to half the clock period. Of course, the clock period is specified by the designer of the circuit. The component instance id_{tl} corresponds to the instantiation of the $\mathcal{H}$-VHDL design `tl`, i.e. the one we want to simulate. The `clock` input port of id_{tl} is connected to the `clock` signal of the embedding design. Thus, when the value of the clock signal changes every half clock period, the processes that react to the changes of the clock signal, i.e. the so-called *synchronous* processes, are executed in the internal behavior of the component instance id_{tl}. Then, it is the turn of *combinational* processes to be executed until stabilization of all signal values. Using the terms of the LRM simulation algorithm, what will happen when trying to simulate the design of Listing 2.4 will be an alternation between one time cycle to move to the next clock event and execute synchronous processes, followed by many delta cycles corresponding to the execution of combinational processes until stabilization. Thus, we choose to embed this alternation within the definition of our simulation algorithm.

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

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

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

```
   // Initialization phase.
1  $\sigma'_e \leftarrow$ RunAllOnce($\Delta$,$\sigma_e$,$cs$)
2  $\sigma \leftarrow$ Stabilize($\Delta$,$\sigma'_e$,$cs$)
   // Main loop.
3  $T_c \leftarrow 0$
4  $\theta \leftarrow [\sigma]$
5  while $T_c \leq nbOfCycles$ do
6      $\sigma_i \leftarrow$ Inject$_\uparrow$($\Delta$,$\sigma$,$E_p$,$T_c$)
7      $\sigma_\uparrow \leftarrow$ RisingEdge($\Delta$,$\sigma_i$,$cs$)
8      $\sigma' \leftarrow$ Stabilize($\Delta$,$\sigma_\uparrow$,$cs$)
9      $\sigma'_i \leftarrow$ Inject$_\downarrow$($\Delta$,$\sigma'$,$E_p$,$T_c$)
10     $\sigma_\downarrow \leftarrow$ FallingEdge($\Delta$,$\sigma'_i$,$cs$)
11     $\sigma \leftarrow$ Stabilize($\Delta$,$\sigma_\downarrow$,$cs$)
12     $\theta \leftarrow \theta +\!\!+ [\sigma',\sigma]$
13     $T_c \leftarrow T_c + 1$
14 return $\theta$
```

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

2.6.1 Full simulation

The full simulation process is decomposed in two steps. The first step is the elaboration phase that builds an elaborated version of a $\mathcal{H}$-VHDL design along with its default state. Previous to the elaboration phase, the top-level design receives a value for each of its generic constant; we refer to it as the *dimensioning* of the top-level design. The second step is the simulation phase that executes the behavioral part of the top-level design starting from an initial state. The initial state is built by a specfic initialization phase. Then, the simulation is decomposed into simulation cycles. Each simulation cycle is divided in four parts entailed by the *synchronous* execution of $\mathcal{H}$-VHDL top-level designs, i.e designs whose behavior depend on a clock signal. The four parts are, first, the execution of concurrent statements responding to the rising edge of the clock signal, then, a phase of signal stabilization followed by the execution of concurrent statements responding to the falling edge of the clock signal, and finally another phase of signal stabilization. At each clock event, the value of the primary inputs of the design are captured and injected in the simulation; primary inputs receive values from the design environment. Here, the environment is represented by a function mapping input port identifiers to values depending on the current count of simulation cycles and the considered clock event.

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

FULLSIM

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

where:

- $\mathcal{M}_g \in Gens(\Delta) \nrightarrow value$, the function yelding the values of generic constants for a given top-level design, refered to as the *dimensioning* function. Here, $Gens(\Delta)$ denotes the domain of $Gens(\Delta)$, i.e. the set of generic constant identifiers of Δ.

- $E_p \in (\mathbb{N} \times Clk) \rightarrow (Ins(\Delta) \rightarrow value)$, the function yelding a mapping from primary inputs (i.e, input ports of the top-level design) to values at a given simulation cycle count (i.e, the $\mathbb{N}$ argument), and a given clock event (i.e, the Clk argument, where $Clk = \{\uparrow, \downarrow\}$). Here, $Ins(\Delta)$ denotes the domain of $Ins(\Delta)$, i.e. the set of input port identifiers of Δ.

- τ, the number of simulation cycles to execute. The value of τ is decremented at each clock cycle until it reaches zero (see Section 2.6.2).

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

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

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

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

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

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

2.6.2 Simulation loop

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

$$\textsc{SimEnd}\quad \frac{}{\mathcal{D}, E_p, \Delta, 0, \sigma \vdash cs \rightarrow [\,]}$$

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

2.6.3 Simulation cycle

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

Notation 3 (Overriding union)**.** *For all partial function $f, f' \in X \nrightarrow Y$, $f \overset{\leftarrow}{\cup} f'$ denotes the overriding union of f and f' such that* $f \overset{\leftarrow}{\cup} f'(x) = \begin{cases} f'(x) & \textit{if } x \in \mathtt{dom}(f') \\ f(x) & \textit{otherwise} \end{cases}$

Notation 4 (Differentiated intersection domain)**.** *For all partial function $f, f' \in X \nrightarrow Y$, $f \overset{\neq}{\cap} f'$ denotes the intersection of the domain of f and f' for which f and f' yields different values. That is,* $f \overset{\neq}{\cap} f' = \{\, x \in \mathtt{dom}(f) \cap \mathtt{dom}(f') \mid f(x) \neq f'(x) \,\}$.

Definition 4 (Input port values update)**.** *Given a simulation environment $E_p \in (\mathbb{N} \times \{\uparrow, \downarrow\}) \rightarrow (input\text{-}id \rightarrow value)$, let us define the relation expressing the update of the value of input ports at a given design state $\sigma \in \Sigma$, clock cycle count $\tau \in \mathbb{N}$, and clock event $clk \in \{\uparrow, \downarrow\}$, and thus resulting in a new state $\sigma_i \in \Sigma$. The relation is written* $\mathtt{Inject}_{clk}(\sigma, E_p, \tau, \sigma_i)$ *and verifies that:* $\sigma = <\mathcal{S}, \mathcal{C}, \mathcal{E}>$ *and* $\sigma_i = <\mathcal{S} \overleftarrow{\cup} E_p(\tau, clk), \mathcal{C}, \mathcal{E}>$.

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

SIMCYC

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

2.6.4 Initialization rules

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

INIT

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

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

The *runinit* relation is defined by the Rules PSRUNINIT, COMPRUNINIT, PARRUNINIT and NULLRUNINIT which are detailled right below. The *stabilize* relation is defined in Section 2.6.6.

Evaluation of a process statement

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

Premises

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

Side conditions

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

PsRunInit

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

Evaluation of a component instantiation statement

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

Premises

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

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

Side conditions

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

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

Evaluation of the composition of concurrent statements

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

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

The `merge` function, defined in Listing in pseudo-Coq language, computes a new state based on the original state o, and the states s and s' yielded by the computation of two concurrent statements. In the resulting state, the signal value store $\mathcal{S}_m$ is a function merging together the signal stores at state o, s and s'. S_m yields values from the signal store $\mathcal{S}$ (resp. $\mathcal{S}'$) for all signal that belongs to the set of events at state s (resp. s'), and yields values from the original signal store $\mathcal{S}_o$ for all eventless signals. The same goes for the merged component store C_m. The new set of events $\mathcal{E}_m$ is the union between the set of events at state s and the one at state s'. The `merge` function correctly merges the state o, s and s' only if the set of events of s and s' are disjoint. The ParRunInit rule, which appeals to the `merge` function, defines the condition of disjoint set of events as a side condition.

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

Listing 2.5: The `merge` function that fuses together an origin state o, with two states s and s' generated by the execution of two concurrent statements.

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

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

NullRunInit

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

2.6.5 Clock phases rules

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

Evaluation of a process statement

The following rules describe the evaluation of a process statement at the occurrence of the rising or the falling edge of the clock signal. In the case where a process does not contain the clk identifier in its sensitivity list, then its statement body is not executed during the clock phases (see Rules PsRENoClk and PsFENoClk). Otherwise, its statement body is executed. Depending on the considered clock event, falling blocks or rising blocks are executed when encountered in the body of a process (see Rules PsREClk and PsFEClk).

PsRENoClk

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

Premises

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

PsREClk

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

PsFENoClk

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

Premises

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

PsFEClk

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

Evaluation of a component instantiation statement

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

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

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

Evaluation of the composition of concurrent statements

The following rules describe the evaluation of the composition of concurrent statements and the evaluation of null statements during the clock phases. These rules are similar to the ones described for the initialization phase. Thus, the reader can refer to Section 2.6.4 for more details.

$$\textsc{ParFE}$$
$$\dfrac{\mathcal{D}, \Delta, \sigma \vdash \mathrm{cs} \xrightarrow{\downarrow} \sigma' \qquad \mathcal{D}, \Delta, \sigma \vdash \mathrm{cs}' \xrightarrow{\downarrow} \sigma''}{\mathcal{D}, \Delta, \sigma \vdash \mathrm{cs} \mid\mid \mathrm{cs}' \xrightarrow{\downarrow} \texttt{merge}(\sigma, \sigma', \sigma'')} \quad \mathcal{E}' \cap \mathcal{E}'' = \emptyset \qquad \qquad \textsc{NullFE} \quad \dfrac{}{\Delta, \sigma \vdash \texttt{null} \xrightarrow{\downarrow} \sigma}$$

$$\textsc{ParRE}$$
$$\dfrac{\mathcal{D}, \Delta, \sigma \vdash \mathrm{cs} \xrightarrow{\uparrow} \sigma' \qquad \mathcal{D}, \Delta, \sigma \vdash \mathrm{cs}' \xrightarrow{\uparrow} \sigma''}{\mathcal{D}, \Delta, \sigma \vdash \mathrm{cs} \mid\mid \mathrm{cs}' \xrightarrow{\uparrow} \texttt{merge}(\sigma, \sigma', \sigma'')} \quad \mathcal{E}' \cap \mathcal{E}'' = \emptyset \qquad \qquad \textsc{NullRE} \quad \dfrac{}{\Delta, \sigma \vdash \texttt{null} \xrightarrow{\uparrow} \sigma}$$

2.6.6 Stabilization rules

The following rules describe the evaluation of concurrent statements, representing a design's behavior, during a stabilization phase. The stabilization phase triggers the execution of the combinational parts of the behavior by appealing to the *comb* relation. When the execution of the combinational parts of the behavior does not change the design state anymore, then we have reached a stable state and the stabilization phase ends (Rule StabilizeEnd). When the execution of the combinational parts produces some events, i.e. it changes the value of signals or the internal state of component instances, then the stabilization phase must continue until a stable state is reached (Rule StabilizeLoop). In the formalization of the $\mathcal{H}$-VHDL simulation algorithm, the set of events of a design state is useful to merge the states resulting from the execution of multiple concurrent statements (see Definition 2.5), and to determine if a stable state has been reachd. In the LRM simulation algorithm, the kernel process uses the set of events to resume the activity of processes. If one of the signal declared in a process' sensitivity list is registered in the current set of events, then the process body must be executed. We choose to disregard this aspect of the execution of process in the formalization of our simulation algorithm. Thus, all combinational processes are executed when a delta cycle is performed, i.e. through the use of the *comb* relation.

Side conditions

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

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

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

Evaluation of a process statement

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

Premises

- The c flag (for *combinational*) on the ss_c relation indicates that statements responding to clock events (i.e. `falling` and `rising` blocks) and statements executed during the initialization phase only (i.e. `rst` blocks) will not be considered.
- The set of events of state σ is emptied ($NoEv(\sigma)$, see Notation 2) before the evaluation of the process statement body. It corresponds to the consumption of the information brought by the event set. Once the information has been consumed, new events can be generated by executing the process body. Otherwise, the set of events is never empty, and a stable state is never reached.

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

Evaluation of a component instantiation statement

Rule CompComb describes the evaluation of a component instantiation statement during a stabilization phase. This rule is similar in every point to Rule CompRunInit, and Rules CompRE and CompFE, that describe the evaluation of a component instantiation statement during the initialization phase, and the clock phases. The only difference lies in the execution of the internal behavior of the component instance. During a stabilization, the *comb* relation evaluates the internal behavior of component instances. Otherwise, see Section 2.6.4 for more details about the premises and side conditions of Rule CompComb.

CompComb

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

Evaluation of the composition of concurrent statements

The following rules describe the evaluation of the composition of concurrent statements and the evaluation of null statements during a stabilization phase. These rules are similar to the ones describe for the initialization phase. Thus, the reader can refer to Section 2.6.4 for more details.

ParComb

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

NullComb

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

2.6.7 Evaluation of input and output port maps

Evaluation of an input port map

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

MapipSimple

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

MAPIPPARTIAL

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

MAPIPCOMP

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

Evaluation of an output port map

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

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

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

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

OUTO

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

IDXOUTO

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

The following rules define the *mapop* relation.

MAPOPOPEN

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

Side conditions

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

MAPOPSIMPLETOSIMPLE

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

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

MAPOPSIMPLETOPARTIAL

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

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

MAPOPPARTIALTOSIMPLE

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

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

MAPOPPARTIALTOPARTIAL

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

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

MAPOPCOMP

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

2.6.8 Evaluation of sequential statements

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

Signal assigment statement

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

Side conditions

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

SIGASSIGN

$$\frac{\Delta, \sigma, \Lambda \vdash \mathrm{e} \xrightarrow{e} v \qquad v \in_c T}{\Delta, \sigma, \sigma_w, \Lambda \vdash \mathrm{id}_s \Leftarrow \mathrm{e} \xrightarrow{ss} <\mathcal{S}'_w, \mathcal{C}_w, \mathcal{E}'_w>, \Lambda} \quad \begin{array}{l} \mathrm{id}_s \in Sigs(\Delta) \cup Outs(\Delta) \\ \Delta(\mathrm{id}_s) = T \\ \mathcal{S}'_w = \mathcal{S}_w(id_s) \leftarrow v \\ \mathcal{E}'_w = \mathcal{E}_w \cup (\mathcal{S} \overset{\neq}{\cap} \mathcal{S}'_w) \end{array}$$

IDXSIGASSIGN

$$\frac{\begin{array}{ll} \Delta, \sigma, \Lambda \vdash \mathrm{e}_i \xrightarrow{e} v_i & v \in_c T \\ \Delta, \sigma, \Lambda \vdash \mathrm{e} \xrightarrow{e} v & v_i \in_c nat(n, m) \end{array}}{\Delta, \sigma, \sigma_w, \Lambda \vdash \mathrm{id}_s(\mathrm{e}_i) \Leftarrow \mathrm{e} \xrightarrow{ss} <\mathcal{S}'_w, \mathcal{C}_w, \mathcal{E}'_w>, \Lambda} \quad \begin{array}{l} \mathrm{id}_s \in Sigs(\Delta) \cup Outs(\Delta) \\ \Delta(\mathrm{id}_s) = array(T, n, m) \\ \mathcal{S}'_w = \mathcal{S}_w(id_s) \leftarrow \texttt{set\_at}(v, v_i, \mathcal{S}_w(id_s)) \\ \mathcal{E}'_w = \mathcal{E}_w \cup (\mathcal{S} \overset{\neq}{\cap} \mathcal{S}'_w) \end{array}$$

Variable assignment statement

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

VARASSIGN

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

IDXVARASSIGN

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

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

If statement

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

IF$\top$

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

IF$\bot$

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

IFELSE$\top$

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

IFELSE$\bot$

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

Loop statement

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

LOOP$\bot$

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

LOOP⊤

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

LOOPINIT

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

Rising and falling edge block statements

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

RISINGEDGEDEFAULT

$$\dfrac{}{\Delta, \sigma, \sigma_w, \Lambda \vdash \mathtt{rising}\ \mathrm{ss} \xrightarrow{ss_f} \sigma_w, \Lambda} \quad f \neq \uparrow$$

FALLINGEDGEDEFAULT

$$\dfrac{}{\Delta, \sigma, \sigma_w, \Lambda \vdash \mathtt{falling}\ \mathrm{ss} \xrightarrow{ss_f} \sigma_w, \Lambda} \quad f \neq \downarrow$$

RISINGEDGEEXEC

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

FALLINGEDGEEXEC

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

Rst block statement

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

RSTDEFAULT

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

RSTEXEC

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

Composition of sequential statements and null statement

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

$$\textsc{SeqStmt}\quad \frac{\Delta,\sigma,\sigma_w,\Lambda \vdash \mathrm{ss} \xrightarrow{ss} \sigma'_w,\Lambda' \qquad \Delta,\sigma,\sigma'_w,\Lambda' \vdash \mathrm{ss'} \xrightarrow{ss} \sigma''_w,\Lambda''}{\Delta,\sigma,\sigma_w,\Lambda \vdash \mathrm{ss};\mathrm{ss'} \xrightarrow{ss} \sigma''_w,\Lambda''}$$

$$\textsc{NullStmt}\quad \frac{}{\Delta,\sigma,\sigma_w,\Lambda \vdash \texttt{null} \xrightarrow{ss} \sigma_w,\Lambda}$$

2.6.9 Evaluation of expressions

Here, we present the evaluation of expressions used throughout the definition of $\mathcal{H}$-VHDL designs. Rules Nat, False and True describe the evaluation of natural number and boolean constants. Rule Aggreg describes the evaluation of an aggregate expression. Rule Gen presents the evaluation of a generic constant identifier. Rules Sig and Var describe the evaluation of signal and variable identifiers. Rules IdxSig and IdxVar corresponds to the evaluation of indexed signal and indexed variable identifiers. In Rules IdxSig and IdxVar, $\texttt{get\_at}(i,a)$ is a function returning the i-th element of array a.

$$\textsc{Nat}\quad \frac{}{\Delta,\sigma,\Lambda \vdash \mathrm{n} \xrightarrow{e} n}\ \begin{array}{l}\mathrm{n} \in \mathbb{N}\\ \mathrm{n} \leq \texttt{NATMAX}\end{array}$$

$$\textsc{False}\quad \frac{}{\Delta,\sigma,\Lambda \vdash \texttt{false} \xrightarrow{e} \bot}$$

$$\textsc{True}\quad \frac{}{\Delta,\sigma,\Lambda \vdash \texttt{true} \xrightarrow{e} \top}$$

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

$$\textsc{Sig}\quad \frac{}{\Delta,\sigma,\Lambda \vdash \mathrm{id}_s \xrightarrow{e} \sigma(\mathrm{id}_s)}\ \mathrm{id}_s \in \mathit{Sigs}(\Delta) \cup \mathit{Ins}(\Delta)$$

$$\textsc{Var}\quad \frac{}{\Delta,\sigma,\Lambda \vdash \mathrm{id}_v \xrightarrow{e} v}\ \begin{array}{l}\mathrm{id}_v \in \Lambda\\ \Lambda(\mathrm{id}_v) = (T,v)\end{array}$$

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

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

IDXVAR

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

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

NATADD

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

NATSUB

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

ORDOP

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

BOOLBINOP

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

NOTOP

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

EQOP

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

DIFFOP

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

PARENTH

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

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

2.7 An example of full simulation

In this section, we will demonstrate the full simulation of a $\mathcal{H}$-VHDL top-level design on the example of Listing 2.6. The aim here is to illustrate the formal rules of the $\mathcal{H}$-VHDL semantics. Listing 2.6 is the result of the transformation of the SITPN model presented in Figure 2.6 into a $\mathcal{H}$-VHDL design. To keep the examples within a reasonable size, Listing 2.6, and the other listings and derivation trees used in this section, refer to the generic constants, the ports and the internal signals of the `transition` and `place` designs by their short names. See Table **??** for a correspondence between the short names and the full names of constants and signals of the `place` and `transition` designs. In Listing, the generated $\mathcal{H}$-VHDL design, named `tl`, declares its input and output ports at Line 7, and its internal signals at Line 10. The behavior of `tl` is defined by a `place` component instance id_p (Lines 15 to 23), a `transition` component instance id_t (Lines 28 to 16), and two processes, namely: the `marked` process (Lines 41 to 43), and the `fired` process (Lines 48 to 50).

```
1  design tl tla
2
3  -- Generic constants
4  ∅
5
6  -- Ports ($ports_{tl}$)
7  ((in, $id_{c0}$, boolean), (out, $id_{f0}$, boolean), (out, $id_{a0}$, boolean))
8
9  -- Declared (internal) signals ($sigs_{tl}$)
10 (($id_{ft}$, boolean), ($id_{av}$, boolean), ($id_{rt}$, boolean),($id_m$, boolean))
11
12 -- Behavior ($cs_{tl}$)
13
14 -- Place component instance $id_p$
15 comp ($id_p$, place,
16 -- Generic map
17 ((ian, 1), (oan, 1), (mm, 1)),
18
19 -- Input port map
20 ((im, 1), (iaw(0), 1), (oat(0), 0), (oaw(0), 1), (itf(0), $id_{ft}$), (otf(0), $id_{ft}$))
21
22 -- Output port map
23 ((oav(0), $id_{av}$), (pauths, open), (rtt(0), $id_{rt}$), (marked, $id_m$)))
24
```

```
||

-- Transition component instance $id_t$
comp ($id_t$, transition,
-- Generic map
((tt, 0), (ian, 1), (cn, 1)),

-- Input port map
((ic(0), $id_{c0}$), (A, 0), (B, 0), (iav(0), $id_{av}$), (rt(0), $id_{rt}$), (pauths(0), true)),

-- Output port map
((fired, $id_{ft}$)))

||

-- The marked process
process (marked, {clk}, ∅,
(rst ($id_{a0}$ ⇐ false)
     (falling ($id_{a0}$ ⇐ $id_m$ or false))))

||

-- The fired process
process (fired, {clk}, ∅,
(rst ($id_{f0}$ ⇐ false)
     (rising ($id_{f0}$ ⇐ $id_{ft}$ or false))))
```

LISTING 2.6: An example of $\mathcal{H}$-VHDL top-level design generated by the HILECOP transformation.

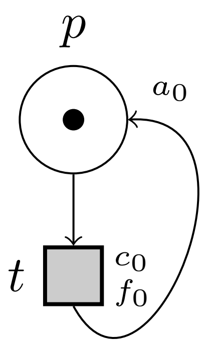

FIGURE 2.6: The SITPN model at the base of the generation of the top-level design presented in Listing 2.6.

The rule of Figure 2.7 states that the full simulation of the `tl` design (presented in Listing 2.6) over 1 clock cycle yields the simulation trace $(\sigma_0 :: \sigma_1 :: \sigma_2)$. The simulation over one clock cycle (the rightmost premise) yields a trace composed of two states: the state σ_1 at half the clock period, and the state σ_2 at the end of the first cycle. The full simulation happens in the context of the HILECOP's design store $\mathcal{D}_{\mathcal{H}}$, the elaborated design Δ, an empty dimensioning function and an simulation environment E_p. Here, $ports_{tl}$ is an alias for the list of ports of `tl`, $sigs_{tl}$ for the list of internal signals of `tl`, and cs_{tl} for the behavior of `tl`. In what follows, we will detail the premises of the FULLSIM rule.

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

FIGURE 2.7: The FULLSIM rule applied to the `tl` design.

2.7.1 Elaboration of the `tl` design

The rule of Figure 2.8 states that the elaborated design Δ and the default design state σ_e are the result of the elaboration of the `tl` design. From left to right in the premises of the rule, the three first premises pertain to the elaboration of the declarative parts of the `tl` design, i.e. the generic constant declaration list, the port declaration list and the internal signal declaration list. The rightmost premise pertains to the elaboration of the behavior of the `tl` design.

$$\dfrac{\dfrac{\vdots}{\emptyset, \emptyset \vdash gens_{tl} \xrightarrow{egens} \Delta_0} \quad \dfrac{\vdots}{\Delta_0, \emptyset \vdash ports_{tl} \xrightarrow{eports} \Delta_1, \sigma_{e1}} \quad \dfrac{\vdots}{\Delta_1, \sigma_{e1} \vdash sigs_{tl} \xrightarrow{esigs} \Delta_2, \sigma_{e2}} \quad \dfrac{\vdots}{\mathcal{D}_{\mathcal{H}}, \Delta_2, \sigma_{e2} \vdash cs_{tl} \xrightarrow{ebeh} \Delta, \sigma_e}}{\mathcal{D}_{\mathcal{H}}, \emptyset \vdash \texttt{design tl tla } gens_{tl}\ ports_{tl}\ sigs_{tl}\ cs_{tl} \xrightarrow{elab} \Delta, \sigma_e}$$

FIGURE 2.8: The DESIGNELAB rule applied to the `tl` design.

Elaboration of the declarative parts

The elaboration of the declarative parts populates the *Gens*, *Ins*, *Outs* and *Sigs* sub-environments of the elaborated design Δ. Here is the content of the *Gens*, *Ins*, *Outs* and *Sigs* sub-environments of Δ_2, where Δ_2 is the partial elaborated design after the elaboration of the declarative parts of the `tl` design (passed as a parameter of third and the fourth premises of the rule in Figure 2.8).

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

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

The elaboration of the declarative parts also gives a default value to the signals in the signal store of the default design state σ_{e2}, where σ_{e2} is the partial default design state at the end of the elaboration of the declarative parts of the `tl` design (passed as a parameter of the third and the fourth premises of the rule in Figure 2.8). Here is the content of the signal store $\mathcal{S}$ of σ_{e2}.

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

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

Elaboration of the behavioral part

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

Elaboration of a process statement

The rule of Figure 2.9 presents the elaboration of the `marked` process defined in the behavior of the `tl` design.

$$\dfrac{\dfrac{}{\Delta_2, \emptyset \vdash \emptyset \xrightarrow{evars} \emptyset} \qquad \dfrac{\vdots}{\Delta_2, \sigma_{e2}, \emptyset \vdash \texttt{valid}_{ss}\begin{pmatrix}\texttt{rst}\ (id_{a0} \Leftarrow \texttt{false})\\ (\texttt{falling}(id_{a0} \Leftarrow id_m\ \texttt{or}\ \texttt{false}))\end{pmatrix}}\ \textsc{WTRst}}{\mathcal{D}_{\mathcal{H}}, \Delta_2, \sigma_{e2} \vdash \texttt{process}(\texttt{marked}, \texttt{clk}, \emptyset, \dots) \xrightarrow{ebeh} \Delta_2 \cup (\texttt{marked}, \emptyset), \sigma_{e2}}\ \textsc{PsElab}$$

FIGURE 2.9: The elaboration of the `marked` process defined in the behavior of the `tl` design.

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

The elaboration of process statement also performs static type-checking on the process statement body leveraging the $\texttt{valid}_{ss}$ relation. The rule of Figure 2.10 details the static type-checking of the statement body of the `marked` process (rightmost premise of the rule presented in Figure 2.9). To keep the example within a reasonable size, we discard the context of some rules with it is not relevant. We annotate the rule names to describe the side conditions associated to a derivation.

$$\dfrac{\dfrac{\dfrac{}{\texttt{false} \xrightarrow{e} \bot}\,\textsc{False} \qquad \dfrac{}{\bot \in_c bool}\,\textsc{IsBool}}{\vdash \texttt{valid}_{ss}(id_{a0} \Leftarrow \texttt{false})}\,\textsc{WTSig}^1 \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\sigma_{e2} \vdash id_m \xrightarrow{e} \bot}\,\textsc{Sig}^2 \qquad \dfrac{}{\texttt{false} \xrightarrow{e} \bot}\,\textsc{False}}{\sigma_{e2} \vdash id_m \ \texttt{or false} \xrightarrow{e} \bot}\,\textsc{BoolBinOp} \qquad \dfrac{}{\bot \in_c bool}\,\textsc{IsBool}}{\sigma_{e2} \vdash \texttt{valid}_{ss}(id_{a0} \Leftarrow id_m \ \texttt{or false})}\,\textsc{WTSig}^1}{\sigma_{e2} \vdash \texttt{valid}_{ss}(\texttt{falling}(id_{a0} \Leftarrow id_m \ \texttt{or false}))}\,\textsc{WTFalling}}{\Delta_2, \sigma_{e2}, \emptyset \vdash \texttt{valid}_{ss}\begin{pmatrix}\texttt{rst}\ (id_{a0} \Leftarrow \texttt{false}) \\ (\texttt{falling}(id_{a0} \Leftarrow id_m \ \texttt{or false}))\end{pmatrix}}\,\textsc{WTRst}$$

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

FIGURE 2.10: Static type-checking of the `marked` process statement body.

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

Elaboration of a component instantiation statement

The rule of Figure 2.11 presents the elaboration of the place component instance id_p belonging to the behavior of the `tl` design.

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

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

FIGURE 2.11: The elaboration of the id_p component instance defined in the behavior of the `tl` design.

The elaboration of a component instantiation statement is divided in three parts. First, a dimensioning function is built out of the generic map of the component instance. Figure 2.12 shows a part of the creation of the dimensioning functioning $\mathcal{M}$ from the generic map of the component instance id_p. Basically, the elaboration of a generic map is a transformation from a syntactic construct, i.e. the generic map, into a function, i.e. the dimensioning function

$\mathcal{M}$. For each association of the generic map, the elaboration checks that the actual part of the association is a locally static expression (see Section 2.5.9).

$$\dfrac{\dfrac{\dfrac{}{SE_l(1)}\,\scriptstyle\text{LSENAT} \qquad \dfrac{}{1 \xrightarrow{e} 1}\,\scriptstyle\text{NAT}}{\emptyset \vdash (\mathtt{ian}, 1) \xrightarrow{emapg} \{(\mathtt{ian}, 1)\}}\,\scriptstyle\text{ASSOCGELAB}^1 \qquad \dfrac{\vdots}{\{(\mathtt{ian}, 1)\} \vdash (\mathtt{oan}, 1), (\mathtt{mm}, 1) \xrightarrow{emapg} \{(\mathtt{ian}, 1), (\mathtt{oan}, 1), (\mathtt{mm}, 1)\}}\,\scriptstyle\text{GMELAB}}{\emptyset \vdash (\mathtt{ian}, 1), (\mathtt{oan}, 1), (\mathtt{mm}, 1) \xrightarrow{emapg} \{(\mathtt{ian}, 1), (\mathtt{oan}, 1), (\mathtt{mm}, 1)\}}\,\scriptstyle\text{GMELAB}$$

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

FIGURE 2.12: The elaboration of the generic map of the id_p component instance defined in the behavior of the `tl` design.

The second step of the elaboration of a component instance is to retrieve from the design store the design associated with the component instance, and to elaborate this design. Here, the design store is the HILECOP design store $\mathcal{D}_{\mathcal{H}}$, and the design associated with id_p is the `place` design. The dimensioning function $\mathcal{M}$ sets the value of the generic constants declared in the `place` design. The full code of `place` design is available in Appendix **??**. In Figures 2.13 and 2.14, we give the elaborated design Δ_p and the default design state σ_p resulting of the elaboration of the `place` design given the dimensioning function $\mathcal{M}$.

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

FIGURE 2.13: An elaborated version of the place design built with the dimensioning function deduced from the generic map of the component instance id_p.

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

Figure 2.14 shows the default design state σ_p of Δ_p.

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

FIGURE 2.14: The default design state σ_p of the elaborated design Δ_p.

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

The final step in the elaboration of a component instantiation statement is to check the well-formedness and the well-typedness of the input and output port maps. The $\mathtt{valid}_{ipm}$ and $\mathtt{valid}_{opm}$ relations, defined in Section 2.5.10, state the validity of the port maps. The rule of Figure 2.15 presents a part of the construction of the $valid_{opm}$ relation applied to the output port map of the place component instance id_p. Note that Δ_p is necessary to check the validity of the output port map of id_p, as it holds the correspondence between port identifiers and port types.

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

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

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

FIGURE 2.15: An example of validity checking performed on the output port map of the place component instance id_p. The bottom proof tree represents the top-right premise of the top proof tree.

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

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

2.7.2 Simulation of the `tl` design

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

Initialization

The rule of Figure 2.16 presents the initialization phase in the proceeding of the simulation of the `tl` design. The initialization phase builds the initial state of the simulation. The first step of the initialization, formalized by the *runinit* relation, runs the processes and the internal behavior of component instances exactly once (with the execution of the first part of reset blocks). Then, a stabilization phase follows.

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

FIGURE 2.16: The initialization phase, first step of the simulation of the `tl` design.

The rule in Figure 2.17 presents the execution of the `tl` design's behavior during the *runinit* phase. The `tl` design's behavior is defined by the composition of concurrent statements. Here, the `marked` process is at the head of the behavior, whereas it is not the case in Listing 2.6. We formally proved, with the Coq proof assistant, that the $||$ composition operator for concurrent statements is commutative and associative with respect to the *runinit* relation. In Figure, the `marked` process is executed and yields the state σ'_e. Then, the rest of the `tl` design's behavior is executed and yields the state σ''_e. Finally, the starting state σ_e and the two states σ'_e and σ''_e are merged into one by the `merge` function.

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

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

FIGURE 2.17: The *runinit* phase applied to the concurrent statements composing the behavior of the `tl` design.

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

Execution of a process statement with the *runinit* relation

The rule in Figure 2.18 shows the execution of the `marked` process during the *runinit* phase. The first part of the reset block defining the statement body of the `marked` process is executed. This first part assigns the expression `false` to signal id_{a0}.

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

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

FIGURE 2.18: The *runinit* phase applied to the concurrent statements composing the behavior of the `tl` design.

In the side conditions of the SIGASSIGN rule, an new event set $\mathcal{E}'_e$ is computed based on the event set $\mathcal{E}_e$ joined to the expression $\mathcal{S}_e \overset{\neq}{\cap} \mathcal{S}'_e$. This expression returns the set of signals with a different value between signal store $\mathcal{S}_e$ and singal store $\mathcal{S}'_e$. The only signal that possibly has a different value from $\mathcal{S}_e$ to $\mathcal{S}'_e$ is the assigned signal id_{a0}. Thus, this expression is a shorthand to test if the value of signal id_{a0} has changed after the execution of the signal assignment statement. If it is the case, then the event set receives the signal identifier id_{a0}; id_{a0} is then an eventful signal. In the present case, the value of signal id_{a0} was $\bot$ at state σ_e and is still $\bot$ after the execution of the signal assignment statement. Therefore, no event is registered on signal id_{a0}. When states σ_e, σ'_e and σ''_e will be merged (cf. Figure 2.17), if id_{a0} is part of the event set of state σ''_e, then, the merged state will return the value associated to id_{a0} in state σ''_e. We would have $\texttt{merge}(\sigma_e, \sigma'_e, \sigma''_e)(id_{a0}) = \sigma''_e(id_{a0})$. However, signal id_{a0} would be a potentially multiply-driven signal because both the `marked` process and the concurrent statement cs'_{tl} (cf. Figure 2.17) both assign the signal value.

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

The rule of Figure 2.19 presents the execution of the `place` component instance id_p during the *runinit* phase. The execution of a component instantiation statement is pretty much the same in all the phases of the simulation algorithm. The difference lies in the choice of the relation used to execute of the internal behavior of the component instance. During the *runinit* phase, it is the *runinit* relation that executes the internal behavior of component instances; during the falling edge phase, it is the $\downarrow$ relation that executes the internal behaviors, etc.

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

$$\sigma'_e = <\mathcal{S}'_e, \mathcal{C}'_e, \mathcal{E}'_e>$$

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

$$\mathcal{C}''_e = \mathcal{C}'_e(id_p) \leftarrow \sigma''_p$$

FIGURE 2.19: The execution of the `place` component instance id_p during the *runinit* phase (first part of the initialization).

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

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

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

The rule of Figure 2.20 gives a part of the evaluation of the input port map of id_p.

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

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

FIGURE 2.20: The evaluation of the input port map of the `place` component instance id_p.

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

The rule of Figure 2.21 gives a part of the evaluation of the output port map of id_p.

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

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

FIGURE 2.21: The evaluation of the output port map of the `place` component instance id_p.

Stabilization

A stabilization phase happens after the *runinit* phase during the initialization phase, but also after the rising edge phase and the falling edge phase in the course of a simulation cycle. The stabilization phase executes the combinational parts of the design's behavior. The `tl` design holds no combinational processes in its behavior. The `marked` and `fired` processes are both synchronous. To illustrate the execution of a combinational process during a stabilization phase, let us consider the `fired_evaluation` process defined in the behavior of the `transition` design. The `fired_evaluation` process will be executed with the internal behavior of the `transition` component instance id_t during the stabilization phase. The rule of Figure 2.22 presents the execution of the internal behavior of the `transition` component instance id_t. As shown, the internal behavior cs_t is executed three times before reaching a stable state. Here, the number of execution before stabilization is arbitrary. In Figure 2.22, σ_{t0} corresponds to the state of id_t after the *runinit* phase and after the evaluation of its input port map. Remember that the evaluation of the input port map of a component instance always precedes the execution of the internal behavior of the component. Since σ_{t0} and σ_{t1} are not stable states, it means that their event set is not empty. Thus, we have $\mathcal{E}(\sigma_{t0}) \neq \emptyset$ and $\mathcal{E}(\sigma_{t1}) \neq \emptyset$. On the contrary, σ_{t2} is a stable state, and thus, $\mathcal{E}(\sigma_{t2}) = \emptyset$.

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

FIGURE 2.22: Three rounds of execution of the combinational parts of the `transition` component instance id_t during a stabilization phase.

Now, let us illustrate the execution of the `fired_evaluation` process, which is a part of cs_t, happening during the stabilization phase. The rule of Figure 2.23 details the execution of the body of process `fired_evaluation` performed by the *comb* relation.

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

$$(1)\ \Delta_t(\texttt{fired\_evaluation}) = \emptyset \qquad (2)\ \begin{array}{l}\Delta_t(\texttt{fired}) = bool\\ NoEv(\sigma_{t0}) = <\mathcal{S},\mathcal{C},\emptyset>\\ \mathcal{S}' = \mathcal{S}(\texttt{fired}) \leftarrow \bot\\ \sigma'_{t0} = <\mathcal{S}',\mathcal{C},\mathcal{S} \overset{\neq}{\cap} \mathcal{S}'>\end{array}$$

FIGURE 2.23: The execution of the `fired_evaluation` process during a stabilization phase. The `fired_evaluation` process is defined in the `transition` design's behavior.

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

Simulation cycle and clock phases

We describe here the execution of the `tl` design over one clock cycle. After the initialization phase, the design under simulation will execute τ simulation cycles, where τ is a natural numbers passed as a parameter. In the rule of Figure 2.24, τ equals 1. Thus, the behavior of the `tl` design is executed during one clock cycle and then the simulation ends. In Figure 2.24, σ_0 is the initial simulation state, i.e. the one at the end of the initialization phase. One simulation cycle yields two states σ_1 and σ_2, where σ_1 is the state in the middle of the first cycle, and σ_2 is the state at the end of the first cycle. The resulting simulation trace is only composed of states σ_1 and σ_2.

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

(1) $1 > 0$

FIGURE 2.24: The execution of the `tl` design's behavior during one clock cycle.

The rule of Figure 2.25 zooms in the first simulation cycle. The state σ_1 is produced by a rising edge phase followed by a stabilization phase. The state σ_2 is produced by a falling edge phase followed by a stabilization phase. The value of the primary input ports of the `tl` design are updated before each clock event phase. States σ_{0i} and σ_{1i} are the new states obtained after the update of the primary input port values. The update corresponds to the capture of values yielded by the given simulation environment E_p. The `tl` design has only one primary input port, i.e. the input port id_{c0}. The value of id_{c0} at state σ_{0i} is equal to the value yielded by the environment E_p at the rising edge of clock during the first clock cycle. Thus, we have $\sigma_{0i}(id_{c0}) = E_p(1, \uparrow)(id_{c0})$. To perform the proof that the HILECOP transformation is semantic preserving, we need the hypothesis that the primary input ports of the top-level design stay stable during a clock cycle. For instance, the value of id_{c0} must be the same during the first clock cycle independently of the considered clock event. Thus, we have $E_p(1, \uparrow)(id_{c0}) = E_p(1, \downarrow)(id_{c0})$. In this setting, we only need to capture the value of primary input ports at the beginning a clock cycle. However, to keep the definition of the $\mathcal{H}$-VHDL semantics as general as possible, we preserve the update of primary input ports at each clock event.

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

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

FIGURE 2.25: Details of the execution of the `tl` design's behavior during the first clock cycle.

The rule of Figure 2.26 describes the execution of the `fired` process, defined in the `tl` design's behavior, during the rising edge phase of the first simulation cycle. During the rising edge phase, rising blocks are executed. Thus, the $\uparrow$ relation triggers the execution of the rising block defined in the body of the `fired` process.

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

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

FIGURE 2.26: The execution of the `fired` process during a rising edge phase. The `fired` process is a part of the `tl` design's behavior.

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

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

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

$\mathcal{H}$-VHDL abstract syntax

The implementation of the $\mathcal{H}$-VHDL abstract syntax is naturally done leveraging the `Inductive` construct of the Coq proof assistant. The result is strictly similar to the formal definition of the abstract syntax given in Section 2.3. The reader can refer to the `AbstractSyntax.v` under the `hvhdl` folder for the details of the implementation.

Elaborated design

Listing 2.7 presents the implementation of the elaborated design structure (cf. Definition 1). Two definitions are involved in the implementation of the elaborated design structure. The first one defines the `SemanticObject` inductive type. Each constructor of this type corresponds to a sub-environment of the elaborated design. For instance, the `Generic` constructor correspond to the couple $(type \times value)$ associated with a generic constant identifier in the *Gens* sub-environment of Definition 1. The `Process` constructor corresponds to the local variable environment associated with the process identifiers in the *Ps* sub-environment. A local variable environment is implemented by the `LEnv` type. The `LEnv` type is a map between identifiers and couples $(type \times value)$. Identifiers are implemented by the `ident` type, an alias of the `nat` type. The `type` and `value` types are the implementation of the semantic *type* and *value* presented in Table 2.2. The `ElDesign` type implements the elaborated design structure. It is an alias to the `IdMap SemanticObject` type. The `IdMap` is the type of maps from identifiers (i.e. belonging to the `ident` type) to instances of the type passed as a parameter. Here, the parameter is the `SemanticObject` type. Thus, an elaborated design is implemented as a map between identifiers and terms of the `SemanticObject` type. We leverage the `FMaps` module defined in the Coq standard library to implement the `IdMap` type. The `IdMap` type ensures that an identifier is only mapped once. Thus, the implementation of the elaborated design structure verifies that there are no intersection between the domains of sub-environments. For instance, a generic constant identifier can not be a input port identifier, and, as it is implemented, an identifier `id` can not be mapped to a `Generic` object and to an `Input` object in the same instance of `ElDesign`.

```
Inductive SemanticObject : Type :=
| Generic (t : type) (v : value)
| Input (t : type)
| Output (t : type)
| Declared (t : type)
| Process (lenv : LEnv)
| Component (Δc : IdMap SemanticObject).

Definition ElDesign := IdMap SemanticObject.
```

LISTING 2.7: The implementation of the elaborated design structure with the Coq proof assistant.

Design state

Listing 2.8 gives the implementation of the design state structure through the definition of the `DState` inductive type. The constructor of the `DState` type defines three fields: `sigstore`, implementing the signal store $\mathcal{S}$ of the design state, `compstore`, implementing the component store C, and `events`, implementing the set of events $\mathcal{E}$ of the design state. The `sigstore` field is a map from idenfifiers to values. The `compstore` field is a map from idenfifiers to design states, justifying the inductive definition of the `DState` type. The `events` field is an instance of the `IdSet` type. The `IdSet` is the type of sets of identifiers (i.e. sets of natural numbers). The `IdSet` type is defined leveraging the `MSets` module of the Coq standard library.

```
Inductive DState : Type := MkDState {
  sigstore :  IdMap value;
  compstore : IdMap DState;
  events    : IdSet;
5 }.
```

LISTING 2.8: The implementation of the design state structure with the Coq proof assistant.

2.8.2 Implementation of the elaboration phase

The design elaboration relation, as presented in Section 2.5.1, is implemented in Coq by the `edesign` relation. Listing 2.9 presents the definition of the `edesign` relation as an inductive type. As usual, a n-ary relation is implemented in Coq by a type defined with n parameters and projecting to the `Prop` type. The `edesign` relation as five parameters. The first parameter is the design store $\mathcal{D}$ of type `IdMap design`, i.e. a map from identifiers to $\mathcal{H}$-VHDL designs as defined by the abstract syntax. The second parameter is the dimensioning function $\mathcal{M}$ of type `IdMap value`, i.e. a map from identifiers to values. The third parameter is the design being elaborated, of type `design`. The fifth and sixth parameters are the elaborated design (of type `ElDesign` and the default design state (of type `DState`) resulting from the elaboration. In Listing 2.9, the `EDesign` constructor implements the DESIGNELAB rule presented in Section 2.5.1. From Line 7 to Line 10, the constructor defines the premises of Rule DESIGNELAB. The empty elaborated design structure, denoted $\Delta_{\emptyset}$, is implemented by the `EmptyElDesign` definition, and the empty design state structure, denoted by $\sigma_{\emptyset}$, is implemented by the `EmptyDState` definition. Line 13 implements the conclusion of Rule DESIGNELAB.

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

LISTING 2.9: The implementation of the design elaboration relation with the Coq proof assistant.

The `edesign` relation necesitates a mutually recursive definition with the `ebeh` relation. The mutually recursive definition is performed leveraging the `with` clause at the end of Listing 2.9. The `ebeh` relation needs the `edesign` relation to elaborate the component instances found in the behavior of a design. Listing 2.10 gives the details of the `with` clause defining the `ebeh` relation.

At Line 2, the `EBehPs` constructor implements the Rule PSELAB defining the elaboration of a process statement (cf. Section 2.5.6). Lines 6 and 7 implement the premises of the rule; the `evars` relation implements the elaboration of the local variable declaration list of the process; the `validss` relation implements the relation that type-checks the statement body of the process. Lines 10 to 14 implement the side conditions of the rule. The term `~NatMap.In` id_p Δ implements the side condition $id_p \notin \Delta$. The `NatMap.In id m` relation states that a given identifier `id` is a key of the `m` map. At Line 13, the `NatSet.In` id_s `sl` term states that id_s belongs to the identifier set `sl`. At Line 14, the term `MapsTo` id_s `(Input t)` Δ states that the identifier id_s is mapped to `Input t` in the elaborated design Δ, i.e. $Ins(\Delta)(id_s) = \mathtt{t}$. More generally, `MapsTo` is a ternary relation stating that a given key `k` of type `nat`, is mapped to a value `v` of a type `A`, in a given map `m`, i.e. `Mapsto k v m`. Line 17 implements the conclusion of Rule PSELAB. The `NatMap.add` function binds the process identifier id_p to the term `Process` Λ in the elaborated design Δ, i.e. $\Delta \cup (id_p, \Lambda)$.

At Line 19, the `EBehComp` constructor implements the Rule COMPELAB (cf. Section 2.5.6). This rule describes the elaboration of a component instantiation statement. Lines 24 to 27 implement the premises of the rule. Line 25 appeals to the `edesign` relation to elaborate the `cdesign` design associated with the component instance id_c; thence, the mutually recursive definition with the `ebeh` relation. As it is stated at Line 32, the `cdesign` design is associated to identifier id_e, i.e. the entity identifier of component instance id_c, in the design store $\mathcal{D}$. Lines 30 to 33 implement the side conditions of the rule. Line 30 checks that the identifier id_c is not already bound to a semantic object in the elaborated design Δ. Line 31 checks that the identifier id_c is not already bound in the component store of σ. Line 33 checks that all identifiers defined in the domain of map $\mathcal{M}$, i.e. the dimensioning function, are bound to generic constants in the elaborated design Δ_c (i.e. $\mathcal{M} \subseteq Gens(\Delta_c)$). Lines 36 to 38 implement the conclusion of Rule COMPELAB. At Line 39, the `cstore_add` function binds id_c to design state σ_c in the component store of state σ and returns the resulting state.

At Line 41, the `EBehNull` constructor implements Rule CSNULLELAB. At Line 43, the `EBehPar` constructor implements Rule CSPARELAB.

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

```

| EBehComp :
    forall Δ σ id_c id_e gmap ipmap opmap
           𝓜 Δ_c σ_c formals actuals cdesign,

      (* Premises *)
      emapg (NatMap.empty value) gmap 𝓜 →
      edesign 𝓓 𝓜 cdesign Δ_c σ_c →
      validipm Δ Δ_c σ ipmap formals →
      validopm Δ Δ_c opmap formals actuals →

      (* Side conditions *)
      ~NatMap.In id_c Δ →
      ~NatMap.In id_c (compstore σ) →
      MapsTo id_e cdesign 𝓓 →
      (forall g, NatMap.In g 𝓜 → exists t v, MapsTo g (Generic t v) Δ_c) →

      (* Conclusion *)
      ebeh 𝓓 Δ σ (cs_comp id_c id_e gmap ipmap opmap)
                  (NatMap.add id_c (Component Δ_c) Δ)
                  (cstore_add id_c σ_c σ)

| EBehNull: forall Δ σ, ebeh 𝓓 Δ σ cs_null Δ σ

| EBehPar:
    forall Δ Δ' Δ'' σ σ' σ'' cstmt cstmt',
      ebeh 𝓓 Δ σ cstmt Δ' σ' →
      ebeh 𝓓 Δ' σ' cstmt' Δ'' σ'' →
      ebeh 𝓓 Δ σ (cs_par cstmt cstmt') Δ'' σ''.
```

LISTING 2.10: The implementation of the `ebeh` behavior elaboration relation with the Coq proof assistant.

2.8.3 Implementation of the simulation algorithm

The full simulation relation (cf. Section 2.6.1) formalizes the $\mathcal{H}$-VHDL simulation algorithm. The Coq implementation of the full simulation relation, presented in Listing 2.11, is a strict translation of Rule FULLSIM. At Lines 14 and 15, the term `(behavior d)` represents the concurrent statements defining the behavior of the $\mathcal{H}$-VHDL design `d` (i.e. d.cs in the formal rule). Line 13 corresponds to the elaboration phase, Line 14 to the initialization phase, and Line 15 to the main simulation loop.

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

```
        (Δ : ElDesign)
        (d : design) : list DState → Prop :=

| FullSim :
    forall σ_e σ_0 θ,

      (* * Premises * *)
      edesign 𝒟 ℳ d Δ σ_e →
      init 𝒟 Δ σ_e (behavior d) σ_0 →
      simloop 𝒟 E_p Δ σ_0 (behavior d) τ θ →

      (* * Conclusion * *)
      fullsim 𝒟 ℳ E_p τ Δ d (σ_0 :: θ).
```

LISTING 2.11: The implementation of the full simulation relation with the Coq proof assistant.

The `simloop` relation appeals to the `simcycle` that implements the simulation cycle relation defined in Section 2.6.3. Listing 2.12 presents the implementation of the `simcycle` relation. The `simcycle` relation is a strict transcription of the SIMCYC rule. At Line 13, the `vrising` relation implements the $\uparrow$ relation, i.e. the rising edge phase of the cycle. At Line 15, the `vfalling` relation implements the $\downarrow$ relation, i.e. the falling edge phase of the cycle. At Lines 14 and 16, the `stabilize` relation implements the $\rightsquigarrow$ relation, i.e. the stabilization phases of the simulation cycle. At Lines 18 and 19, the `IsInjectedDState` relation implements the $\texttt{Inject}_\downarrow$ and $\texttt{Inject}_\uparrow$ relations. Line 18 states that the σ_i state is the result of the injection of the map $(E_p\ \tau\uparrow)$ in the signal store of state σ.

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

LISTING 2.12: The implementation of the simulation cycle relation with the Coq proof assistant.

2.9 Conclusion

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

Ever since the mechanization of the proof of behavior preservation has begun, the semantics of $\mathcal{H}$-VHDL has been evolving. Section 2.3, 2.4, 2.5 and 2.6 present the most recent version of the semantics. However, it will probably be evolving again. With regards to our proof task, we realized that an operational semantics close to the simulation algorithm carried a lot of elements that were of no use. These elements could sometimes complexify the proof. For instance, in the VHDL simulation algorithm, the body of a process is executed during the stabilization phase only if one signal of its sensitivity list is part of the current state's event set. However, it is through the execution of the body of a process with the rules of the $\mathcal{H}$-VHDL semantics that we can determine the *combinational* equation associated with the value of a signal. In the proceeding of the proof of semantic preservation, we must often describe the value of an output signal with regards to the value of its input, or *source*, signals (cf. Section **??**). Due to the event-based system of resuming a process activity, a combinational process could sometimes never be executed during a stabilization phase. Then, we are not able to determine the value of signals. We had to carry extra hypotheses in the definition of our lemmas to deal with this problem. Finally, our current semantics always executes the body combinational processes during a stabilization phase, and this greatly simplifies the proof. By doing this kind of simplification, we realized that we were heading toward a semantics that was closer to the "synthesis" semantics we talked about at the beginning of the chapter. This semantics tends to get closer to the combinational logic and the synchronous logic rules. These rules that a hardware system designer has in mind when devising a model with a hardware description language.

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

Bibliography

[1] Peter J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann, Oct. 7, 2010. 933 pp. ISBN: 978-0-08-056885-0. Google Books: XbZr8DurZYEC.

[2] Dominique Borrione and Ashraf Salem. "Denotational Semantics of a Synchronous VHDL Subset". In: *Formal Methods in System Design* 7.1-2 (Aug. 1995), pp. 53–71. ISSN: 0925-9856, 1572-8102. DOI: 10.1007/BF01383873. URL: http://link.springer.com/10.1007/BF01383873 (visited on 09/18/2019).

[3] Peter T. Breuer, Luis Sánchez Fernández, and Carlos Delgado Kloos. "A Functional Semantics for Unit-Delay VHDL". In: *Formal Semantics for VHDL*. Ed. by Carlos Delgado Kloos and Peter T. Breuer. Red. by Jonathan Allen. Vol. 307. Boston, MA: Springer US, 1995, pp. 43–70. ISBN: 978-1-4613-5941-8 978-1-4615-2237-9. DOI: 10.1007/978-1-4615-2237-9_3. URL: http://link.springer.com/10.1007/978-1-4615-2237-9_3 (visited on 09/09/2019).

[4] Peter T. Breuer, Luis Sánchez Fernández, and Carlos Delgado Kloos. "A Simple Denotational Semantics, Proof Theory and a Validation Condition Generator for Unit-Delay VHDL". In: *Formal Methods in System Design* 7.1 (Aug. 1, 1995), pp. 27–51. ISSN: 1572-8102. DOI: 10.1007/BF01383872. URL: https://doi.org/10.1007/BF01383872 (visited on 02/20/2020).

[5] Egon Börger, Uwe Glässer, and Wolfgang Muller. "A Formal Definition of an Abstract VHDL'93 Simulator by EA-Machines". In: *Formal Semantics for VHDL*. Ed. by Carlos Delgado Kloos and Peter T. Breuer. Red. by Jonathan Allen. Vol. 307. Boston, MA: Springer US, 1995, pp. 107–139. ISBN: 978-1-4613-5941-8 978-1-4615-2237-9. DOI: 10.1007/978-1-4615-2237-9_5. URL: http://link.springer.com/10.1007/978-1-4615-2237-9_5 (visited on 09/12/2019).

[6] Alain Colmerauer. "An Introduction to Prolog III". In: *Computational Logic* (1990), pp. 37–79. DOI: 10.1007/978-3-642-76274-1_2. URL: https://link.springer.com/chapter/10.1007/978-3-642-76274-1_2 (visited on 03/31/2020).

[7] Frank Dederichs, Claus Dendorfer, and Rainer Weber. "Focus: A Formal Design Method for Distributed Systems". In: *Parallel Computer Architectures* (1993), pp. 190–202. DOI: 10.1007/978-3-662-21577-7_14. URL: https://link.springer.com/chapter/10.1007/978-3-662-21577-7_14 (visited on 03/31/2020).

[8] David Déharbe and Dominique Borrione. "Semantics of a Verification-Oriented Subset of VHDL". In: *Correct Hardware Design and Verification Methods*. Advanced Research Working Conference on Correct Hardware Design and Verification Methods. Springer, Berlin,

Heidelberg, Oct. 2, 1995, pp. 293–310. DOI: 10.1007/3-540-60385-9_18. URL: https://link.springer.com/chapter/10.1007/3-540-60385-9_18 (visited on 04/01/2020).

[9] Gert Döhmen and Ronald Herrmann. "A Deterministic Finite-State Model for VHDL". In: *Formal Semantics for VHDL*. Ed. by Carlos Delgado Kloos and Peter T. Breuer. The Kluwer International Series in Engineering and Computer Science. Boston, MA: Springer US, 1995, pp. 170–204. ISBN: 978-1-4615-2237-9. DOI: 10.1007/978-1-4615-2237-9_7. URL: https://doi.org/10.1007/978-1-4615-2237-9_7 (visited on 03/02/2020).

[10] Max Fuchs and Michael Mendler. "A Functional Semantics for Delta-Delay VHDL Based on Focus". In: *Formal Semantics for VHDL*. Ed. by Carlos Delgado Kloos and Peter T. Breuer. Red. by Jonathan Allen. Vol. 307. Boston, MA: Springer US, 1995, pp. 9–42. ISBN: 978-1-4613-5941-8 978-1-4615-2237-9. DOI: 10.1007/978-1-4615-2237-9_2. URL: http://link.springer.com/10.1007/978-1-4615-2237-9_2 (visited on 09/03/2019).

[11] K. G. W. Goossens. "Reasoning about VHDL Using Operational and Observational Semantics". In: *Correct Hardware Design and Verification Methods*. Advanced Research Working Conference on Correct Hardware Design and Verification Methods. Springer, Berlin, Heidelberg, Oct. 2, 1995, pp. 311–327. DOI: 10.1007/3-540-60385-9_19. URL: https://link.springer.com/chapter/10.1007/3-540-60385-9_19 (visited on 04/01/2020).

[12] Graham Hutton. "Introduction to HOL: A Theorem Proving Environment for Higher Order Logic by Mike Gordon and Tom Melham (Eds.), Cambridge University Press, 1993, ISBN 0-521-44189-7". In: *Journal of Functional Programming* 4.4 (Oct. 1994), pp. 557–559. ISSN: 1469-7653, 0956-7968. DOI: 10.1017/S0956796800001180. URL: https://www.cambridge.org/core/journals/journal-of-functional-programming/article/introduction-to-hol-a-theorem-proving-environment-for-higher-order-logic-by-gordon-mike-and-melham-tom-eds-cambridge-university-press-1993-isbn-0521441897/682CAD7058D7014549AE3F9580D0220B (visited on 04/22/2020).

[13] IEEE Computer Society et al. *IEEE Standard VHDL Language Reference Manual*. New York, N.Y.: Institute of Electrical and Electronics Engineers, 2000. ISBN: 978-0-7381-1948-9 978-0-7381-1949-6. URL: http://ieeexplore.ieee.org/lpdocs/epic03/standards.htm (visited on 09/16/2019).

[14] *IEEE/IEC 62142-2005 - IEC/IEEE International Standard - Verilog(R) Register Transfer Level Synthesis*. URL: https://standards.ieee.org/standard/62142-2005.html (visited on 06/30/2021).

[15] Mark P. Jones. *The Implementation of the Gofer Functional Programming System*. Yale University, Department of Computer Science, May 1994, p. 52.

[16] 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).

[17] Joan Moschovakis. "Intuitionistic Logic". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Winter 2018. Metaphysics Research Lab, Stanford University, 2018. URL: https://plato.stanford.edu/archives/win2018/entries/logic-intuitionistic/.

[18] Serafín Olcoz. "A Formal Model of VHDL Using Coloured Petri Nets". In: *Formal Semantics for VHDL*. Ed. by Carlos Delgado Kloos and Peter T. Breuer. The Kluwer International Series in Engineering and Computer Science. Boston, MA: Springer US, 1995, pp. 140–169. ISBN: 978-1-4615-2237-9. DOI: 10.1007/978-1-4615-2237-9_6. URL: https://doi.org/10.1007/978-1-4615-2237-9_6 (visited on 03/02/2020).

[19] S. Owre et al. "A Tutorial on Using PVS for Hardware Verification". In: *Theorem Provers in Circuit Design*. International Conference on Theorem Provers in Circuit Design. Springer, Berlin, Heidelberg, Sept. 26, 1994, pp. 258–279. DOI: 10.1007/3-540-59047-1_53. URL: https://link.springer.com/chapter/10.1007/3-540-59047-1_53 (visited on 03/31/2020).

[20] S.L. Pandey, K. Umamageswaran, and P.A. Wilsey. "VHDL Semantics and Validating Transformations". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 18.7 (July 1999), pp. 936–955. ISSN: 1937-4151. DOI: 10.1109/43.771177.

[21] Volnei A. Pedroni. *Circuit Design with VHDL, Third Edition*. MIT Press, Apr. 14, 2020. 609 pp. ISBN: 978-0-262-35392-2. Google Books: fVzbDwAAQBAJ.

[22] Ralf Reetz and Thomas Kropf. "A Flow Graph Semantics of VHDL: A Basis for Hardware Verification with VHDL". In: *Formal Semantics for VHDL*. Ed. by Carlos Delgado Kloos and Peter T. Breuer. The Kluwer International Series in Engineering and Computer Science. Boston, MA: Springer US, 1995, pp. 205–238. ISBN: 978-1-4615-2237-9. DOI: 10.1007/978-1-4615-2237-9_8. URL: https://doi.org/10.1007/978-1-4615-2237-9_8 (visited on 03/02/2020).

[23] Krishnaprasad Thirunarayan and Robert L. Ewing. "Structural Operational Semantics for a Portable Subset of Behavioral VHDL-93". In: *Formal Methods in System Design* 18.1 (Jan. 1, 2001), pp. 69–88. ISSN: 1572-8102. DOI: 10.1023/A:1008786720393. URL: https://doi.org/10.1023/A:1008786720393 (visited on 03/02/2020).

[24] John P. Van Tassel. "An Operational Semantics for a Subset of VHDL". In: *Formal Semantics for VHDL*. Ed. by Carlos Delgado Kloos and Peter T. Breuer. Red. by Jonathan Allen. Vol. 307. Boston, MA: Springer US, 1995, pp. 71–106. ISBN: 978-1-4613-5941-8 978-1-4615-2237-9. DOI: 10.1007/978-1-4615-2237-9_4. URL: http://link.springer.com/10.1007/978-1-4615-2237-9_4 (visited on 09/12/2019).