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

En Informatique

École doctorale : Information, Structures, Systèmes

Unité de recherche LIRMM

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

Présenté par Vincent IAMPIETRO

Le Date de la soutenance

Sous la direction de David Delahaye
et David Andreu

Devant le jury composé de

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

Acknowledgements

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

Contents

List of Figures

List of Tables

List of Abbreviations

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

For/Dedicated to/To my...

Chapter 1

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

- Make a remark on the notation of design states, e.g. $\mathcal{E}''$ refers to the set of events of design state σ'' when there is no ambiguity.
- Two points of view to consider the semantics of VHDL: simulation or synthesis. Simulation described in the LRM; synthesis, problem is that there are no standard, unlike Verilog (cite Verilog standard synthesis semantics).

1.1 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 [20].

1.1.1 Main concepts

The VHDL acronym stands for Very high speed integrated circuit Hardware Description Language. As its name indicates, the main purpose of the VHDL language is to describe hardware circuits. There are two approaches to the description of circuits. The first aims at the simulation of the described circuits, and the second aims at the synthesis of described circuits on physical supports. Thus, the constructs of the VHDL language must be interpreted depending on the purpose of the designer. For instance, the language gives the possibility to describe the connection of wires inside a circuit. A wire is represented by the concept of *signal*. In the context of circuit simulation, a *signal* can be compared to a variable; it has a given type and holds a value that fluctuates in the course of the simulation. In the context of synthesis, a signal really

describes a physical wire and must be considered as so. From these two approaches to circuit description arise two ways of considering the semantics of the language (see Section 1.2).

In VHDL, a top-level program is called a *design*. A design describes a hardware circuit. As explained in Chapter **??**, the hilecop transformation generates a VHDL design implementing the input SITPN model. To do so, the transformation generates and connects the component instances of two previously defined VHDL designs: the *place* design that implements the concept of a SITPN place, and the *transition* design that implements a SITPN transition. These designs were defined by the INRIA CAMIN team at the creation of the HILECOP methodology. In the following sections, we will be using excerpts of the definition of the place and transition designs to illustrate the content of VHDL programs and the rules of the VHDL language semantics. The reader will find the source code of the place and transition designs in concrete and abstract syntax in Appendices A and B.

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 1.1 is an excerpt of the transition design's entity that defines the generic constants, the input and output port interfaces of the design. Figure 1.1 is a visual representation of the interfaces of the transition design.

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

LISTING 1.1: The entity part of the transition design in concrete VHDL syntax.

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 1.1, 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.

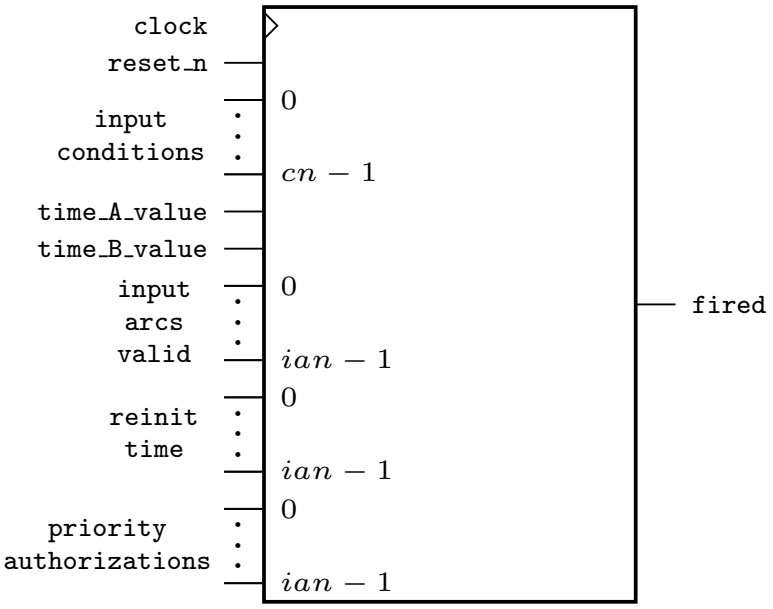

FIGURE 1.1: A representation of the transition design entity. On the left side, the input port interface of the transition design; *cn* stands for *conditions_number* and *ian* stands for *input_arcs_number*, i.e. two of the generic constants declared in the generic clause of the transition design entity; 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 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 is to specify one or multiple processes. Processes are concurrent statements that describes the wiring or the operations performed on the signals of a given design. A process declares a sensitivy list that corresponds to the signals read in the process statement body; also, it possibly declares internal variables. The sensitivity list is only useful for the purpose of simulation. It permits to resume the execution of a process when the value of one of the signals of its sensitivity list changes. Listing 1.2 gives an excerpt of the transition design architecture containing the declarative part of the architecture (i.e. 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.

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

LISTING 1.2: An excerpt of the architecture part of the transition design in concrete VHDL syntax.

In Listing 1.2, from Line 2 to Line 8, the architecture declares the internal signals of the transition design. Then, Line 11 begins the declaration of the `condition_evaluation` process. The sensitivity list of the `condition_evaluation` process holds one signal, the `input_conditions` input port, and declares a local variable `v_internal_condition`.

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 1.2, the signal assignment statement, represented with the ⇐ operator, assigns the value of the internal variable `v_internal_condition` to signal `s_condition_evaluation`; the `v_internal_variable` that itself holds the Boolean product between the members 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 1.2 gives a synthesis-oriented view of the processes described in Listing 1.2.

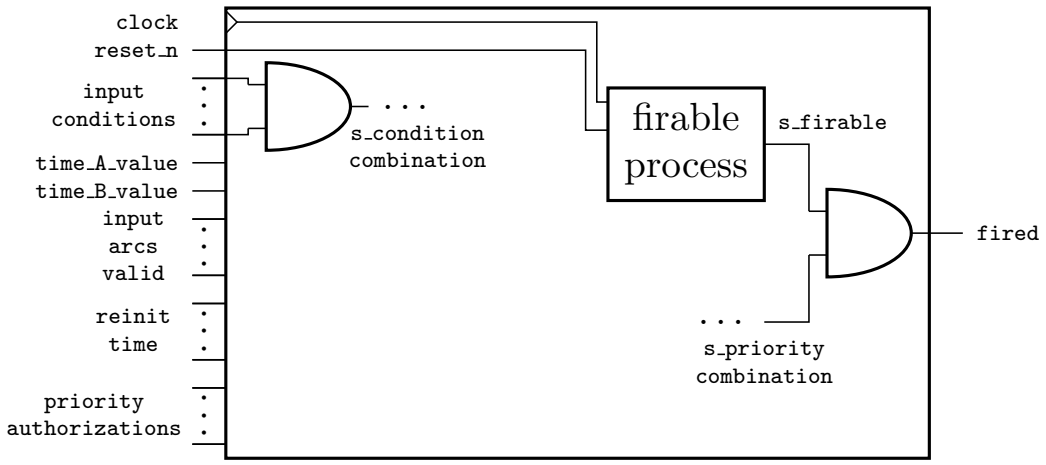

FIGURE 1.2: A representation of a part of the transition design architecture comprising three processes. On the left side, the `condition_evaluation` process connecting the `input_conditions` input port to the `s_condition_combination` internal signal; the `firable` process in the middle; on the right side, the `fired_evaluation` process connecting the `s_firable` and the `s_priority_combination` signals to the `fired` output port.

In Figure 1.2, 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 `firable` process is a *synchronous* process. It responds to the event of the `clock` signal. 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, i.e. processes that are executed independently of the occurrence of a clock signal. However, in the point of view of synthesis, processes responding to a `clock` signal follow the rules of the synchronous 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 propagation time 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 here because they are the witnesses of the two time paradigms that inhabit the simulation algorithm describe the semantics of the VHDL language: δ time and real time.

Behavior specification with design instances

The second way to specify the behavior of a design is to use other designs, or rather instances of other designs, as components of the circuits. 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 1.3 shows an example of instantiation of the HILECOP's transition design. This instance is involved in the definition of the behavior of an embedding design called `toplevel`.

```
1  architecture toplevel_architecture of toplevel is
2  begin
3    ...
4    id_t : entity transition
5    generic map (
6      transition_type ⇒ NOT_TEMPORAL,
7      input_arcs_number ⇒ 1,
8      conditions_number ⇒ 1,
9      maximal_time_counter ⇒ 1
10   )
11   port map (
12     clock ⇒ clock,
13     reset_n ⇒ reset_n,
```

```
14      time_A_value ⇒ 0,
15      time_B_value ⇒ 0,
16      input_conditions(0) ⇒ id_0,
17      input_arcs_valid(0) ⇒ id_1,
18      priority_authorizations(0) ⇒ '1',
19      reinit_time(0) ⇒ id_2,
20      fired ⇒ id_3
21    );
22    ...
23  end toplevel_architecture;
```

LISTING 1.3: An example of design instantiation statement in the architecture of the `toplevel` design. Here, the design being instantiated is the transition design.

In Listing 1.3, the transition component instance 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`. 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 as output port association describes a direct wiring. In the port map described in Listing 1.3, the association `fired` $\Rightarrow$ id_3 expresses that the `fired` output port is connected to the signal id_3, where signal id_3 is defined in the embedding design. Figure 1.3 illustrates the transition design instance id_t and the wiring of its input and output port interfaces inside the toplevel design.

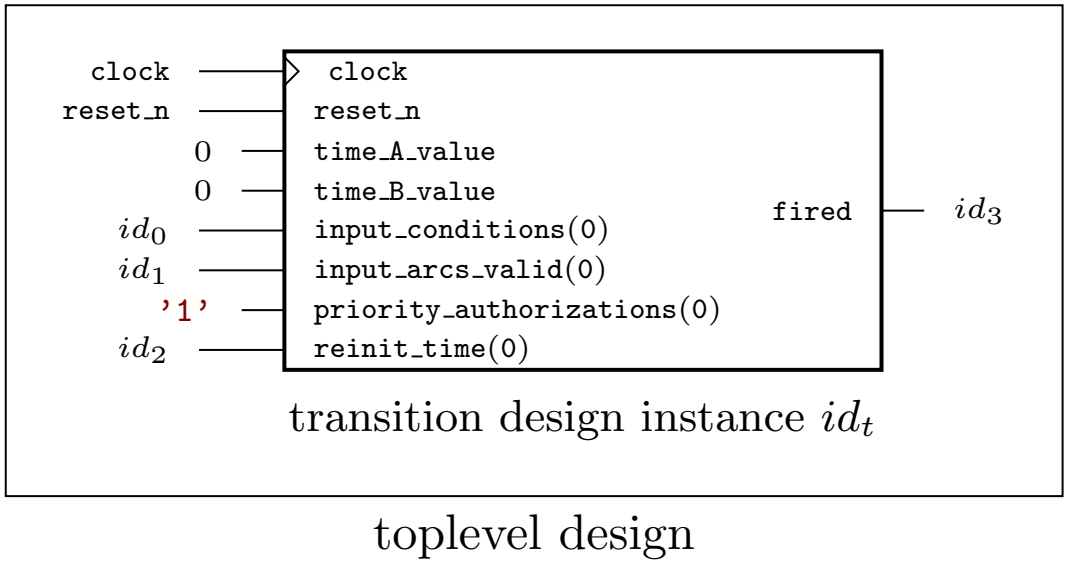

FIGURE 1.3: Visual representation of a design instantiation statement. Here, the figure represents the transition design instance described in Listing 1.3.

1.1.2 Informal semantics of the VHDL language

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. 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 that permits to transform a design into a simulation-ready execution model. The elaboration phase has several goals. First, it builds the simulation environment and a starting design simulation state. The simulation environment is built based on the declarative parts of the top-level design; it maps the signals to their types. In the design simulation state, each signal is associated with a current value 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 necessity of drivers is explained by the presence of unit-delay signal assignments. A unit-delay signal assignment specify a time clause indicating when a giving assignment must be performed, e.g. $a \Leftarrow b$ `in 3ms` (signal `a` takes the value of signal `b` in 3 milliseconds). Thus, when a unit-delay signal assignment is executed in the course of a simulation, its effect is to change the driver of the target signal by posting a new transaction. For instance, let T_c by the current simulation time, the execution of statement $a \Leftarrow$ `true in 2ns` sets a new transaction in the driver of signal a. The new

transaction associates the value `true` to the time point $T_c + 2\texttt{ns}$. Note that without unit-delay signal assignments, drivers are not needed as all assignments take their effects at the current simulation time. Second, 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, 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 resumes the execution of processes. Figure 1.4, which is an excerpt from [5], describes the structure of the VHDL simulation algorithm.

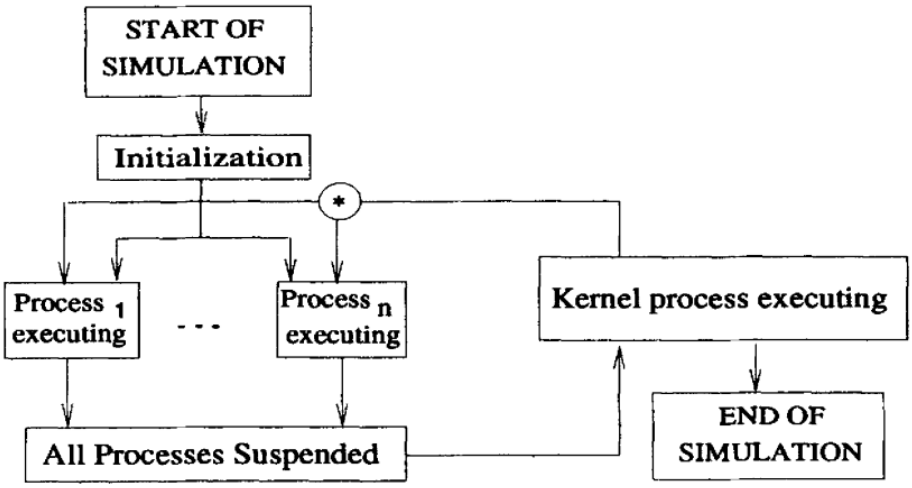

FIGURE 1.4: The VHDL simulation loop. Excerpt from [5].

The simulation starts with the 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 1.5 shows the activity diagram associated with the kernel process.

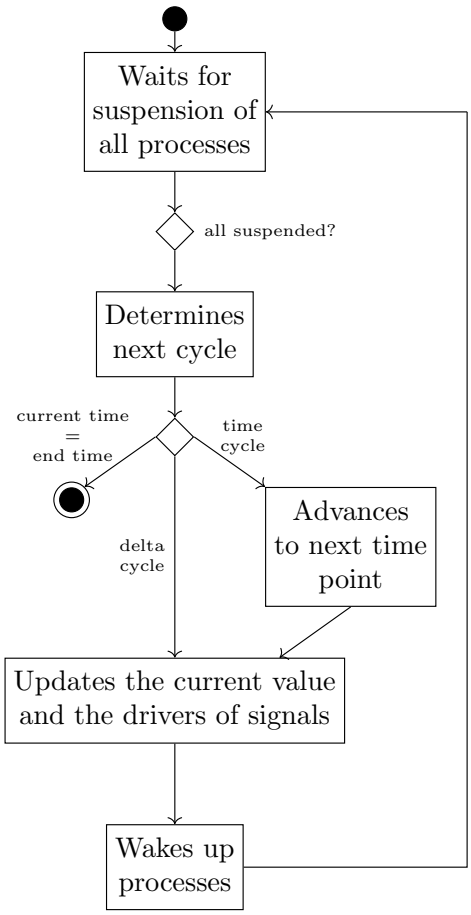

FIGURE 1.5: The activity diagram of the kernel process. Square boxes represent activities, diamond nodes are decision nodes. The black circle at the top represents the starting point of the activities; the other black circle in the middle of the diagram represents the end of all activities.

As shown in Figure 1.5, the kernel process will then determine the kind of simulation cycle that will be performed next. There are two kinds of cycles: delta cycle or time cycle. 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.

1.2 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 research 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.

1.2.1 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 transport-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 wait statements or delayed signal assignments) as input.

Concerning component instantiation statement, 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 instance 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. 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 constant clauses of the HILECOP place and transition designs (see Section Presentation of the VHDL language) help limiting the type ranges. 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 (even though it seems a trivial formalization task compared to the core of the VHDL semantics).

The need for synchronous execution

The second property of HILECOP's generated VHDL programs is their synchronous execution. Indeed, 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 of a semantics that deal with synchronism, and that explicitly integrates the synchronization with a clock signal into the expression of the simulation cycle. Thus, the two dimensions of time, respectively delta time and "real" time [21], that are part of the VHDL language, must be handled by our semantics. Delta time pertains to delta-cycles, which are triggered by the execution of delta-delay signal assignments, that is, the only kind of signal assignments found in HILECOP VHDL programs. Real time pertains to time-steps. When all signal values are stable (when there are no more delta-cycles), the simulation advances the current time value to the next time point where a relevant event happens. A relevant event could be the execution of the next pending signal assignment that was associated with a delay, or the end of a `wait for x` statement (where `x` specifies a time delay). In our case, neither delayed signal assignments nor wait statements are part the covered VHDL language subset. The only two relevant events to which the simulation advances during a time-step are the rising edge and the falling edge of the clock signal.

Other considerations

Considering the kind of proof that needs to be established, we would rather consider an operational semantics for VHDL than a denotational one. 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 promote the semantics that explicitly formalize a synchronized execution of a VHDL design.

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

- *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 promote 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 promote the first kind of semantics.

1.2.2 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 that is 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...Such tasks that are more fit for a denotational semantics.

In [10], the authors give a denotational semantics to the VHDL language, this leveraging the Focus [7] 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. Resolution functions are used in case of multiply-driven signals (i.e, signals that receive a value from multiple processes). The authors only consider 0-delay signal assignment 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 statement is not treated by the semantics.

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 (no 0-delay), and therefore does not cover the VHDL subset that we are interested in. The hierarchical structure of designs is also not preserved, and, although expressible, the semantics does explicitly express a synchronous simulation cycle.

The denotational semantics expressed in [19] 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 [18]. 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. The semantics works on fully-elaborated designs, therefore, it does not deal with component instance 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.

Joining the common opinion, 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 is interested in expressing 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. The only critic that can be made to this semantics regarding our needs is that it is expressed in denotational style.

Operational semantics

Multiple works formalize an operational semantics for VHDL. Naturally, these works are interested in the formal description of the VHDL simulator, more or less closely to the description of the LRM.

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. 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 [23], 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 the subset of the VHDL language that we are interested in, even if, it also covers some constructs pertaining to unit delays that are irrelevant to us (like wait and unit-delay signal assignment statements). 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 arbitrary wait statements, and both unit and delta-delay signal assignments. There is a interested 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 process have reached a wait statement that stalled the execution of their inner body. The author is interested in the comparison of the behavior, and therefore, the equivalence between to VHDL programs. 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 oberservational 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 to capture 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 (i.e, A actions, execution of a process body that does not end in a wait statement) performed by a VHDL program. Indeed, a natural operational semantics in the style of Van Tassel's [23] is sufficient in our case. In [22], 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 or postponed processes, therefore, this work is not really relevant to us.

Translational semantics

Another kind of semantics, called "translational", is interested in establishing a formal semantics for VHDL 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 [21], 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 blocked in a wait configuration, 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 (for instance, after a wait statement), therefore achieving determinism in the control flow of the overall automaton.

In [17], 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. 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 smeantics 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 or *for* clause in wait statements). The author claims that a VHDL design is implemented by an abstract machine if they have the same observational behavior, i.e, for the same value in their inputs they yield the same values in their outputs. Each process statement part is transformed into a decision diagram (control flow graph); then, the decision diagram encodes the transition functions over states and outputs in the abstract machine implementing the corresponding process. Process statements are composed in relation to some composition operator. 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 few papers 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 1.1 summarizes the analysis of the VHDL semantics encountered during our literature review. Table 1.1 compares the different VHDL semantics in relation to the qualifying criterions (see Section 1.2.1).

		Fuchs and Mendler [10]	Breuer et al. [4]	Pandey et al. [19]	Borrione and Salem [2]	Breuer et al. [3]	Börger et al. [5]	Van Tassel [23]	Goossens [11]	Reetz and Kropf [21]	Döhmen and Herrmann [9]	Olcoz [17]	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 [18]	?	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 1.1: 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 1.1, 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. Cell are labelled in light grey when their content is neutral in relation to the semantics adoption. Now comparing

the entries of Table 1.1 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 the observations, we have decided to formalize our own VHDL semantics inspired from the semantics of Borrione and Salem's [2] and Van Tassel's [23]. 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.

1.3 Abstract syntax for $\mathcal{H}$-VHDL

In this section, we describe the abstract syntax of $\mathcal{H}$-VHDL, the subset of VHDL covering all the constructs present in the generated $\mathcal{H}$-VHDL programs. Terminals of the language will be written in `typewriter` font, or will be 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.

1.3.1 Design declaration

As in [23], we define the *design* construct in the $\mathcal{H}$-VHDL's abstract syntax which has no equivalent in the concrete syntax of VHDL.

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

1.3.2 Generic constant, port and internal signal declaration.

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

1.3.3 Concurrent statements.

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

Process statement.

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

Component instantiation statement.

cistmt	::=	`comp` (id_c, id_e, gmap, ipmap, opmap)
gmap	::=	$assoc_g^*$
ipmap	::=	$assoc_{ip}^*$
opmap	::=	$assoc_{op}^*$
$assoc_g$	::=	(id,e)
$assoc_{ip}$	::=	(name,e)
$assoc_{op}$	::=	(id,(name | `open`)) | (id(e),name)

1.3.4 Sequential statement.

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

1.3.5 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)

Under the expression entry, the natural non-terminal represents the set of natural numbers ($\mathbb{N}$). The id non-terminal represents the set of identifiers.

1.4 Preliminaries to $\mathcal{H}$-VHDL semantics

1.4.1 Semantics Domains

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

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

TABLE 1.2: The *type* and *value* semantical types.

In Table 1.2, the *type* type is in any way similar to the τ entry of the abstract syntax, however, all constraint bounds in the `nat` and `array` types 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 (that is index 0).

Notation 1 (Partial functions)**.** *Here, we present our notations pertaining to partial functions:*

- *The $\nrightarrow$ arrow denotes a partial function.*
- *The $\rightarrow$ denotes an application (i.e, a total function).*
- *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.*

Now, let us define the structure of an elaborated design which is a structure bound to a given $\mathcal{H}$-VHDL design and to a design store, i.e a global environment mapping identifiers to $\mathcal{H}$-VHDL designs. Only the designs referenced into the global design store can be instantiated as subcomponents of a given design. The elaborated design structure is used in the expression of the elaboration relation presented in Section Elaboration rules, as well as in the expression of the simulation rules. Let $ElDesign(d, \mathcal{D})$ be the set of the elaborated designs for a given $\mathcal{H}$-VHDL design d and a design store $\mathcal{D}$. An elaborated design is a composite environment built out of multiple sub-environments. Each sub-environment is a table, represented as a partial 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)**.** *For a given $\mathcal{H}$-VHDL design $d \in design$ s.t. $d =$ `design` id_{ent} id_{arch} gens ports sigs behavior and a given design store $\mathcal{D} \in entity\text{-}id \nrightarrow design$, an elaborated design $\Delta \in ElDesign(d, \mathcal{D})$ is a record $<Gens, Ins, Outs, Sigs, Ps, Comps>$ where:*

– *$Gens \in generic\text{-}id \nrightarrow (type \times value)$ where $generic\text{-}id = \{id \mid (id, \tau, e) \in gens\}$, is the partial function yielding the type and the value of generic constants.*

– $Ins \in input\text{-}id \nrightarrow type$ *where* $input\text{-}id = \{id \mid (\texttt{in}, id, \tau) \in ports\}$, *is the partial function yielding the type of input ports.*

– $Outs \in output\text{-}id \nrightarrow type$ *where* $output\text{-}id = \{id \mid (\texttt{out}, id, \tau) \in ports\}$, *the partial function yielding the type of output ports.*

– $Sigs \in declared\text{-}signal\text{-}id \nrightarrow type$ *where* $declared\text{-}signal\text{-}id = \{id \mid (id, \tau) \in sigs\}$, *the partial function yelding the type of declared signals.*

– $Ps \in process\text{-}id \nrightarrow (variable\text{-}id(id_p) \nrightarrow (type \times value))$ *where* $process\text{-}id = \{id_p \mid \texttt{process}(id_p, sl, vars, ss) \in behavior\}$, *the partial function associating processes to their local environment. Local environments are functions mapping local variable identifiers to their corresponding type and value. Therefore, each set of local variable identifiers* $variable\text{-}id(id_p)$ *depends on the process identifier (represented by* id_p*) passed as the first argument of the Ps function.*

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

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

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

Definition 2 (Design state)**.** *A design state* $\sigma \in \Sigma(\Delta)$*, for a given design* $d \in design$*, a given design store* $\mathcal{D}$ *and an elaborated design* $\Delta \in ElDesign(d, \mathcal{D})$*, is a record* $<\mathcal{S}, \mathcal{C}, \mathcal{E}>$ *where:*

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

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

– $\mathcal{E} \subseteq signal\text{-}id \cup component\text{-}id$*, the set of signal and component instance ids that generated an event at the current simulation cycle.*

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. $id \in_{\mathcal{E}} \sigma$ states that an identifier belongs to the event set $\mathcal{E}$, whereas $id \in \sigma$ states that an identifier is defined in either the signal store $\mathcal{S}$ or the component store $\mathcal{C}$ fields. $\sigma \cup_{\mathcal{E}} \{id\}$ denotes a new design state, in all points similar to σ but enriched with the identifier id in its events set.

Notation 2 (No events design state)**.** *For a given $\mathcal{H}$-VHDL design d, a design store $\mathcal{D}$, and an elaborated design $\Delta \in ElDesign(d,\mathcal{D})$, the function $NoEv \in \Sigma(\Delta) \to \Sigma(\Delta)$ returns a design state similar to the one passed in parameter only with an empty set of events. I.e, for all design state $\sigma \in \Sigma(\Delta)$ s.t. $\sigma = <\mathcal{S},\mathcal{C},\mathcal{E}>$, $NoEv(\sigma) = <\mathcal{S},\mathcal{C},\emptyset>$.*

1.5 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. The elaboration relation also covers type-checking operations for the declarative and behavioral parts of design d.

1.5.1 Design elaboration.

Definition 3 (Design store)**.** *$\mathcal{D} \in$ entity-id $\nrightarrow$* design *is the partial function mapping entity ids 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 $\mathcal{D}_{\mathcal{H}}$ holds the description of the `place` and `transition` designs which full definition in abstract syntax are given in Appendices A and B.*

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

DesignElab

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

where $\Delta_{\emptyset}$ 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.

1.5.2 Generic clause elaboration.

Premises

- $etype_g$ transforms a subtype indication, specifically attached to a generic constant declaration, into a *type* instance and checks its well-formedness (see 1.5.5).
- The e relation evaluates expression e to value v (see 1.6.9).
- SE_l states that an expression is *locally* static (see 1.5.9).
- $v \in_c T$ 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 1.5.8).

GENELABDIMEN

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

GENELABDEFAULT

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

GENELABCOMP

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

1.5.3 Port clause elaboration.

Premises

The $defaultv$ relation yields the implicit *default* value for a given type.

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

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

1.5.4 Architecture declarative part elaboration.

SigElab

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

SigElabComp

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

1.5.5 Type indication elaboration.

The *etype* relation checks the well-formedness of a type indication τ, and transforms it into a semantical *type* (as defined in Table 1.2). A type indication τ is well-formed in the context Δ if τ denotes the `boolean` keyword or the `nat` or `array` keywords with a *well-formed* constraint, and a well-formed element type in the `array` case.

ETypeBool

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

ETypeNat

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

ETypeArray

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

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

- its bounds are globally static expressions [13, p.36] of the `nat` type.
- 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` *type (i.e, value of type nat).*

Premises

SE_g states that an expression is *globally* static (see 1.5.9).

ECONSTR

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

When considering a type indication in a generic constant declaration, the definition of well-formedness differs slightly from the general definition. A type indication τ associated to a generic constant declaration is well-formed if τ denotes the `boolean` keyword, or the `nat` keyword with a *well-formed* constraint.

The $etype_g$ relation is specially defined to check the well-formedness of a subtype indication associated with a generic constant declaration.

ETYPEGBOOL

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

ETYPEGNAT

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

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 c is well-formed if:

- its bounds are locally static expressions [13, p.36] of the `nat` type.
- its lower bound value is inferior or equal to its upper bound value.

ECONSTRG

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

1.5.6 Behavior elaboration.

Elaboration of concurrent statements.

As it is, the elaboration of the composition of concurrent statements is performed in a sequential manner.

CsParElab

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

CsNullElab

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

Process elaboration.

Premises

$\texttt{valid}_{ss}$ states that a sequential statement is well-typed.

Side conditions

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

PsElab

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

Process declarative part elaboration.

The *evars* relation builds a local environment out of a process declarative part.

VarElab

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

VarElabComp

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

Component instantiation elaboration.

Premises

- The *emapg* relation binds a generic map to a function $\mathcal{M} \in id \nrightarrow value$ (see definition below).
- $\texttt{valid}_{ipm}$ (resp. $\texttt{valid}_{opm}$) states that a in port map (resp. out port map) is valid, i.e well-formed and well-typed (see 1.5.10).

Side conditions

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

CompElab

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

A port map is a mapping between signals coming from an embedding design (Δ) and ports of an internal component instance (Δ_c). The formal part of an port map entry (i.e, left of the arrow) belongs to the internal component, whereas the actual part (i.e, right of the arrow) refers to the embedding design. Therefore, we need both Δ and Δ_c to verify if a port map is well-typed leveraging the $\texttt{valid}_{pm}$ 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, a inconsistent mapping $\mathcal{M}$ is generated by the* emapg *that will make the* elab *relation, taking $\mathcal{M}$ as a parameter, never derivable. Therefore, the* elab *relation does an implicit validity check on the generic map.*

AssocGElab

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

GmapElabComp

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

An assoc_g entry doesn't allow indexed id in its formal part, due to the restriction of generic constants to scalar types. Note that this restriction is not imposed by the VHDL language reference; it is our stance towards a simplification of the VHDL semantics.

1.5.7 Implicit default value.

According to the VHDL reference, when declaring 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.

DefaultVBool

$$\frac{}{\texttt{bool} \xrightarrow{defaultv} \bot}$$

DefaultVCNat

$$\frac{}{\texttt{nat}(n, m) \xrightarrow{defaultv} n}\ n \leq m$$

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

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

1.5.8 Typing relation.

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

1.5.9 Static expressions.

Static expressions are either locally static or globally static.

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.

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

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.

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

1.5.10 Valid port map.

The $\texttt{valid}_{ipm}$ predicate states that an *in* 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:

- All ports defined in Δ_c are exactly mapped once in the portmap.
- For each 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 port map association list.
- $\texttt{check}_{pm}$ checks the validity of a port map based on the corresponding port list and the set built by the $list_{ipm}$ relation.

VALIDIPM

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

Side conditions

$\text{id}_p \notin \mathcal{L}$ checks that the port identifier id_p is not already mapped.

LISTIPMSIMPLE

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

Premises

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

Side conditions

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

ListIPMPartial

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

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 $Ports \in id \twoheadrightarrow type$ appear in $\mathcal{L}$ as a simple identifier, or if the port identifier is of type array, then all couples (id,i) must belong to $\mathcal{L}$, where i denotes all indexes of the array range and id, the port id.

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

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 appears at most once in the output port map.
- Two different output ports cannot be connected to the same signal.
- For each port map entry, the formal and the actual part are of the exact same type (i.e, in the sense of the Leibniz equality).

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 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

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

Side conditions

- $\mathrm{id}_p \notin \mathcal{L}$ checks that the port identifier id_p is not already mapped (i.e, is not already in the formal part of the port map).
- $\mathrm{id}_s \notin \mathcal{L}_{ids}$ checks that the signal identifier id_s is not already mapped (i.e, is not already in the actual part of the port map).
- $\Delta_c(\mathrm{id}_p) = \Delta(\mathrm{id}_s) = T$ checks that id_p and id_s are exactly of the same type.

LISTOPMSIMPLETOSIMPLE

$$\Delta, \Delta_c, \mathcal{L}, \mathcal{L}_{ids} \vdash \mathrm{id}_p \Rightarrow \mathrm{id}_s \xrightarrow{list_{opm}} \mathcal{L} \cup \{id_p\}, \mathcal{L}_{ids} \cup \{id_s\}$$

$\mathrm{id}_p \notin \mathcal{L}, \mathrm{id}_s \notin \mathcal{L}_{ids}$
$\mathrm{id}_p \in Outs(\Delta_c)$
$\mathrm{id}_s \in Sigs(\Delta) \cup Outs(\Delta)$
$\Delta_c(\mathrm{id}_p) = \Delta(\mathrm{id}_s) = T$

Side conditions

$Outs_c(\mathrm{id}_p) = T$ and $Sigs(\mathrm{id}_s) = \texttt{array}(T, n, m)$ checks that the type of id_p and the type of the elements of id_s are the same. Note that id_s must denote an array as id_p is mapped to one of id_s's partial.

LISTOPMSIMPLETOPARTIAL

$$\frac{SE_l(\mathrm{e}_i) \quad \vdash \mathrm{e}_i \xrightarrow{e} v_i \quad v_i \in_c \texttt{nat}(n, m)}{\Delta, \Delta_c, \mathcal{L}, \mathcal{L}_{ids} \vdash \ \mathrm{id}_p \Rightarrow \mathrm{id}_s(\mathrm{e}_i) \xrightarrow{list_{opm}} \mathcal{L} \cup \{id_p\}, \mathcal{L}_{ids} \cup \{(id_s, v_i)\}}$$

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

LISTOPMSIMPLETOOPEN

$$\Delta, \Delta_c, \mathcal{L}, \mathcal{L}_{ids} \vdash \ \mathrm{id}_p \Rightarrow \texttt{open} \xrightarrow{list_{opm}} \mathcal{L} \cup \{id_p\}, \mathcal{L}_{ids}$$

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

Remark 3 (Unconnected output port.). *We forbid the case where an indexed formal part corresponding to the subelement of a composite output port is unconnected, i.e* $\mathrm{id}_p(e_i) \Rightarrow \texttt{open}$*, 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

$$\frac{SE_l(\mathrm{e}_i) \quad \vdash \mathrm{e}_i \xrightarrow{e} v_i \quad v_i \in_c \texttt{nat}(n, m)}{\Delta, \Delta_c, \mathcal{L} \vdash \mathrm{id}_p(\mathrm{e}_i) \Rightarrow \mathrm{id}_s \xrightarrow{list_{opm}} \mathcal{L} \cup \{(id_p, v_i)\}, \mathcal{L}_{ids} \cup \{id_s\}}$$

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

ListOPMPartialToPartial

$$\frac{\begin{array}{lll} SE_l(\mathrm{e'}_i) & \vdash \mathrm{e'}_i \xrightarrow{e} v'_i & v'_i \in_c \mathtt{nat}(n', m') \\ SE_l(\mathrm{e}_i) & \vdash \mathrm{e}_i \xrightarrow{e} v_i & v_i \in_c \mathtt{nat}(n, m) \end{array}}{\Delta, \Delta_c, \mathcal{L} \vdash \mathrm{id}_p(\mathrm{e}_i) \Rightarrow \mathrm{id}_s(\mathrm{e'}_i) \xrightarrow{list_{opm}} \mathcal{L} \cup \{(id_p, v_i)\}, \mathcal{L}_{ids} \cup \{(id_s, v'_i)\}}$$

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

ListOPMCons

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

1.5.11 Valid sequential statements.

The $\mathtt{valid}_{ss}$ predicate states that a sequential statement is well-typed.

Well-typed signal assignment.

WellTypedSigAssign

$$\frac{\Delta, \sigma, \Lambda \vdash e \xrightarrow{e} v \qquad v \in_c T}{\Delta, \sigma, \Lambda \vdash \mathtt{valid}_{ss}(\mathrm{id}_s \Leftarrow \mathrm{e})}$$

$\mathrm{id}_s \in Sigs(\Delta) \cup Outs(\Delta)$
$\Delta(\mathrm{id}_s) = T$

WellTypedIdxSigAssign

$$\frac{\begin{array}{ll} \Delta, \sigma, \Lambda \vdash e \xrightarrow{e} v & v \in_c T \\ \Delta, \sigma, \Lambda \vdash e_i \xrightarrow{e} v_i & v_i \in_c \mathtt{nat}(n, m) \end{array}}{\Delta, \sigma, \Lambda \vdash \mathtt{valid}_{ss}(\mathrm{id}_s(\mathrm{e}_i) \Leftarrow \mathrm{e})}$$

$\mathrm{id}_s \in Sigs(\Delta) \cup Outs(\Delta)$
$\Delta(\mathrm{id}_s) = \mathtt{array}(T, n, m)$

Well-typed variable assignment.

WellTypedVarAssign

$$\frac{\Delta, \sigma, \Lambda \vdash \mathrm{e} \xrightarrow{e} v \qquad v \in_c T}{\Delta, \sigma, \Lambda \vdash \mathtt{valid}_{ss}(\mathrm{id}_v := \mathrm{e})}$$

$\mathrm{id}_v \in \Lambda$
$\Lambda(\mathrm{id}_v) = (T, val)$

WellTypedIdxVarAssign

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

$\mathrm{id}_v \in \Lambda$
$\Lambda(\mathrm{id}_v) = (\mathtt{array}(T, n, m), val)$

Well-typed if statements.

WellTypedIf

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

WellTypedIfElse

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

Well-typed loop statement.

WellTypedLoop

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

Well-typed rising and falling edge blocks

WellTypedRising

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

WellTypedFalling

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

Well-typed rst blocks

WellTypedRst

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

Well-typed null statement

WellTypedNull

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

1.6 Simulation rules

1.6.1 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, and type-checks the design. 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 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 being currrently simulated 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. This leads to the following hypothesis:

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 Stable primary inputs 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 Stabilization arises from the previous facts.

The $full$ simulation relation takes in parameter a top-level design d, a design store $\mathcal{D} \in id \rightharpoonup design$, an elaborated design $\Delta \in ElDesign(d)$, a dimensioning function $\mathcal{M}_g \in Gens(\Delta) \rightharpoonup value$, a primary input environment $E_p \in (\mathbb{N} \times Clk) \to (Ins(\Delta) \to value)$, a simulation cycle count $\tau \in \mathbb{N}$, and a simulation trace $\theta \in \texttt{list}(\Sigma(\Delta))$, corresponding to the list of states yielded by design d after τ simulation 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

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

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.
- $E_p \in (\mathbb{N} \times Clk) \rightarrow (ident \nrightarrow value)$, 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 $Clk = \{\uparrow, \downarrow\}$).
- τ, the number of simulation cycles to execute. The value of τ is decremented at each clock cycle until it reaches zero (see Section 1.6.2).

1.6.2 Simulation loop.

SIMEND

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

SIMLOOP

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

1.6.3 Simulation cycle.

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

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

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 an $\mathcal{H}$-VHDL design $d \in design$, a design store $\mathcal{D} \in id \rightarrow design$, an elaborated design $\Delta \in ElDesign(d, \mathcal{D})$, a simulation environment $E_p \in (\mathbb{N} \times \{\uparrow, \downarrow\}) \rightarrow (Ins(\Delta) \rightarrow value)$, let us define the relation expressing the update of the values of the input ports of Δ at a given design state $\sigma \in \Sigma(\Delta)$, clock cycle count $\tau \in \mathbb{N}$, and clock event $clk \in \{\uparrow, \downarrow\}$, and thus resulting in a new state $\sigma_i \in \Sigma(\Delta)$. The relation is written $\mathtt{Inject}_{clk}(\sigma, E_p, \tau, \sigma_i)$ 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 cycle relation states that the design state σ'' is the result of the execution of concurrent statement *cs* over one simulation cycle, this starting from state σ. As told in Hypothesis 1, we update the input port values at each clock event. New input port values are coming from the environment E_p. The updates are made in the definitions of σ_i and σ_i'. These definitions are expressed as side conditions.

SIMCYC

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

Remark 4 (Input ports and signal store)**.** *For a given* $\Delta \in Design$, $\sigma \in \Sigma(\Delta)$, $E_p \in \mathbb{N} \to Clk \to (Ins(\Delta) \to value)$, $\tau \in \mathbb{N}$, $clk \in Clk$, *we have* $\texttt{dom}(E_p(\tau, clk)) \subseteq \texttt{dom}(\mathcal{S})$, *where* $\sigma = <\mathcal{S},\mathcal{C},\mathcal{E}>$. *Indeed, the input ports of* Δ *that constitutes the domain of* $E_p(\tau, clk)$ *are a subset of the set of signals. The set of signals constitutes the domain of the signal store of* σ *(i.e,* $\mathcal{S}$*); thus we have* $\texttt{dom}(E_p(\tau, clk)) \subseteq \texttt{dom}(\mathcal{S})$.

1.6.4 Initialization rules

INIT

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

At the initialization phase, the block of sequential instructions of all processes is executed exactly once (*runinit*), then a stabilization phase follows (*stabilize*). It is during the initialization phase that the first part of `rst` blocks is executed. A block (`rst` ss ss') is equivalent to (`if` (rst = false) `then` ss `else` ss') where `rst` is a reserved signal identifier. Therefore, when considering a `rst` block, the *runinit* relation executes the ss block; at every other moment of the simulation, the ss' block is executed. This mimicks the conventional proceeding of a simulation where the `rst` signal (for *reset* signal) is set to false during the initialization (only during the *runinit* phase, not during the stabilization phase), and then is set to true for the rest of the simulation.

Evaluation of a process statement

Premises

- The *i* flag of the ss_i relation indicates that all sequential instructions responding to the initialization phase (i.e, `rst` blocks) will be executed.
- The set of events of state σ is emptied ($NoEv(\sigma)$) before the evaluation of the process

statement body, and the resulting state is the starting state that will be written through the execution of the process statement body.

PSRUNINIT

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

Evaluation of a component instantiation statement

COMPRUNINIT

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

Evaluation of the composition of concurrent statements

PARRUNINIT

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

NULLRUNINIT

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

The `merge` function 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 store functions 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 unchanged signals. The same goes for the resulting component instance state store C_m. The new set of events $\mathcal{E}_m$ is the union between set of events at state s and s'. The `merge` correctly merges the state o, s and s' only if the set of events of s and s' are disjoint. Fortunately, the PARRUNINIT rule that calls to the `merge` function defines the condition of disjoint set of events as a side condition.

```
1 Definition merge(o,s,s') :=
2    let o = (S_o,C_o,E_o) in
3    let s = (S,C,E) in
4    let s' = (S',C',E') in
5    let S_m = λid. if id ∈ E then S(id) else if id ∈ E' then S'(id) else S_o(id)
6    let C_m = λid. if id ∈ E then C(id) else if id ∈ E' then C'(id) else C_o(id)
7    let E_m = E ∪ E' in (S_m,C_m,E_m).
```

Remark 5 (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.*

1.6.5 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` in their sensitivity list.

Evaluation of a process statement

PsRENoClk

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

Premises

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

PsREClk

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

PsFENoClk

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

Premises

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

PsFEClk

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

Evaluation of a component instantiation statement

COMPRE

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

COMPFE

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

Evaluation of the composition of concurrent statements

PARFE

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

NULLFE

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

PARRE

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

NULLRE

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

1.6.6 Stabilization rules

STABILIZEEND

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

STABILIZELOOP

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

Evaluation of a process statement

Premises

The c flag (for *combinational*) on the ss_c relation indicates that instructions responding to clock events (`falling` and `rising` blocks) and instructions executed during the initialization phase only (`rst` blocks) will not be considered.

PSCOMB

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

Evaluation of a component instantiation statement

COMPCOMB

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

Evaluation of the composition of concurrent statements

PARCOMB

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

NULLCOMB

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

1.6.7 Evaluation of input and output port maps

MAPIPSIMPLE

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

MAPIPPARTIAL

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

MAPIPCOMP

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

Remark 6 (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 permit the evaluation of output port identifiers as regular signal identifier expressions.*

MAPOPOPEN

$$\frac{}{\Delta, \Delta_c, \sigma, \sigma_c \vdash \text{id}_f \Rightarrow \texttt{open} \xrightarrow{mapop} \sigma_c}$$

MAPOPSIMPLETOSIMPLE

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

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

MAPOPSIMPLETOPARTIAL

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

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

MAPOPPARTIALTOSIMPLE

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

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

MAPOPPARTIALTOPARTIAL

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

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

MAPOPCOMP

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

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

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

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

1.6.8 Evaluation of sequential statements

The *ss* symbol indicates that the evaluation of the considered sequential statement does not depend on a specific flag (i.e, the *c*, *i*, ↑ or ↓ flag). In the rules of the *ss* relation, a *ss* 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).

Premises

The premise $\mathcal{S}(id_s) \in_c T$ checks that the value associated to signal id_s in the signal store of σ complies with type T, where T is the type associated with signal id_s in Δ.

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

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

Variable assignment statement

A variable assignment statement modifies the variable values in the local environment.

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

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

If statement

IF⊤

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

IF⊥

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

IFELSE⊤

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

IFELSE⊥

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

Loop statement

LOOP⊥

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

LOOP⊤

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

LOOPINIT

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

Rising and falling edge block statements

RISINGEDGEDEFAULT

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

FALLINGEDGEDEFAULT

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

RISINGEDGEEXEC

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

FALLINGEDGEEXEC

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

Rst block statement

RSTDEFAULT

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

RSTEXEC

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

Composition of sequential statements and null statement

SEQSTMT

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

NULLSTMT

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

1.6.9 Evaluation of expressions

NAT

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

FALSE

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

TRUE

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

AGGREG

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

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

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

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

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

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

where $\mathtt{get\_at}(i, a)$ is a function returning the i-th element of array a.

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

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

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

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

DIFFOP

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

where *eq* is the equality relation established for all types defined in the semantics.

PARENTH

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

Appendix A

The place design in concrete and abstract VHDL syntax

```
entity place is
  generic(
    input_arcs_number : natural := 1;
    output_arcs_number : natural := 1;
    maximal_marking : natural := 1
  );
  port(
    clock : in std_logic;
    reset_n : in std_logic;
    initial_marking : in natural range 0 to maximal_marking;
    input_arcs_weights : in weight_vector_t(input_arcs_number-1 downto 0);
    output_arcs_types : in arc_vector_t(output_arcs_number-1 downto 0);
    output_arcs_weights : in weight_vector_t(output_arcs_number-1 downto 0);
    input_transitions_fired : in std_logic_vector(input_arcs_number-1 downto 0);
    output_transitions_fired : in std_logic_vector(output_arcs_number-1 downto 0);
    output_arcs_valid : out std_logic_vector(output_arcs_number-1 downto 0);
    priority_authorizations : out std_logic_vector(output_arcs_number-1 downto 0);
    reinit_transitions_time : out std_logic_vector(output_arcs_number-1 downto 0);
    marked : out std_logic
  );
end place;

architecture place_architecture of place is

  subtype local_weight_t is natural range 0 to maximal_marking;

  signal s_input_token_sum : local_weight_t;
  signal s_marking : local_weight_t;
  signal s_output_token_sum : local_weight_t;

begin

```

```
33
34  input_tokens_sum : process(input_arcs_weights, input_transitions_fired)
35    variable v_internal_input_token_sum : local_weight_t;
36  begin
37    v_internal_input_token_sum := 0;
38
39    for i in 0 to input_arcs_number -1 loop
40      if (input_transitions_fired(i) = '1') then
41
42        v_internal_input_token_sum := v_internal_input_token_sum + input_arcs_weights(i)
              ;
43
44      end if;
45    end loop;
46
47    s_input_token_sum <= v_internal_input_token_sum;
48  end process input_tokens_sum;
49
50  output_tokens_sum : process(output_arcs_types, output_arcs_weights,
      output_transitions_fired)
51    variable v_internal_output_token_sum : local_weight_t;
52  begin
53    v_internal_output_token_sum := 0;
54
55    for i in 0 to output_arcs_number -1 loop
56      if (output_transitions_fired(i) = '1' and output_arcs_types(i) = arc_t(BASIC)) then
57        v_internal_output_token_sum := v_internal_output_token_sum +
              output_arcs_weights(i);
58      end if;
59    end loop;
60
61    s_output_token_sum <= v_internal_output_token_sum;
62  end process output_tokens_sum;
63
64  marking : process(clock, reset_n, initial_marking)
65  begin
66    if (reset_n = '0') then
67      s_marking <= initial_marking;
68    elsif rising_edge(clock) then
69      s_marking <= s_marking + (s_input_token_sum -s_output_token_sum);
70    end if;
71  end process marking;
72
73  determine_marked : process(s_marking)
74  begin
75    if (s_marking = 0) then
76      marked <= '0';
```

```
77      else
78        marked <= '1';
79      end if;
80    end process determine_marked;
81
82    marking_validation_evaluation : process(output_arcs_types, output_arcs_weights,
          s_marking)
83    begin
84      for i in 0 to output_arcs_number -1 loop
85        if ( ( ((output_arcs_types(i) = arc_t(BASIC)) or (output_arcs_types(i) = arc_t(TEST)))
              and (s_marking >= output_arcs_weights(i)) )
86             or ( (output_arcs_types(i) = arc_t(INHIBITOR)) and (s_marking <
                  output_arcs_weights(i)) ) )
87        then
88          output_arcs_valid(i) <= '1';
89        else
90          output_arcs_valid(i) <= '0';
91        end if;
92      end loop;
93    end process marking_validation_evaluation;
94
95    priority_evaluation : process(output_arcs_types, output_arcs_weights,
          output_transitions_fired, s_marking)
96      variable v_saved_output_token_sum : local_weight_t;
97    begin
98      v_saved_output_token_sum := 0;
99
100     for i in 0 to output_arcs_number -1 loop
101       if (s_marking >= v_saved_output_token_sum + output_arcs_weights(i)) then
102         priority_authorizations(i) <= '1';
103       else
104         priority_authorizations(i) <= '0';
105       end if;
106
107       if ((output_transitions_fired(i) = '1') and (output_arcs_types(i) = arc_t(BASIC)))
            then
108         v_saved_output_token_sum := v_saved_output_token_sum + output_arcs_weights(i);
109       end if;
110
111     end loop;
112   end process priority_evaluation;
113
114   reinit_transitions_time_evaluation : process(clock, reset_n)
115   begin
116     if (reset_n = '0') then
117       reinit_transitions_time <= (others => '0');
118     elsif rising_edge(clock) then
```

```
119       for i in 0 to output_arcs_number -1 loop
120         if ( (((output_arcs_types(i) = arc_t(BASIC)) or (output_arcs_types(i) = arc_t(TEST)))
121                 and (s_marking -s_output_token_sum < output_arcs_weights(i))
122                 and (s_output_token_sum > 0))
123               or output_transitions_fired(i) = '1' )
124         then
125           reinit_transitions_time(i) <= '1';
126         else
127           reinit_transitions_time(i) <= '0';
128         end if;
129       end loop;
130     end if;
131   end process reinit_transitions_time_evaluation;
132
133 end place_architecture;
```

LISTING A.1: The place design in concrete VHDL syntax.

Appendix B

The transition design in concrete and abstract VHDL syntax

```
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;

architecture transition_architecture of transition is

  signal s_condition_combination : std_logic;
  signal s_enabled : std_logic;
  signal s_firable : std_logic;
  signal s_firing_condition : std_logic;
  signal s_priority_combination : std_logic;
  signal s_reinit_time_counter : std_logic;
  signal s_time_counter : natural range 0 to maximal_time_counter;

begin

```

```
33  condition_evaluation : process(input_conditions)
34    variable v_internal_condition : std_logic;
35  begin
36    v_internal_condition := '1';
37
38    for i in 0 to conditions_number -1 loop
39      v_internal_condition := v_internal_condition and input_conditions(i);
40    end loop;
41
42    s_condition_combination <= v_internal_condition;
43  end process condition_evaluation;
44
45  enable_evaluation : process(input_arcs_valid)
46    variable v_internal_enabled : std_logic;
47  begin
48    v_internal_enabled := '1';
49
50    for i in 0 to input_arcs_number -1 loop
51      v_internal_enabled := v_internal_enabled and input_arcs_valid(i);
52    end loop;
53
54    s_enabled <= v_internal_enabled;
55  end process enable_evaluation;
56
57  reinit_time_counter_evaluation : process(reinit_time, s_enabled)
58    variable v_internal_reinit_time_counter : std_logic;
59  begin
60    v_internal_reinit_time_counter := '0';
61
62    for i in 0 to input_arcs_number -1 loop
63      v_internal_reinit_time_counter := v_internal_reinit_time_counter or reinit_time(i)
          ;
64    end loop;
65
66    s_reinit_time_counter <= v_internal_reinit_time_counter;
67  end process reinit_time_counter_evaluation;
68
69  time_counter : process(reset_n, clock)
70  begin
71    if (reset_n = '0') then
72      s_time_counter <= 0;
73    elsif falling_edge(clock) then
74      if ((s_enabled = '1') and (transition_type /= transition_t(NOT_TEMPORAL))) then
75        if (s_reinit_time_counter = '0') then
76          if (s_time_counter < maximal_time_counter) then
77            s_time_counter <= s_time_counter + 1;
78          end if;
```

```
79          else
80            s_time_counter <= 1;
81          end if;
82        else
83          s_time_counter <= 0;
84        end if;
85      end if;
86    end process time_counter;
87
88    firing_condition_evaluation : process (s_enabled, s_condition_combination,
          s_reinit_time_counter, s_time_counter)
89    begin
90      if ( (s_condition_combination = '1')
91           and (s_enabled = '1')
92           and ( (transition_type = transition_t(NOT_TEMPORAL))
93
94                 or ( (transition_type = transition_t(TEMPORAL_A_B))
95                      and (s_reinit_time_counter = '0')
96                      and (s_time_counter >= (time_A_value-1))
97                      and (s_time_counter < time_B_value)
98                      and (time_A_value /= 0)
99                      and (time_B_value /= 0) )
100
101                or ( (s_reinit_time_counter = '0')
102                     and (time_A_value /= 0)
103                     and ( ((transition_type = transition_t(TEMPORAL_A_A))
104                            and (s_time_counter = (time_A_value-1)))
105                     or ( (transition_type = transition_t(TEMPORAL_A_INFINITE))
106                          and (s_time_counter >= (time_A_value-1)) )
107                )
108
109                or ( (transition_type /= transition_t(NOT_TEMPORAL))
110                     and (s_reinit_time_counter = '1')
111                     and (time_A_value = 1) )
112                )
113     ) then
114       s_firing_condition <= '1';
115     else
116       s_firing_condition <= '0';
117     end if;
118   end process firing_condition_evaluation;
119
120   priority_authorization_evaluation : process(priority_authorizations)
121     variable v_priority_combination : std_logic;
122   begin
123     v_priority_combination := '1';
124
```

```
125     for i in 0 to input_arcs_number -1 loop
126       v_priority_combination := v_priority_combination and priority_authorizations(i);
127     end loop;
128
129     s_priority_combination <= v_priority_combination;
130   end process priority_authorization_evaluation;
131
132   firable : process(reset_n, clock)
133   begin
134     if (reset_n = '0') then
135       s_firable <= '0';
136     elsif falling_edge(clock) then
137       s_firable <= s_firing_condition;
138     end if;
139   end process firable;
140
141   fired_evaluation : process (s_firable, s_priority_combination)
142   begin
143     fired <= s_firable and s_priority_combination;
144   end process fired_evaluation;
145
146 end transition_architecture;
```

LISTING B.1: The transition design in concrete VHDL syntax.

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] 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).

[18] 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).

[19] 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.

[20] 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.

[21] 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).

[22] 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).

[23] 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).