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).
- There are different kinds of component instantiaton statement but we only refer to design instantiation statment.

1.1 Presentation of the VHDL language

1.2 Choosing a formal semantics for VHDL

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

In the above entry, id_e indicates the entity identifier and id_a the architecture identifier of the declared design. The gens entry corresponds to the generic clause, i.e. the declaration list for the generic constants of the design. A generic constant is declared via the gdecl entry; it generic

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

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

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

1.3.2 Concurrent statements

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

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

Process statement

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

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

Component instantiation statement

The identifier id_c represents the name of component instance. Identifier id_e points out the name of the design, i.e. the entity identifier, being instantiated here. The gmap entry describes the list

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)

of associations between generic constant identifiers and expressions. The ipmap entry is the list of associations between input port identifiers (or indexed identifiers) and expressions. The opmap entry is the list of associations between output port identifiers (or indexed identifiers) and signal names, or the `open` keyword. Associating the `open` keyword with an output port identifier in an output port pmap indicates that the port is not connected.

1.3.3 Sequential statements

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

The ss entry defines the sequential statements that compose the body of processes. The signal assignment statement is represented with the $\Leftarrow$ operator; the variable assignment statement with the := operator. Among the control flow statements, the `falling` statement (resp. `rising` ss) declares a block of sequential statements to be executed only at the falling edge (resp. rising edge) of the clock signal (see Section 1.6.5). Also, the `rst` statement declares two blocks, the first one must be executed during the initialization phase of the simulation; otherwise, the second one is executed (see Section 1.6.4).

1.3.4 Expressions, names and types.

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

The expression entry declares a set of operators over Boolean expressions, and natural numbers expressions. The natural non-terminal represents the set of natural numbers ($\mathbb{N}$). The id non-terminal represents the set of identifiers, comparable to the set of strings, or any infinitly enumerable set. In what follows, identifiers will be enclosed in double quotes at the time of their declaration, and they will appear without double quotes when used in expressions.

1.3.5 Examples of $\mathcal{H}$-VHDL abstract syntax

Listing 1.1 gives the translation in abstract $\mathcal{H}$-VHDL syntax of the declarative part of the transition design presented in Listings **??** and **??** in concrete VHDL syntax.

```
1  design "transition" "transition_architecture"
2    -- Generic clause
3    (("transition_type", natural(0, 2), 0),
4     ("input_arcs_number", natural(0, NATMAX), 1),
5     ("conditions_number", natural(0, NATMAX), 1),
6     ("maximal_time_counter", natural(0, NATMAX), 1))
7
8    -- Port clause
9    ((in, "input_conditions", array(boolean, 0, conditions_number−1)),
10    (in, "time_A_value", natural(0, maximal_time_counter)),
11    (in, "time_B_value", natural(0, maximal_time_counter)),
12    (in, "input_arcs_valid", array(boolean, 0, input_arcs_number−1)),
13    (in, "reinit_time", array(boolean, 0, input_arcs_number−1)),
14    (in, "priority_authorizations", array(boolean, 0, input_arcs_number−1)),
15    (out, "fired", boolean))
16
17   -- Internal signals
18   (("s_condition_combination", boolean),
19    ("s_enabled", boolean),
20    ...
21    ("s_time_counter", natural(0, maximal_time_counter)))
22
23   -- Concurrent statements
24   cs
```

LISTING 1.1: The transition design's declarative parts in abstract $\mathcal{H}$-VHDL syntax.

In Listing 1.1, the type indication of the "`transition_type`" generic constant has been transformed from the `temporal_t` enumeration type to the `natural(0,2)` type. The `temporal_t` enumeration type, defined through the three values `{NOT_TEMPORAL, TEMPORAL_A_A, TEMPORAL_A_B}`, is naturally transformed into a natural range. This transformation is only valid because the `temporal_t` type is only defined and used in the transition design which has a static behavior. Also, the `std_logic` type defined in the VHDL Std Logic 1164 library [4] is transformed into the Boolean type, and the `std_logic_vector` is transformed into the array of Booleans type. This transformation is valid because, in the transition and place designs and also in the generated designs, we are only referring to the values '0' and '1' among all the values enumerated by the `std_logic` type (see [4, p. 2]). Also, note that the `clock` and `reset_n` input ports declared in the port clause of the transition design are removed in the hvhdl version. In the $\mathcal{H}$-VHDL abstract syntax, the if statements that were testing the value of the `clock` and the `reset_n` signals have been turned into specific sequential statements (i.e `rst`, `falling` and `rising` blocks). Thus, we don't need the `clock` and `reset_n` signals in the port declaration list anymore.

Listing 1.2 presents the translation in $\mathcal{H}$-VHDL abstract syntax of the two of the processes presented in Listing **??**.

```
process ("condition_evaluation", ("input_conditions"),
        (("v_internal_condition", boolean)),
  v_internal_condition := ⊤;
  (for ("i", 0, conditions_number − 1)
    (v_internal_condition := v_internal_conditions and input_conditions(i)));
  s_condition_combination ⇐ v_internal_condition
) | |
process ("firable", ("clk"), ∅,
  rst (s_firable ⇐ ⊥)
      (falling (s_firable ⇐ s_firing_condition))
)
```

LISTING 1.2: The `condition_evaluation` and `firable` processes in $\mathcal{H}$-VHDL abstract syntax. The two processes are defined in the behavior of the transition design.

In Listing 1.2, inner blocks of sequential statements are enclosed between parentheses to ease the reading, albeit parentheses are not part of the $\mathcal{H}$-VHDL syntax. The body of the `firable` process gives an example of the use of a `rst` block and a `falling` block. One can see, from the comparison of Listing **??** and 1.2, that a `rst` ss ss' statement is the translation of the concrete VHDL statement `if reset_n = '0' then ss else ss' endif;`. Comparing the same listings, a `falling` ss statement is the translation of the concrete VHDL statement `if falling_edge(clock) then ss end if;`.

Listing 1.3 shows the translation in $\mathcal{H}$-VHDL abstract syntax of the component instantiation statement laid out in Listing **??**.

```
comp (id_t, "transition",
  -- Generic map
  (transition_type ⇒ 0,
   input_arcs_number ⇒ 1,
   conditions_number ⇒ 1,
   maximal_time_counter ⇒ 1),
  -- Input port map
  (time_A_value ⇒ 0,
   time_B_value ⇒ 0,
   input_conditions(0) ⇒ id_0,
   input_arcs_valid(0) ⇒ id_1,
   priority_authorizations(0) ⇒ ⊤,
   reinit_time(0) ⇒ id_2),
  -- Output port map
  (fired ⇒ id_3)
```

LISTING 1.3: An example of instantiation of the transition design in $\mathcal{H}$-VHDL abstract syntax.

The statement of Listing 1.3 is almost similar to the one of Listing **??**. The `NOT_TEMPORAL` value associated to the `transition_type` constant is turned into 0 (remember the `temporal_t` enumeration type is transformed into a natural range).

1.4 Preliminary definitions

1.4.1 Semantic domains

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

type	::=	`bool` | `nat(`*nat*`,`*nat*`)` | `array (`*type*`,`*nat*`,`*nat*`)`
value	::=	*bool* | *nat* | *array*
bool	::=	'⊤' | '⊥'
nat	::=	`0` | `1` | `...` | `NATMAX`
array	::=	`(`$value^{+}$`)`

TABLE 1.1: The *type* and *value* semantical types.

In Table 1.1, 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*.*

1.4.2 Elaborated design and design state

Now, let us define the structure of an elaborated design which is a structure bound to a given $\mathcal{H}$-VHDL design and to a design store, i.e a global environment mapping identifiers to $\mathcal{H}$-VHDL designs. Only the designs referenced into the global design store can be instantiated as component instances in the behavior of a given design. The elaborated design structure is built

during the elaboration phase (see Section 1.5). Then, the elaborated design will act as a run-time environment in the expression of the simulation rules. Let $ElDesign(d, \mathcal{D})$ be the set of the elaborated designs for a given $\mathcal{H}$-VHDL design d and a design store $\mathcal{D}$. An elaborated design is a composite environment built out of multiple sub-environments. Each sub-environment is a table, represented as a function, mapping identifiers of a certain category of constructs (e.g, input port identifiers) to their declaration information (e.g, type indication for input ports). We represent an elaborated design as a record where the fields are the sub-environments. An elaborated design is defined as follows:

Definition 1 (Elaborated Design). *For a given $\mathcal{H}$-VHDL design $d \in design$ s.t. $d =$ `design` id_e id_a gens ports sigs behavior and a given design store $\mathcal{D} \in entity\text{-}id \rightharpoonup design$, an elaborated design $\Delta \in ElDesign(d, \mathcal{D})$ is a record $<Gens, Ins, Outs, Sigs, Ps, Comps>$ where:*

- *$Gens \in generic\text{-}id \rightarrow (type \times value)$ where $generic\text{-}id = \{id \mid (id, \tau, e) \in gens\}$, is the partial function yielding the type and the value of generic constants.*
- *$Ins \in input\text{-}id \rightarrow type$ where $input\text{-}id = \{id \mid (\texttt{in}, id, \tau) \in ports\}$, is the partial function yielding the type of input ports.*
- *$Outs \in output\text{-}id \rightarrow type$ where $output\text{-}id = \{id \mid (\texttt{out}, id, \tau) \in ports\}$, the partial function yielding the type of output ports.*
- *$Sigs \in declared\text{-}signal\text{-}id \rightarrow type$ where $declared\text{-}signal\text{-}id = \{id \mid (id, \tau) \in sigs\}$, the partial function yelding the type of declared signals.*
- *$Ps \in process\text{-}id \rightarrow (variable\text{-}id(id_p) \rightarrow (type \times value))$ 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.*
- *$Comps \in component\text{-}id \rightarrow ElDesign(d_e, \mathcal{D})$, 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). When there is no ambiguity, we write $\Delta(x)$ to denote the value returned for identifier x, where x is looked up in the appropriate field of Δ. We write $x \in \Delta$ to state that identifier x is defined in one of Δ's fields. We note $\Delta(x) \leftarrow v$ the overriding of the value associated to identifier x with value v in the appropriate field of Δ, $\Delta \cup (x, v)$ to note the addition the mapping from identifier x to value v in the appropriate field of Δ, that assuming $x \notin \Delta$. We write $x \in \mathcal{F}(\Delta)$, where $\mathcal{F}$ is a field of Δ, when more precision is needed regarding the lookup of identifier x in the record Δ.

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

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

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

Notation 2 (No events design state)**.** *For a given $\mathcal{H}$-VHDL design d, a design store $\mathcal{D}$, and an elaborated design $\Delta \in ElDesign(d, \mathcal{D})$, the function $NoEv \in \Sigma(\Delta) \rightarrow \Sigma(\Delta)$ returns a design state similar to the one passed in parameter 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. Even though the elaboration of a design is described in the LRM, the formalization of this phase has been performed in few works only [1, 2, 5], and never in a setting that covers both syntactical well-formedness and type-checking of the designs. We are interested in the formalization of the elaboration phase because we are interested in the *well-formedness* of the programs generated by the HILECOP transformation. Here, the term well-formedness refers to a syntactically valid design, w.r.t. the syntactic rules of the VHDL language, and to a well-typed design, w.r.t. the typing rules defined in the LRM. Formalizing the elaboration phase is also a way to define how the runtime environment and the runtime state of the simulation are built. For now, we haven't tackle down the proof that the $\mathcal{H}$-VHDL designs generated by HILECOP are elaborable, i.e. syntactically well-formed and well-typed. As explained in Chapter **??**, this task is foreseen in our work perspectives. Contrary to what is prescribed in the LRM [3, p. 166], we are not dealing with the transformation of the component instantiation statements into block statements in our formalization of the elaboration phase. We prefer to preserve the hierarchical structure of the design (i.e. its

composite structure) during its elaboration. We argue that dealing with component instantiation statements instead of block statements does not complexify the semantics of the $\mathcal{H}$-VHDL simulation rules.

In the following sections describing the elaboration and the simulation rules of the $\mathcal{H}$-VHDL semantics, the green frames adds explanations about the premises of the rules and the ref frames brings explanations about the side conditions of the rules.

1.5.1 Design elaboration

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

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

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

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.

Example of a design elaboration

The elaborated design and the default design state resulting of the elaboration of the transition design for a given dimensioning function $\mathcal{M}$ are the presented through the Δ and σ_e structures in Figures 1.1 and 1.2. The value of the dimensioning function $\mathcal{M}$ is the set of couples $\{(\text{"}transition\_type\text{"}, 2), (\text{"}conditions\_number\text{"}, 2), (\text{"}input\_arcs\_number\text{"}, 4)\}$.

$$
\begin{aligned}
\Delta := \{ & \\
& Gens := \{(\text{"}transition\_type\text{"}, (nat(0,2),2)), \\
& \qquad (\text{"}conditions\_numbers\text{"}, (nat(0,\texttt{NATMAX}),2)) \\
& \qquad (\text{"}input\_arcs\_number\text{"}, (nat(0,\texttt{NATMAX}),4)) \\
& \qquad (\text{"}maximal\_time\_counter\text{"}, (nat(0,\texttt{NATMAX}),1))\}, \\
& \\
& Ins := \{(\text{"}input\_conditions\text{"}, array(bool,0,1)), \\
& \qquad (\text{"}time\_A\_value\text{"}, nat(0,1)), \\
& \qquad (\text{"}time\_B\_value\text{"}, nat(0,1)), \\
& \qquad (\text{"}input\_arcs\_valid\text{"}, array(bool,0,3)), \\
& \qquad (\text{"}reinit\_time\text{"}, array(bool,0,3)), \\
& \qquad (\text{"}priority\_authorizations\text{"}, array(bool,0,3))\}, \\
& \\
& Outs := \{(\text{"}fired\text{"}, bool)\} \\
& Sigs := \{(\text{"}s\_condition\_combination\text{"}, bool), \\
& \qquad (\text{"}s\_enabled\text{"}, bool), \\
& \qquad (\text{"}s\_time\_counter\text{"}, nat(0,1))\}, \\
& \\
& Ps := \{(\text{"}condition\_evaluation\text{"}, \{(\text{"}v\_internal\_condition\text{"}, (bool, \bot))\}), \\
& \qquad (\text{"}firable\text{"}, \emptyset)\} \\
& \\
& Comps := \emptyset \\
\} &
\end{aligned}
$$

FIGURE 1.1: An elaborated version of the transition design; the sub-environments of the elaborated design structure (i.e. functions) are represented by sets of couples.

For the sake on conciseness, Figure 1.1 gives only a part of the *Sigs* and *Ps* sub-environments resulting from the elaboration of the transition design. In Figure 1.1, the generic constants of the transition design received their values from the dimensioning function $\mathcal{M}$. As it is not defined in the domain of $\mathcal{M}$, the `maximal_time_counter` generic constant is associated with its default value. Also, all the types associated with ports and internal signals have been *resolved*; i.e. the expressions qualifying the upper and lower bound of ranges, in the definition of natural range types or array types, have been evaluated. Due to the presence of generic constant identifiers in the expression of type ranges, we had to wait for the generic constants to receive a value. For example, `array(boolean, 0, conditions_number-1)` is the type indication associated with the `input_conditions` input port. The dimensioning function sets the value of the `conditions_number` generic constant to 2. After elaboration, the type indication

`array(boolean, 0, conditions_number-1)` is transformed into the type $array(bool, 0, 1)$. The *Ps* sub-environment associates each process identifier to a local environment, i.e. a mapping between local variables declared in the process and a couple $(type, value)$. In a local environment, each local variable has an initial value corresponding to the implicit default value of its type (see Section 1.5.7). The behavior of the transition design is set with processes only. Thus, the *Comps* sub-environment, that maps each component instance identifier to an elaborated design structure, is empty.

Figure 1.2 shows the default design state σ_e of Δ. In the default design state of an elaborated design, the value of all signals corresponds to the implicit default value of signals (i.e. deduced from the type of signals, see Section 1.5.7).

$$
\begin{aligned}
\sigma_e := \{ & \\
\quad \mathcal{S} := \quad & \{(\text{"}input\_conditions\text{"}, (\bot, \bot)), \\
& (\text{"}time\_A\_value\text{"}, 0), \\
& (\text{"}time\_B\_value\text{"}, 0), \\
& (\text{"}input\_arcs\_valid\text{"}, (\bot, \bot, \bot)), \\
& (\text{"}reinit\_time\text{"}, (\bot, \bot, \bot)), \\
& (\text{"}priority\_authorizations\text{"}, (\bot, \bot, \bot)), \\
& (\text{"}fired\text{"}, \bot), \\
& (\text{"}s\_condition\_combination\text{"}, \bot), \\
& (\text{"}s\_enabled\text{"}, \bot), \\
& (\text{"}s\_time\_counter\text{"}, 0)\}, \\
\\
\quad \mathcal{C} := & \emptyset \\
\quad \mathcal{E} := & \emptyset \\
\} &
\end{aligned}
$$

FIGURE 1.2: The default design state associated built during the elaboration phase along the elaborated design of Figure 1.1. We use values enclosed between parentheses to represent array values.

The component store of design state σ_e is empty as there are no component instances defining the behavior of the transition design. Also, the set of events of a default design state is always empty.

1.5.2 Generic clause elaboration

The *egens* relation elaborates a list of generic constant declarations.

Premises

- $etype_g$ transforms a type indication, specifically attached to a generic constant declaration, into a *type* instance and checks its well-formedness (see Section 1.5.5).
- The e relation links an expression e to its value v in a given context (see Section 1.6.9).

The context of evaluation for an expression is composed of a given elaborated design, a given design state, and given local environment. We omit symbols at the left of the thesis when they refer to empty structures. For instance, $\vdash e \xrightarrow{e} v$ is a notation for $\Delta_\emptyset, \sigma_\emptyset, \Lambda_\emptyset \vdash \mathrm{e} \xrightarrow{e} v$.

- SE_l states that an expression is *locally* static (see Section 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 Section 1.5.8).

Side conditions

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

GENELABDIMEN

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

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

GENELABDEFAULT

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

GENELABCOMP

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

1.5.3 Port clause elaboration

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

Premises

- The *etype* relation associates a type indication to its corresponding semantic type and checks its well-formedness (see Section 1.5.5).
- The *defaultv* relation associates a given semantic type to its implicit *default* value.

InPortElab

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

OutPortElab

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

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

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

SigElab

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

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 semantic *type* (as defined in Table 1.1). 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.

$$\textsc{ETypeBool}\ \frac{}{\Delta \vdash \texttt{boolean} \xrightarrow{etype} bool} \qquad \textsc{ETypeNat}\ \frac{\Delta \vdash (\mathrm{e}, \mathrm{e}') \xrightarrow{econstr} (v, v')}{\Delta \vdash \texttt{natural}(\mathrm{e}, \mathrm{e}') \xrightarrow{etype} nat(v, v')}$$

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

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

- its bounds are globally static expressions [3, 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 [3, 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

- The $\in_c$ relation states that a given value conforms to a given type (see Section 1.5.5).
- The SE_g relation states that an expression is *globally* static (see Section 1.5.9).

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

When considering a type indication in a generic constant declaration, the definition of well-formedness differs slightly from the general definition. A type indication τ associated to a

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

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

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

- its bounds are locally static expressions [3, p.36] of the `nat` type.
- its lower bound value is inferior or equal to its upper bound value.

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

1.5.6 Behavior elaboration

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

Elaboration of concurrent statements

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

$$\textsc{CsParElab}\ \frac{\mathcal{D}, \Delta, \sigma \vdash \mathrm{cs} \xrightarrow{ebeh} \Delta', \sigma' \qquad \mathcal{D}, \Delta', \sigma' \vdash \mathrm{cs}' \xrightarrow{ebeh} \Delta'', \sigma''}{\mathcal{D}, \Delta, \sigma \vdash \mathrm{cs} \,||\, \mathrm{cs}' \xrightarrow{ebeh} \Delta'', \sigma''} \qquad \textsc{CsNullElab}\ \frac{}{\mathcal{D}, \Delta, \sigma \vdash \texttt{null} \xrightarrow{ebeh} \Delta, \sigma}$$

Process statement elaboration

To elaborate a process statement, the *ebeh* relation associates the process identifier to a local environment. The *ebeh* builds the local environment from the process's local variable declaration list (see the *evars* relation). The *ebeh* relation also checks that the sequential statements composing the body of the process are well-typed (see the $valid_{ss}$ relation in Section 1.5.11).

Premises

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

Side conditions

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

PsElab

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

Process declarative part elaboration

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

VarElab

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

VarElabComp

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

Component instantiation statement elaboration

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

Premises

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

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

A port map is a mapping between expressions and signals coming from an embedding design (Δ) and 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.*

The $emap_g$ relation builds a dimensioning function out of generic map.

AssocGElab

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

GmapElabComp

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

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

Example of component instantiation statement elaboration

The following rule describes the elaboration of the transition component instance presented Listing 1.3. Here, Δ represents the (partially-built) elaborated version of the design that contains the transition component instance id_t in the definition of its behavior. σ represents the (partially-built) default state of the same embedding design. Due to the size of definitions, the generic map (resp. the input port and output port map) of the transition component instance id_t is aliased by g_t (resp. i_t and o_t).

$$\frac{\begin{array}{cc}\mathcal{M}_\emptyset \vdash \mathrm{g}_t \xrightarrow{emapg} \mathcal{M} & \Delta, \Delta_t, \sigma \vdash \texttt{valid}_{ipm}(\mathrm{i}_t) \\ \mathcal{D}, \mathcal{M} \vdash \mathcal{D}_\mathcal{H}(\text{"}transition\text{"}) \xrightarrow{elab} \Delta_t, \sigma_t & \Delta, \Delta_c \vdash \texttt{valid}_{opm}(\mathrm{o}_t)\end{array}}{\mathcal{D}_\mathcal{H}, \Delta, \sigma \vdash \texttt{comp}\ (\mathrm{id}_t, \text{"}transition\text{"}, \mathrm{g}_t, \mathrm{i}_t, \mathrm{o}_t) \xrightarrow{ebeh} \Delta \cup (id_t, \Delta_t), \sigma \cup (id_t, \sigma_t)} \quad \begin{array}{l}\mathrm{id}_t \notin \Delta, \mathrm{id}_t \notin \sigma \\ \text{"}transition\text{"} \in \mathcal{D}_\mathcal{H} \\ \mathcal{M} \subseteq Gens(\Delta_t)\end{array}$$

First, a dimensioning function $\mathcal{M}$ is built out of the generic map of id_t. Figure 1.3 shows the resulting dimensioning function.

$$\mathcal{M} := \{(\text{"}transition\_type\text{"}, 0), (\text{"}input\_arcs\_number\text{"}, 1), (\text{"}conditions\_number\text{"}, 1), (\text{"}maximal\_time\_counter\text{"}, 1)\}$$

FIGURE 1.3: The dimensioning function obtained from the transition component instance id_t defined in Listing 1.3.

Then, the transition design declaration is retrieved from the design store $\mathcal{D}_\mathcal{H}$, which is the specific HILECOP design store. By definition, the HILECOP design store maps the transition and the place identifiers to their corresponding design declaration (the side condition $\text{"}transition\text{"} \in \mathcal{D}_\mathcal{H}$ is true). Thus, $\mathcal{D}_\mathcal{H}(\text{"}transition\text{"})$ returns `design` "transition" "transition_architecture" $gens_t$ $ports_t$ $sigs_t$ cs_t where $gens_t$, $ports_t$, $sigs_t$ and cs_t correspond to the declarative and behavioral parts of the transition design. Then, the *elab* relation builds the elaborated design Δ_t and the default design state σ_t from the transition design declaration given the dimensioning function $\mathcal{M}$. We do not detail the content of Δ_t and σ_t as it is really close to the content of Figures 1.1 and 1.2. Finally, a mapping between the identifier id_t and Δ_t is added to the *Comps* sub-environment of Δ, i.e. $\Delta \cup (id_t, \Delta_t)$. Also, a mapping between the identifier id_t and the default state σ_t is added to the component store $\mathcal{C}$ of σ, i.e. $\sigma \cup (id_t, \sigma_t)$.

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 [3, p.61, 64, 173]. The $defaultv$ relation determines the default value for a given type.

$$\frac{}{\texttt{bool} \xrightarrow{defaultv} \bot}\ \text{DEFAULTVBOOL} \qquad \frac{}{\texttt{nat}(n, m) \xrightarrow{defaultv} n}\ \text{DEFAULTVCNAT}\ n \leq m$$

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

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

1.5.8 Typing relation

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

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

1.5.9 Static expressions

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

Locally static expressions

An expression is *locally* static if:

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

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

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

Globally static expressions

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

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

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

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

1.5.10 Valid port map

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

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

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 $Ports \in id \nrightarrow type$ 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 [3, 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 $f, f' \in X \nrightarrow Y$, $f \overset{\leftarrow}{\cup} f'$ denotes the overriding union of f and f' such that* $f \overset{\leftarrow}{\cup} f'(x) = \begin{cases} f'(x) & if\ x \in \mathtt{dom}(f') \\ f(x) & otherwise \end{cases}$

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 $\sigma_i = <\mathcal{S} \overset{\leftarrow}{\cup} E_p(\tau, clk), \mathcal{C}, \mathcal{E}>$.*

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

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

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

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

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.

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

Evaluation of a component instantiation statement

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

Evaluation of the composition of concurrent statements

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

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

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'. $\mathcal{S}_m$ 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

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

COMPFE

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

Evaluation of the composition of concurrent statements

PARFE

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

NULLFE

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

PARRE

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

NULLRE

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

1.6.6 Stabilization rules

STABILIZEEND

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

STABILIZELOOP

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

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

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

NullComb

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

1.6.7 Evaluation of input and output port maps

MapipSimple

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

MapipPartial

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

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

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

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

MAPOPSIMPLETOPARTIAL

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

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

MAPOPPARTIALTOSIMPLE

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

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

MAPOPPARTIALTOPARTIAL

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

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

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.

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

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

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$\top$

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

IF$\bot$

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

IFELSE$\top$

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

IFELSE$\bot$

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

Loop statement

LOOP$\bot$

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

LOOP$\top$

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

LOOPINIT

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

Rising and falling edge block statements

RISINGEDGEDEFAULT

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

FALLINGEDGEDEFAULT

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

RISINGEDGEEXEC

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

FALLINGEDGEEXEC

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

Rst block statement

RSTDEFAULT

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

RSTEXEC

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

Composition of sequential statements and null statement

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

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

FALSE

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

TRUE

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

AGGREG

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

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

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

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

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

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

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

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

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

$$\textsc{BoolBinOp}\ \frac{\Delta,\sigma,\Lambda \vdash \mathrm{e} \xrightarrow{e} v \qquad \Delta,\sigma,\Lambda \vdash \mathrm{e}' \xrightarrow{e} v'}{\Delta,\sigma,\Lambda \vdash \mathrm{e}\ \mathrm{op}_{bool}\ \mathrm{e}' \xrightarrow{e} v\ op_{\mathbb{B}}\ v'}\ \mathrm{op}_{bool} \in \{\mathtt{and}, \mathtt{or}\} \qquad \textsc{NotOp}\ \frac{\Delta,\sigma,\Lambda \vdash \mathrm{e} \xrightarrow{e} v}{\Delta,\sigma,\Lambda \vdash \mathtt{not}\ \mathrm{e} \xrightarrow{e} \neg v}$$

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

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_number1 downto 0);
    output_arcs_types : in arc_vector_t(output_arcs_number1 downto 0);
    output_arcs_weights : in weight_vector_t(output_arcs_number1 downto 0);
    input_transitions_fired : in std_logic_vector(input_arcs_number1 downto 0);
    output_transitions_fired : in std_logic_vector(output_arcs_number1 downto 0);
    output_arcs_valid : out std_logic_vector(output_arcs_number1 downto 0);
    priority_authorizations : out std_logic_vector(output_arcs_number1 downto 0);
    reinit_transitions_time : out std_logic_vector(output_arcs_number1 downto 0);
    marked : out std_logic
  );
end place;

architecture place_architecture of place is

  subtype local_weight_t is natural range 0 to maximal_marking;

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

begin

```

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

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

```
119                 and (s_output_token_sum > 0))
120               or output_transitions_fired(i) = '1' )
121           then
122             reinit_transitions_time(i) <= '1';
123           else
124             reinit_transitions_time(i) <= '0';
125           end if;
126         end loop;
127       end if;
128     end process reinit_transitions_time_evaluation;
129
130   end place_architecture;
```

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

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_number1 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_number1 downto 0);
    reinit_time : in std_logic_vector(input_arcs_number1 downto 0);
    priority_authorizations : in std_logic_vector(input_arcs_number1 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_value1))
97                      and (s_time_counter < time_B_value)
98                      and (time_A_value /= 0)
99                      and (time_B_value /= 0) )
100
101                or ( (s_reinit_time_counter = '0')
102                     and (time_A_value /= 0)
103                     and ( ((transition_type = transition_t(TEMPORAL_A_A))
104                            and (s_time_counter = (time_A_value1)))
105                     or ( (transition_type = transition_t(TEMPORAL_A_INFINITE))
106                          and (s_time_counter >= (time_A_value1)) )
107                )
108
109                or ( (transition_type /= transition_t(NOT_TEMPORAL))
110                     and (s_reinit_time_counter = '1')
111                     and (time_A_value = 1) )
112                )
113     ) then
114       s_firing_condition <= '1';
115     else
116       s_firing_condition <= '0';
117     end if;
118   end process firing_condition_evaluation;
119
120   priority_authorization_evaluation : process(priority_authorizations)
121     variable v_priority_combination : std_logic;
122   begin
123     v_priority_combination := '1';
124
```

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

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

Bibliography

[1] 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).

[2] 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).

[3] 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).

[4] "IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164)". In: *IEEE Std 1164-1993* (May 1993), pp. 1–24. DOI: 10.1109/IEEESTD.1993.115571.

[5] 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).