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

En Informatique

École doctorale : Information, Structures, Systèmes

Unité de recherche LIRMM

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

Présenté par Vincent IAMPIETRO
Le Date de la soutenance

Sous la direction de David Delahaye
et David Andreu

Devant le jury composé de

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

UNIVERSITÉ DE MONTPELLIER

Acknowledgements

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

Contents

List of Figures

List of Tables

List of Abbreviations

SITPN	**S**ynchronously executed **I**nterpreted **T**ime **P**etri **N**et with priorities
VHDL	**V**ery high speed integrated circuit **H**ardware **D**escription **L**anguage
CIS	**C**omponent **I**nstantiation **S**tatement
PCI	**P**lace **C**omponent **I**nstance
TCI	**T**ransition **C**omponent **I**nstance
GPL	**G**eneric **P**rogramming **L**anguage
HDL	**H**ardware **D**escription **L**anguage
LRM	**L**anguage **R**eference **M**anual
DSL	**D**omain **S**pecific **L**anguage
MDE	**M**odel-Driven **E**ngineering

For/Dedicated to/To my…

Chapter 1

The HILECOP model-to-text transformation

The aim of this chapter is to present the details of the HILECOP model-to-text transformation that we propose to verify as semantic preserving. The chapter is structured as follows. First, we make an overall description of the HILECOP transformation. Then, we present, in Section 1.2, a literature review of the works pertaining to transformation functions in the context of formal verification. The literature review focuses on the expression of transformation functions and on their implementation. In Section 1.3, we thoroughly present the HILECOP transformation function in the form of a pseudo-code algorithm. Finally, in Section 1.4, we describe the Coq implementation of the algorithm. Note that, in the following chapter, we refer to the generic constant, internal signal and port identifiers defined in the `place` and `transition` designs through their abbreviated names (see Table **??**).

1.1 Informal presentation of the HILECOP transformation

This section outlines the main phases of the HILECOP transformation function. The goal is to give to the reader the means to appreciate the differences and the similarities between the HILECOP transformation and the other transformations presented in the literature review of Section 1.2. Then, Section 1.3 will enter the details of the transformation by presenting the transformation algorithm.

The HILECOP model-to-text transformation function takes an SITPN model as input; then, it generates a top-level $\mathcal{H}$-VHDL design out of the input model. We will illustrate the HILECOP model-to-text transformation on the input SITPN model presented in Figure 1.1.

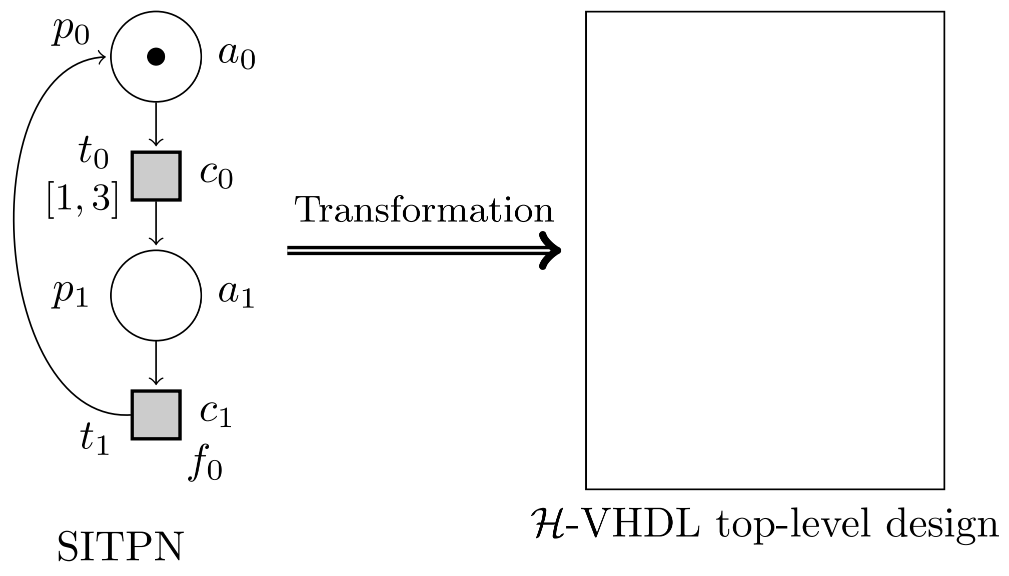

FIGURE 1.1: Transformation of an input SITPN model into a top-level $\mathcal{H}$-VHDL design. The input model is composed of two places, p_0 and p_1, and two transitions, t_0 and t_1. The transition t_0 is associated with the time interval $[1, 3]$ and the condition c_0. The transition t_1 is associated with the condition c_1, and its firing triggers the execution of the function f_0. The action a_0 is activated when the place p_0 is marked, and the action a_1 is activated when the place p_1 is marked.

The generated top-level design implements the structure of the input SITPN. As a first step, the transformation generates, for each place of the input SITPN, a component instance of the `place` design, and, for each transition of the input SITPN, a component instance of the `transition` design. These subcomponents constitute the main part of the $\mathcal{H}$-VHDL top-level design's behavior. Figure 1.2 shows the behavior of the top-level design resulting of this first generation step.

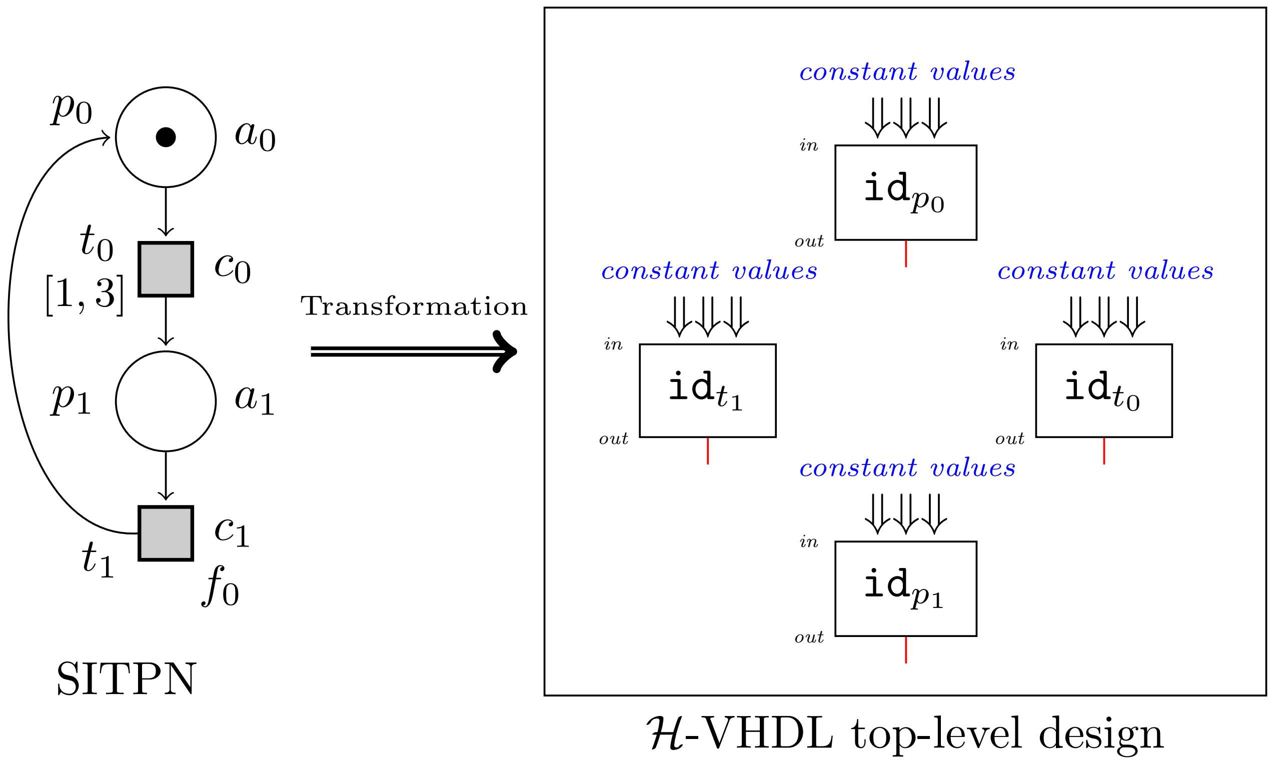

FIGURE 1.2: Generation of the `place` and `transition` component instances based on the set of places and transitions of the input SITPN. The `PCI` $\mathtt{id}_{p_0}$ implements the place p_0, TCI $\mathtt{id}_{t_0}$ the transition t_0... In red, the internal signals connected to the `marked` port of PCIs and to the `fired` port of TCIs.

During this step, each PCI and TCI receive a value for each of their generic constants through the creation of generic maps. In the generic map of a TCI id_t (implementing a transition t), the `ian` constant is associated with the number of input arcs of t, the `cn` constant with the number of conditions attached to t, etc. In the generic map of a PCI id_p, the `ian` constant is associated with the number of input arcs of p, the `oan` constant with the number of output arcs of p, and the `mm` constant with the maximal marking value of p. The maximal marking value associated with a given place p of the input SITPN is an information passed as a parameter to the transformation function. This information comes from the analysis of the input SITPN pertaining to the *boundedness* of the input model. In the definition of the HILECOP methodology, this analysis takes place before the transformation of the input SITPN into a $\mathcal{H}$-VHDL design.

During the first transformation step, illustrated in Figure 1.2, the input and output port maps of PCIs and TCIs are also partly generated. In the manner of the generic constants in generic maps, some input ports are associated with constant values in the input port maps of PCIs and TCIs. All these associations are generated during this first step. Also, the `marked` output port of every PCI is associated with an internal signal in the output port map of the PCI. The internal signal will be connected later in the course of the transformation. The same holds for the `fired` output port of every TCI.

After this first step, the component instances are interconnected through their port interfaces. Figure 1.3 illustrates the behavior of the top-level design after the interconnection of

PCIs and TCIs.

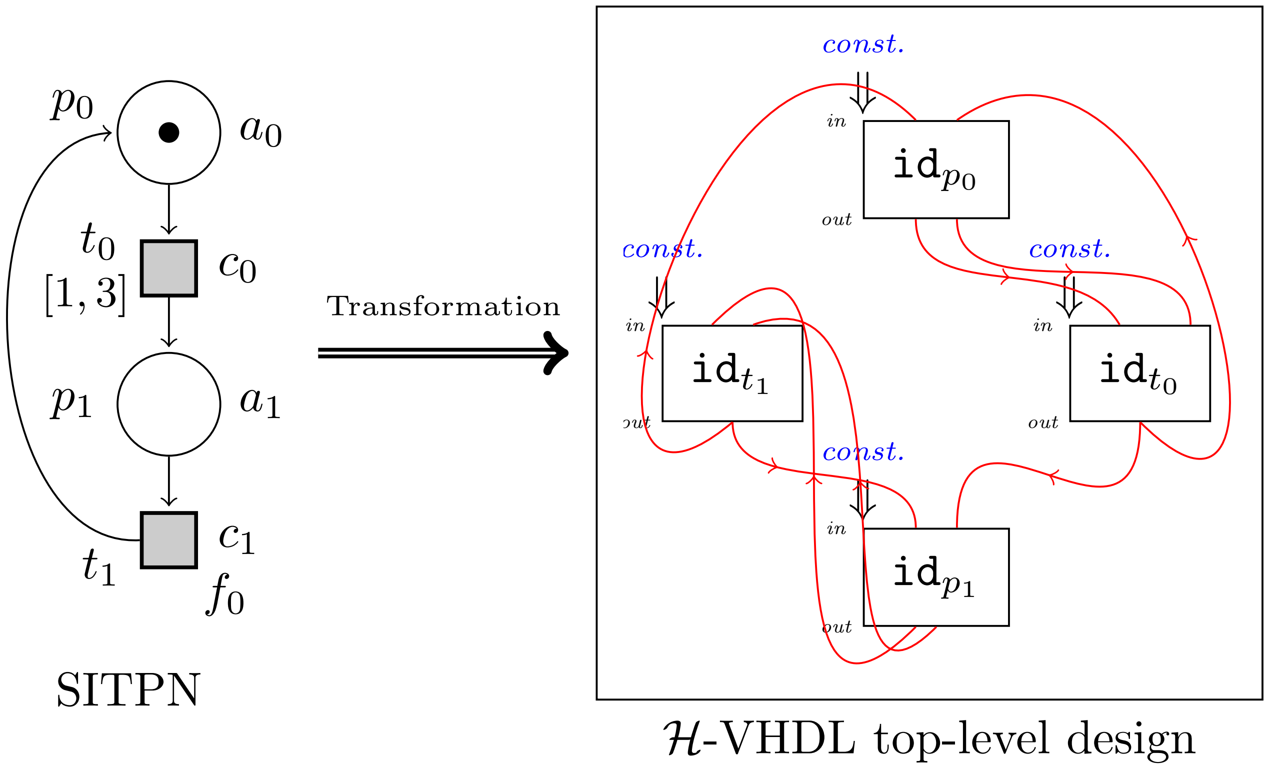

FIGURE 1.3: Generation of the interconnections between the `place` and `transition` component instances. In red, the internal signals interconnecting the PCIs and the TCIs. These signals are generated by the transformation. The arrows indicate the sense of propagation of the information. In blue, the constant associations (i.e. the generic maps and a part of the input port maps) produced during the previous transformation step.

The PCIs and TCIs interact through their interfaces to exchange informations. For instance, a PCI id_p, implementing a given place p, informs its output TCIs (i.e. the TCIs implementing the output transitions of p) that its current marking enables them. The marking of a PCI is represented by the value of its internal signal `s_marking`. A PCI is the only one to have access to the current value of its internal signals. Thus, a PCI must communicate to its output TCIs their sensitization status. To perform this exchange of information, the transformation generates an internal signal to connect a specific output port of a PCI (the `oav` port) to a specific input port of the output TCIs (the `iav` port). Likewise, a TCI informs its input and output PCIs about its firing status. The transformation generates an internal signal to connect the `fired` output port of a TCI to the `itf` and `otf` input ports of the input and output PCIs. These interconnections are performed by adding new associations in the input port map and output port map of PCIs and TCIs. Through the execution of the internal behavior of each PCI and TCI, and, through the interconnection of component instances, the transformation aims at generating a design's behavior that, by its very structure, carries the rules of the SITPN semantics and conforms to the execution of the input SITPN model. To reduce the size of circuits after the synthesis on an FPGA card, PCIs and TCIs only communicate with Boolean signals through their interfaces.

The last part of the transformation deals with the interpretation elements of the input SITPN, i.e. the conditions, the actions and the functions. Each condition of the input SITPN leads to the declaration of a Boolean input port in the port clause of the top-level design. Then, in the design's behavior, each input port representing a condition is connected to the `ic` input port of TCIs. The interconnection of an input port of the top-level design to the `ic` input port of a TCI reflects an existing association between a transition and a condition of the input SITPN model. For each action and function of the input SITPN, the transformation generates a Boolean output port, a.k.a. an *action* or a *function* port. At runtime, the value of these output ports represent the activation or execution status of the corresponding actions and functions. To determine the value of the action and function ports, the transformation generates two processes: the `action` process and the `function` process. The `action` process is a synchronous process responding to the falling edge of the clock signal. At the occurrence of the falling edge of the clock signal, the `action` process sets the value of the *action* ports computed from the values of the `marked` output ports. The `marked` port is an output port of the `place` design. Through the `marked` port, the PCIs inform the outside about their marking status, i.e. if they possess at least one token or not. Remember that the transformation generated an association between the `marked` output port and an internal signal in the output port map of PCIs during the first transformation step. These internal signals are read by the `action` process to assign a value to the *action* ports. The `function` process is a synchronous process responding to the rising edge of the clock signal. At the occurrence of the rising edge of the clock signal, the `function` process sets the value of the *function* ports computed from the values of the `fired` output ports. The `fired` port is an output port of the `transition` design. Through the `fired` port, the TCIs inform the outside about their firing status, i.e. if they are fired or not. Remember that, during the first transformation step, the transformation generated an association between the `fired` output port and an internal signal in the output port map of TCIs. These internal signals are read by the `function` process to assign a value to the *function* ports. Figure 1.4 presents the top-level $\mathcal{H}$-VHDL design at the end of the transformation.

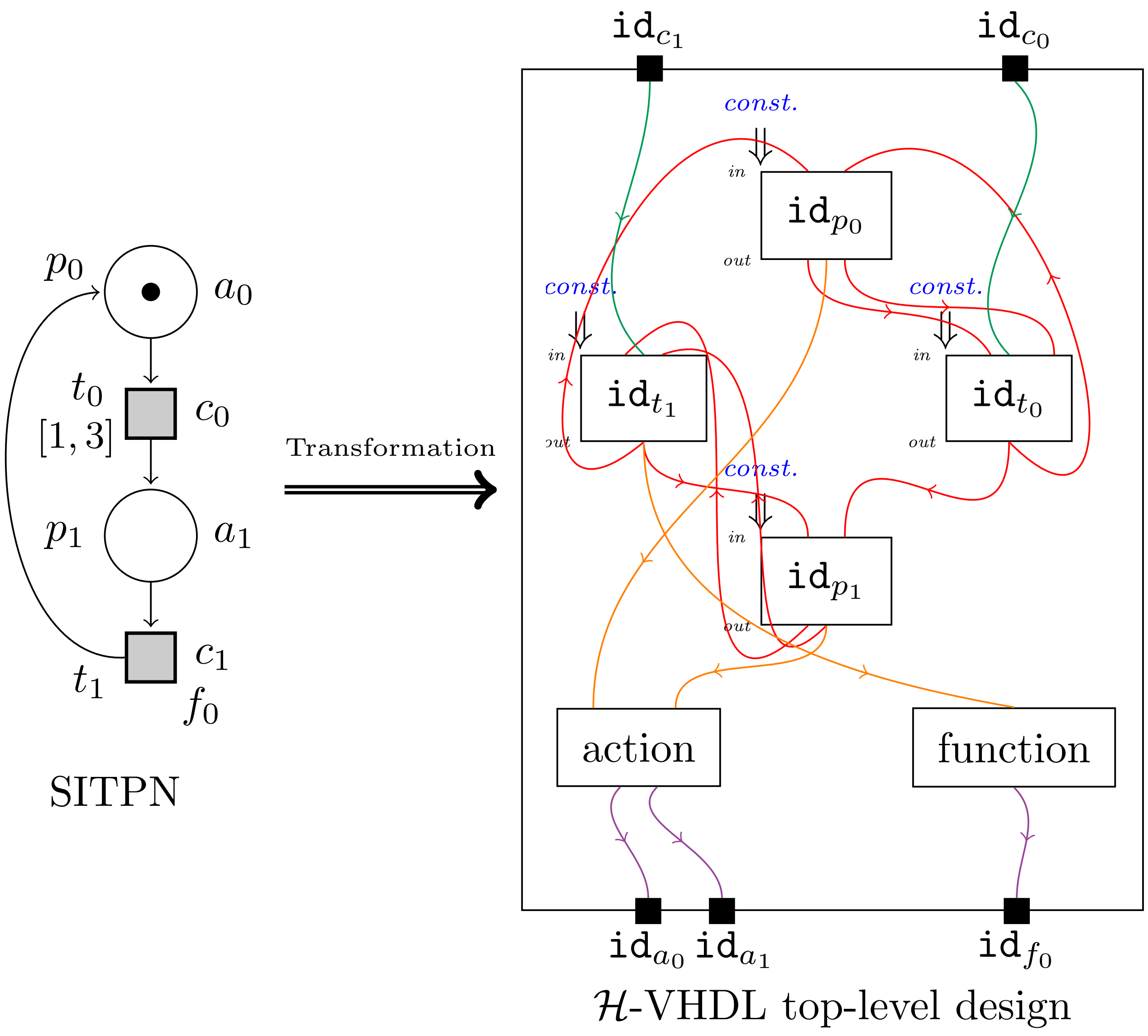

FIGURE 1.4: Generation of the input and output ports, and of the `action` and the `function` process in the $\mathcal{H}$-VHDL top-level design. The *primary* input port $\mathtt{id}_{c_1}$ (resp. $\mathtt{id}_{c_0}$) implements the condition c_1 (resp. c_0). In green, the internal signals, generated by the transformation, connecting the input ports of the top-level design to the `input_conditions` input port of TCIs. The $\mathtt{id}_{a_0}$ and $\mathtt{id}_{a_1}$ output ports reflect the activation status of the actions a_0 and a_1. The $\mathtt{id}_{f_0}$ output port reflects the activation status of the function f_0. In orange, the internal signals, generated by the transformation, connecting the `marked` and `fired` output ports of PCIs and TCIs to the `action` and `function` processes. In purple, the representation of the assignments performed by the `action` and `function` processes and that set the value of the action and function ports.

1.2 Expressing transformation functions

In this section, we present our literature review pertaining to transformation functions in the context of formal verification. Here, a transformation function is understood as any kind of mapping from a source representation to a target representation, where the source and target representations possess a behavior of their own (i.e. they are executable). We use the same articles to perform our literature review in this chapter and in the following chapter, i.e. Chapter **??**. However, our research questions, i.e. the questions we try to give an answer to while reading the chosen articles, and our presentation axis differ from one chapter to the other. Here, the following questions guide our reading:

- Is there a proper way to build a transformation function? Are there standards depending on the application domain?
- How can we build a modular, extensible transformation function?
- How can we build a transformation function that will ease the proof of semantic preservation?

The goal is to inspire ourselves with the works of the literature, and to see how far the correspondence holds between our specific case of transformation, and other cases of transformations. The material we used for the literature review is divided in three categories. Each category covers a specific case of transformation function. The three categories are:

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

Note that, in the case of compilers for programming languages, the term *translation* is preferred over transformation to talk about the generation of a target program from a source program.

1.2.1 Building transformation functions

As the authors state in [15], "Although theoretically possible, verifying a compiler that is not designed for verification would be a prohibitive amount of work in practice." The question is to know how to design such a compiler? How to anticipate the fact that we will have to prove that the compiler is semantic preserving? Now, let us consider these questions in the more general context of transformation functions that map a source representation to a target one.

Compilers for generic programming languages

In the context of formally verified compilers for generic programming languages, the translation from a source program to a target program is straight forward. While descending recursively through the AST of the input program, each construct of the source language is mapped to one or many constructs of the target language. Figure 1.5 gives an example of the translation from Java program expressions to Java bytecode expressions, set in the context of a compiler for Java programs written within the Isabelle/HOL theorem prover [14]. Here, the mapping between source and target constructs is clearly defined.

```
mkExpr jmb (NewC c) = [New c]
mkExpr jmb (Cast c e) = mkExpr jmb e @ [Checkcast c]
mkExpr jmb (Lit val) = [LitPush val]
mkExpr jmb (BinOp bo e1 e2) = mkExpr jmb e1 @ mkExpr jmb e2 @
   (case bo of
       Eq => [Ifcmpeq 3,LitPush(Bool False),Goto 2,LitPush(Bool True)]
     | Add => [IAdd])
mkExpr jmb (LAcc vn) = [Load (index jmb vn)]
mkExpr jmb (vn::=e) = mkExpr jmb e @ [Dup , Store (index jmb vn)]
mkExpr jmb ( cne..fn ) = mkExpr jmb e @ [Getfield fn cn]
mkExpr jmb (FAss cn e1 fn e2 ) =
   mkExpr jmb e1 @ mkExpr jmb e2 @ [Dup_x1 , Putfield fn cn]
mkExpr jmb (Call cn e1 mn X ps) =
   mkExpr jmb e1 @ mkExprs jmb ps @ [Invoke cn mn X]
mkExprs jmb [] = []
mkExprs jmb (e#es) = mkExpr jmb e @ mkExprs jmb es
```

FIGURE 1.5: Translation from Java expressions to Java bytecode expressions

In the works pertaining to the well-known CompCert project [11, 2], the many passes building the compiler from C programs to assembly languages are also clearly mapping each construct of source program to target program constructs. Moreover, the pattern matching possibilities offered by languages like Coq, Isabelle, HOL and other interactive theorem provers enable a clear and concise implementation of compilers.

The cases of optimizing compilers like [11] and [16] show that, to avoid to write too complex functions when passing from a source to a target program, the compilation is decomposed into many passes. No more than 12 passes for the CakeML compiler, and up to 7 passes for CompCert. This is a way to keep the translation functions simple enough in order to ease reasoning afterwards. Indeed, the more the gap is important between the source representation and the target one, the more the translation function will be complex.

Another point that is noticeable while expressing a translation function is the necessity to keep a binding between the source and the target representations. For instance, in CompCert, when passing from transformed C programs to an RTL representation (based on registers and control flow graphs), a binding function γ links the variables of a C program to the registers generated in the RTL representation of the program. The binding is necessary for both the translation and the proof of semantic preservation. During the translation, it permits to replace the variables by their corresponding registers in the RTL code. During the proof of semantic preservation, the link that exists between a variable and a register indicates which elements must be compared to prove that the execution state of the source representation is similar to the execution state of the target representation. The generation of this binding function must be integrated to the design of the translation function.

In [11], and [7], compilers are written within the Coq proof assistant. Compilers are expressed using the state-and-error monad, thus mimicking the traits of imperative languages into a functional programming language setting. In Section 1.3, we present the HILECOP transformation in the form of an imperative pseudo-code algorithm. The state-and-error monad is well-suited to the implementation of this kind of algorithm with a functional language like Coq; thus, we chose to apply this monad to our implementation of the tranformation algorithm (see Section 1.4).

Compilers for hardware description languages

The other category of compilers that we are interested in are compilers for hardware description languages (HDL). The HILECOP methodology's goal is the design of harware circuits. For that reason, we are interested in studying the case of compilers for HDLs. However, one should notice that compiling a HDL program into a lower level representation is one level of abstraction down compared to the transformation we propose to verify. Indeed, it corresponds to Step 3 in the HILECOP methodology, i.e. the transformation of VHDL source code into a RTL representation.

In the context of formal verification applied to HDLs compilers, only a few works describe the specificities of their translation function.

In [5], the authors define the FeSi language (a refinement of the BlueSpec language, a specification language for hardware circuit behaviors), and its embedding in Coq. The authors present the syntax and semantics of the FeSi language and of the RTL language which is the target language of the compiler. FeSi programs are composed of simple expressions, and actions permitting to read or write from different types of memory (registers). Therefore, the abstract syntax is divided into the definition of expressions and the definition of actions, i.e: control flow instructions and operations on memory. The RTL language is composed of expressions and write operations to registers. The authors are more interested in proving that a FeSi specification is well-implemented by a given Coq program, than giving the details of the translation from FeSi to RTL. However, the translation seems straight-forward, and proceeds as usual by descending through the AST of FeSi programs.

In [3], the authors present a compiler for the language Koîka, which is also a simpler version of the BlueSpec language. A Koîka program is composed of a list of rules; each rule describes actions that must be performed atomically. Actions are read and write operations on registers. A Koîka program is accompanied by a scheduler that specify an execution order for the rules. The described compiler transforms Koîka programs into RTL descriptions of hardware circuits. The translation function builds an RTL circuit by descending recursively down the AST of rules. Each action is translated into a specific RTL representation which are afterwards composed together to get complex circuits. The translation becomes trickier when it comes to decide the composition of RTL circuits to respect the execution order prescribed by the scheduler.

In [4], the authors present the verification of a compiler toolchain from Lustre programs to an imperative language (Obc), and from Obc to Clight. The Clight target is the one defined in

CompCert[11]. Lustre permits the definition of programs composed of nodes that are executed synchronously. Nodes treat input streams and yield output streams of values. A node body is composed of sequence of equations that determine the values of output streams based on the input. Obc programs are composed of class declarations. A class declaration has a vector of memory variables, a vector of instances of other classes, and method declarations. The translation turns each node of a Lustre program into a class of Obc accompanied by two methods: the reset method, for the initialization of the streams, and the step method, for the update of values resulting of a synchronous step.

In [12], the authors describe a compiler that transforms Verilog programs into netlists targetting certain FPGA models. Verilog programs are a lot like VHDL programs; they describe a hardware circuit behavior in terms of processes. A netlist is composed of registers, variables and a list of cells corresponding to combinational components. During the translation process, the expressions of the Verilog programs are turned into netlist cells, and the composition of statements leads to the creation of complex circuits by means of cell composition.

Model transformations

We are now presenting the works pertaining to model-to-model and model-to-text transformations in the context of formal verification. Because of the very nature of the transformation we propose to verify, i.e a model-to-text transformation, the following works are of particular interest to us. We will focus here on the manner to express transformations in the case of model-to-model and model-to-text transformations. Also, we tried to find articles related to model transformations involving Petri nets.

In [1], the authors observe that Model-Driven Engineering (MDE) is all about model transformation operations. They propose to set a formal context within the Coq proof assistant to verify that model transformations preserve the structure of the source models into the target models. To illustrate their methodology, they choose to transform UML state machine diagrams into Petri net models. The translation rules from source to target models are expressed within the setting of the OMG standard QVT language (Query/View/Transform). The QVT language offers a formal way to express model transformations, partly based on the Object Constraint Language (OCL). The translation rules maps the different kind of structures that can be found in UML state diagrams to specific structures of Petri nets. Even though the two models used as source and target of transformations are executable, the authors leverage the formal context provided by Coq to prove that the expressed transformations preserve certain structural properties.

In [6], the authors describe a process for model transformation where transformation rules are expressed with the Atlas Transformation Language (ATL). Transformation rules in ATL involve both declarative (OCL) and imperative (match rules) instructions. The authors show how the ATL rules can easily be translated into Coq relations. An example is given on the kind of model-to-model transformations that can be implemented that way. The example is a UML class diagram to relational database model transformation.

In [8], the authors explore the different ways to give a formal semantics to a Domain-Specific Language (DSL) in the context of MDE. Here, the syntax of a given DSL is expressed with a meta-model. An instantiation of this meta-model (a model) yields a DSL program. The authors specify a transformation from a DSL model to another executable model, thus providing an *translational* semantics to the DSL model. The authors illustrate their approach with a source DSL named xSPEM, which is a process description language. The target models are timed PNs. The transformation is expressed through a structural mapping; i.e, each element of an xSPEM model is mapped to a particular PN: an activity is mapped to a subnet, a resource to a single place, connection from activity to resource through parameter is mapped to a connection of transitions and places in the resulting PN...

In [9], the authors address the problem of expressing model transformations by using transformation graphs. Precisely, the kind of transformation graphs that are used are called Triple Graph Grammar (TGG). A TGG is a triplet $<s, c, t>$ where the "correspondence model c explicitly stores correspondence relationships between source model s and target model t".

The work described is [10] is really close to our own verification task. The article describes how Coloured Petri Nets (CPNs, specifically LLVM-labelled Petri nets) are transformed into LLVM programs representing the state space (the graph of reachable markings) of these PNs. The aim is to enable an efficient model-checking of the CPNs. LLVM-labelled PNs are CPNs whose places, transitions and arcs have LLVM constructs for colour domains. Places are labelled with data types. Transitions are labelled with boolean expressions that correspond to the guard of the transition. Arcs are labelled by multisets of expressions. A marking is a function that maps each place to a multiset of values belonging to the place's type. The authors define data structures (multisets, sets, markings,...) with interfaces, i.e. sets of operations over structures, to represent the Petri nets in LLVM. They define interpretation functions that draw equivalences between Petri nets objects and LLVM data structures. The authors define two algorithms: `fire_t` and `succ_t` to compute the graph of reachable states. These are the functions that transform CPNs into concrete LLVM programs.

In [13], the author describes a transformation from UML state machine diagrams to Coloured Petri Nets (CPNs). The aim is to leverage the means of analysis provided by Petri nets to certify certain properties over UML state machine diagrams. The authors want to verify that the transformation preserve structural properties between source and target models. The transformation function does not use a standard setting as QVT or ATL, or transformation graphs. It is expressed as a specific function written in Isabelle/HOL.

In [18], the author presents a transformation from Architecture Analysis and Design Language (AADL) models to Timed Abstract State Machines (TASMs). AADL is a language widely used in avionics to describe both hardware and software systems. AADL doesn't have a lot of tools to analyze and simulate the designed systems; therefore transforming AADL models into TASM enables the use of an important toolbox for analysis, and simulation. The transformation from AADL to TASMs are described with ATL rules.

Discussions on how to build transformation functions in the context of semantic preservation

Transformation functions are mappings from a source representation to a target representation. The more the mapping from source to target is straight-forward the easier the comparison will be when proving that the transformation is semantic preserving. Thus, in [11, 16, 7] where complex case of optimizing compilers are presented, the compilation is split into many simple pass to ease the verification effort coming afterwards. In the case of the HILECOP transformation, we are not yet concerned with the optimization of the generated VHDL code. Thus, our transformation algorithm performs the generation of the target $\mathcal{H}$-VHDL design in a single pass. We do not need to use intermediary representations between the input SITPN model and the generated $\mathcal{H}$-VHDL design.

Also, while transforming source programs, the compiler must often generate fresh constructs belonging to the target language (for instance, generating an fresh RTL register for each variable referenced in a source C program in [11]). The compiler must keep a binding, that is, a memory of the mapping between the elements of the source program and their mirror in the target program. This consideration is of interest in our case of transformation where the elements of SITPNs are also mirrored by elements in the generated $\mathcal{H}$-VHDL design.

It remains hard to establish a standard way to express a transformation function as it really depends on the form of the input and the output representation. Compilers for programming languages tend to be a lot more compositional than model transformations. Here, the word *compositional* means that the translation rules can be split into simple and independent cases of translation, e.g translation of expressions, then translation of statements, then translation of function bodies,... This is a huge advantage to perform the proof of semantic preservation. Indeed, this decomposition of a translation function permits to reason on simple translation cases; yet, each of these translations cases yields a piece of target code that can be executed or interpreted in an independent manner. In the case of the HILECOP, we tried as much as possible to express the transformation in a compositional way. First, we tried to devise the transformation by building up transformation functions for each element of the SITPN structure, i.e.: a transformation function for the places, another for the transitions... However, due to the interconnections that exist between the component instances of the generated $\mathcal{H}$-VHDL design, it is impossible to define transformation functions that would yield stand-alone executable code.

In the world of models, there exist some standard formalisms to express transformation rules (QVT, ATL, transformation graphs...). However, the complexity of the transformation rules depends on the richness of the elements composing the source model, and the distance to the concepts of the target model. In our case, we could not see what would the perks of using such formalisms as QVT or ATL to devise our transformation.

1.3 The transformation algorithm

Before detailling the algorithm underlying the HILECOP model-to-text transformation, we want to explain why this transformation must be automatized. Judging by the appearance of the $\mathcal{H}$-VHDL design generated from the input SITPN model, the reader could rightly ask why the designers of hardware circuits that are using the HILECOP methodology do not start directly by writing down the VHDL code. The reasons are many. First, handling the interconnections between PCIs and TCIs is simple enough when the number of places and transitions of the input SITPN is few, however, it becomes a lot more tedious with the increase of the size of models. To give an example, the Neurrinov company[1], which follows the HILECOP methodology to design critical digital circuits, has developed a digital circuit model for the control of the

[1] https://neurinnov.com/

electro-stimulation in neuroprotheses. Once flattened down, the model is composed of 1097 places and 1666 transitions. The top-level VHDL design generated from this model represents up to 140000 lines of code. Obviously, the hand-coding of this input model into a VHDL design would be too error-prone. Moreover, the PN models offer a lot of opportunities in terms of analysis and model-checking compared to the ones that exist for VHDL code. Finally, the graphical aspect of PNs appears to be more fit for the task of circuit design in comparison to plain source code, as it facilitates the discussions between designers. For these reasons, we choose to preserve SITPNs as the input models of the HILECOP methodology, and to automatize the transformation into top-level $\mathcal{H}$-VHDL designs.

In this section, we give the algorithm underlying the HILECOP model-to-text transformation. This algorithm is the base of the Coq implementation of the HILECOP transformation; the implementation is presented in Section 1.4. As stated in Chapter **??**, there exists a Java implementation of the HILECOP methodology. This implementation performs the generation of VHDL code from a SITPN model. However, the algorithm of the transformation has never been devised, nor a formal specification given. The following algorithm is one of the contribution of this thesis. It has been devised through the examination of the code of the existing Java implementation, and through the discussions with the designers of the HILECOP methodology.

1.3.1 The `sitpn_to_hvhdl` function

The HILECOP transformation algorithm, presented in Algorithm 1, generates a $\mathcal{H}$-VHDL design and a SITPN-to-$\mathcal{H}$-VHDL binder from an input SITPN. A SITPN-to-$\mathcal{H}$-VHDL design binder is a structure that binds the elements of an SITPN (places, transitions, actions...) to the elements of a $\mathcal{H}$-VHDL design (component instances or signals). Such a binder is generated alongside the transformation and links a SITPN element to its $\mathcal{H}$-VHDL *implementation*, i.e. the $\mathcal{H}$-VHDL element that will supposedly behave similarly to the source SITPN element at runtime. Thus, the SITPN-to-$\mathcal{H}$-VHDL design binder is at the center of the state similarity relation, presented in Chapter **??**, and that enables the comparison between an SITPN state and an $\mathcal{H}$-VHDL design state. The formal definition of an SITPN-to-$\mathcal{H}$-VHDL design binder is as follows.

Definition 1 (SITPN-to-$\mathcal{H}$-VHDL design binder)**.** *Given a $sitpn \in SITPN$ and a $\mathcal{H}$-VHDL design $d \in design$, a SITPN-to-$\mathcal{H}$-VHDL design binder $\gamma \in WM(sitpn, d)$ is a tuple $<PMap, TMap, CMap, AMap, FMap>$ where:*

- $sitpn = <P, T, pre, test, inhib, post, M_0, \succ, \mathcal{A}, \mathcal{C}, \mathcal{F}, \mathbb{A}, \mathbb{C}, \mathbb{F}, I_s>$
- $d = \texttt{design}\ id_e\ id_a\ gens\ ports\ sigs\ cs$
- $PMap \in P \rightarrow \{id \mid \texttt{comp}(id, \texttt{place}, g, i, o) \in cs\}$
- $TMap \in T \rightarrow \{id \mid \texttt{comp}(id, \texttt{transition}, g, i, o) \in cs\}$
- $CMap \in \mathcal{C} \rightarrow \{id \mid (\texttt{in}, id, t) \in ports\}$
- $AMap \in \mathcal{A} \rightarrow \{id \mid (\texttt{out}, id, t) \in ports\}$
- $FMap \in \mathcal{F} \rightarrow \{id \mid (\texttt{out}, id, t) \in ports\}$

As presented in Definition 1, the binder is composed of five sub-environments that map the different SITPN sets to identifiers. The $PMap$ and $TMap$ sub-environments map the places to their corresponding PCI identifiers, and the transitions to their corresponding TCI identifiers. The $CMap$ sub-environment maps the conditions to input port identifiers. The $AMap$ and $FMap$ sub-environments map the actions and functions to output port identifiers. In what follows, for a given binder γ and an element of an SITPN structure $e \in P \sqcup T \sqcup \mathcal{C} \sqcup \mathcal{A} \sqcup \mathcal{F}$, we write $\gamma(e)$ where e is looked up in the appropriate function. For instance, for a given $f \in \mathcal{F}$, $\gamma(f)$ is a shorthand for $FMap(f)$ where $\gamma = <\ldots, FMap>$.

Algorithm 1 is the algorithm of the HILECOP model-to-text transformation. The algorithm as four parameters; the first one is the input SITPN model; id_e and id_a are the entity and the architecture identifiers for the generated $\mathcal{H}$-VHDL design; $mmf \in P \to \mathbb{N}$ is the function associating a maximal marking value to each place of the input SITPN. This function is the result of the analysis of the input SITPN.

Remark 1 (Bounded SITPN). *A part of the analysis is interested in determining the maximal number of tokens that a place can hold during the execution of a SITPN. If each place of the SITPN can only hold a limited number of tokens during the execution, then the model is said to be* bounded. *In that case, it is possible to compute a function that associates the places of the SITPN with a maximal marking value. Thus, the presence of the mmf function as a parameter of the `sitpn_to_hvhdl` function implies that the input SITPN model is bounded. In the case of an unbounded input model, there exists a place that can accumulate an infinite number of tokens during the model execution. In the world of hardware description, and especially when aiming at hardware synthesis, every element must have a finite dimension. In the definition of the `place` design, the internal signal `s_marking` represents the marking value of a place. The maximal value of the `s_marking` signal is bounded by the generic constant `maximal_marking`. Thus, when generating, a PCI from a place in the course of the transformation, we must be able to give a value to the `maximal_marking` generic constant. However, even with a settled `maximal_marking` value, the execution of a $\mathcal{H}$-VHDL design, resulting from the transformation of an unbounded SITPN model, could lead to the overflow of the value of the `s_marking` signals in the internal states of PCIs. Thus, it is impossible to prove the equivalence between the behavior of an unbounded SITPN model and its corresponding $\mathcal{H}$-VHDL design.*

Algorithm 1: `sitpn_to_hvhdl`($sitpn$, id_e, id_a, mmf)

```
1 $d \leftarrow$ design $id_e$ $id_a$ $\emptyset$ $\emptyset$ $\emptyset$ null
2 $\gamma \leftarrow \emptyset$
3 generate_architecture($sitpn, d, \gamma, mmf$)
4 generate_interconnections($sitpn, d, \gamma$)
5 generate_ports($sitpn, d, \gamma$)
6 return ($d,\gamma$)
```

In Algorithm 1, Line 1 creates the initial $\mathcal{H}$-VHDL design structure and assigns it to the variable d. Initially, the design has an empty port declaration set, an empty internal signal declaration set, and a behavior defined by the `null` statement. The design generated by the `sitpn_to_hvhdl` function has an empty set of generic constants; this set stays empty even at the end of the transformation. Line 2 initializes the γ binder with empty sub-environments. From Lines 3 to 5, the called procedures modify the design and the binder structures. Each part of the sequence corresponds to one step of the transformation, which were outlined in Section 1.1. The content of the `generate_architecture` function is detailled in Algorithms 3, 4 and 5. The content of the `generate_interconnections` function is detailled in Algorithm 8. The content of the `generate_ports` function is detailled in Algorithms 9, 10, 11 and 12.

1.3.2 Primitive functions and sets

The description of further functions and algorithms appeals to some primitive functions and set definitions that we introduce here. Below are all the sets that we use in the description of the algorithms.

- $\texttt{input}(p) = \{t \mid \exists \omega \ s.t. \ post(t, p) = \omega\}$, the set of input transitions of a place p.
- $\texttt{output}(p) = \{t \mid \exists \omega, a \ s.t. \ pre(p, t) = (\omega, a)\}$, the set of output transitions of a place p.
- $\texttt{acts}(p) = \{a \mid \mathbb{A}(p, a) = \texttt{true}\}$, the set of actions associated with a place p.
- $\texttt{input}(t) = \{p \mid \exists \omega, a \ s.t. \ pre(p, t) = (\omega, a)\}$, the set of input places of a transition t.
- $\texttt{output}(t) = \{p \mid \exists \omega \ s.t. \ post(t, p) = \omega\}$, the set of output places of a transition t.
- $\texttt{conds}(t) = \{c \mid \mathbb{C}(t, c) = 1 \vee \mathbb{C}(t, c) = -1\}$, the set of conditions associated with a transition t.
- $\texttt{trs}(c) = \{t \mid \mathbb{C}(t, c) = 1 \vee \mathbb{C}(t, c) = -1\}$, the set of transitions to which a condition c is associated.
- $\texttt{pls}(a) = \{p \mid \mathbb{A}(p, a) = \texttt{true}\}$, the set of places to which an action a is associated.
- $\texttt{trs}(f) = \{t \mid \mathbb{F}(t, f) = \texttt{true}\}$, the set of transitions to which a function f is associated.

Every set presented above are *unordered*. However, we assume that, every time we iterate over the elements of an unordered set with a **foreach** statement, the iteration respects an arbitrary order. This order is always the same through the multiple calls to **foreach** statements. Of course, the iteration over the elements of an *ordered* set with a **foreach** statement respects natural order of the set.

Now, let us introduce some primitive functions and procedures that we use in the description of the following algorithms.

- $\texttt{output}_c \in P \rightarrow 2^T$. The $\texttt{output}_c$ function takes a place p as input and yields an ordered set of transitions computed as follows:
  1. if all conflicts between the output transitions of p are solved by mutual exclusion, or if the set of conflicting transitions of p is a singleton, then $\texttt{output}_c$ returns an empty set.
  2. otherwise, the function tries to establish a total ordering over the set of conflicting transitions of p w.r.t the firing priority relation:

  - if no such ordering can be established (in that case, the firing priority relation is ill-formed, and the input SITPN is not well-defined), $\texttt{output}_c$ raises an error.
  - otherwise, the function returns the ordered set, with the top-level priority transition at the head.

- $\texttt{output}_{nc} \in P \to 2^T$. The $\texttt{output}_{nc}$ function takes a place p as input and yields an unordered set of transitions computed as follows:

  - If all conflicts between the output transitions of p are solved by mutual exclusion, or if the set of conflicting transitions of p is a singleton, then, the function returns the set of output transitions of p, i.e $\texttt{output}(p)$ as defined above.
  - Otherwise, the function returns the set of output transitions of p connected through a `test` or an `inhib` arc, i.e. $\{t \mid \exists \omega\ s.t.\ pre(p,t) = (\omega, \texttt{test}) \lor pre(p,t) = (\omega, \texttt{inhib})\}$.

- $\texttt{cassoc}(map, id, x)$ where map is either a generic map, an input port map or an output port map, id is an identifier, x is an expression, a name (i.e. an simple or indexed identifier) or the `open` keyword. The `cassoc` procedure adds an association of the form $(id(i), x)$ to the map structure. The index i is computed as follows based on the content of map:

  1. looks up $id(j)$ with $max(j)$ in the formal parts of map
  2. if no such j, adds $(id(0), x)$ in map
  3. if such j, adds $(id(j+1), x)$ in map

  Examples:

  - $\texttt{cassoc}(\ \{(\texttt{s}(0),\texttt{true}),(\texttt{s}(1),\texttt{false})\},\ \texttt{s},\ \texttt{true})$ yields the resulting map $\{(\texttt{s}(0),\texttt{true}),(\texttt{s}(1),\texttt{false}),(\texttt{s}(2),\texttt{true})\}$.
  - $\texttt{cassoc}(\ \{(\texttt{s}(0),\texttt{true}),(\texttt{s}(1),\texttt{false})\},\ \texttt{a},\ \texttt{open})$ yields the resulting map $\{(\texttt{s}(0),\texttt{true}),(\texttt{s}(1),\texttt{false}),(\texttt{a}(0),\texttt{open})\}$.

- $\texttt{get\_comp}(id_c, cstmt)$ where id_c is an identifier, and $cstmt \in \text{cs}$ is a $\mathcal{H}$-VHDL concurrent statement. The `get_comp` function looks up $cstmt$ for a component instantiation statement labelled with id_c as a component instance identifier, and returns the component instantiation statement when found. The `get_comp` function throws an error if no component instantiation statement with identifier id_c exists in $cstmt$, or if there exist multiple component instantiation statements with identifier id_c in $cstmt$.

- $\texttt{put\_comp}(id_c, cistmt, cstmt)$ where id_c is an identifier, $cistmt$ is a component instantiation statement, and $cstmt \in \text{cs}$ is a $\mathcal{H}$-VHDL concurrent statement. The `put_comp` procedure looks up in $cstmt$ for a component instantiation statement with identifier id_c, and replaces the statement with $cistmt$ in $cstmt$. If no CIS with identifier id_c exists in $cstmt$, then $cistmt$ is directly composed with $cstmt$ with the `||` operator. The `put_comp` procedure throws an error if multiple CIS with identifier id_c exist in $cstmt$.

- actual(id, map) where id is an identifier and map is a generic, an input port or an output port map. The actual function returns the actual part associated with the formal part id in map, i.e. returns a if $(id, a) \in map$. The function throws an error if id is not a formal part in map, or if there are multiple association with id as a formal part in map.
- genid(). The genid function returns a fresh and unique identifier. During the transformation, we appeal to it when a new internal signal, a new port or a new component instance must be declared or generated.

Algorithm 2 presents the connect procedure. This procedure takes an output port map o, an input port map i, a name n (i.e. a simple or indexed identifier), an identifier id and a $\mathcal{H}$-VHDL design d as parameters. It generates an internal signal id_s and adds it to the internal signal declaration list of design d. Then, the procedure adds the association between the n name and the internal signal id_s to the output port map o. Moreover, the procedure adds an association between a subelement of id, which element will be determined by the cassoc function, and the internal signal id_s to the input port map i. As the result, n is connected to a subelement of id through the internal signal id_s.

Algorithm 2: connect(o, i, n, id, d)

```
1 $id_s \leftarrow$ genid()
2 $d.sigs \leftarrow d.sigs \cup \{(id_s, \texttt{boolean})\}$
3 $o \leftarrow o \cup \{(n, id_s)\}$
4 cassoc$(i, id, id_s)$
```

Complementary notations

When there is no ambiguity, id_p (resp. id_t) denotes the PCI (resp. TCI) identifier associated with a given place p (resp. transition t) through $\gamma(p) = id_p$ (resp. $\gamma(t) = id_t$), where γ is the binder returned by the transformation function. Similarly, id_c (resp. id_a and id_f) denotes the input port (resp. output port) identifier associated with a given condition c (resp. action a and function f) through $\gamma(c) = id_c$.

1.3.3 Generation of component instances and constant parts

The first step of the transformation generates the PCIs and TCIs, their generic map, and the constant part of their input port maps, in the behavior of the $\mathcal{H}$-VHDL design. At this moment of the transformation, places are bound to PCI identifiers, and transitions are bound to TCI identifiers in the γ binder. Also, the marked output port and the fired output port are connected to internal signals in the output port map of PCIs and TCIs. Algorithm 3 presents the content of the generate_architecture procedure that implements this first part of code generation. The generate_architecture procedure is decomposed in two procedures: the generate_PCIs and the generate_TCIs procedures.

Algorithm 3: generate_architecture$(sitpn, d, \gamma, mmf)$

```
1 generate_PCIs$(sitpn, d, \gamma, mmf)$
2 generate_TCIs$(sitpn, d, \gamma)$
```

The generate_PCIs procedure, presented in Algorithm 4, has four parameters: $sitpn \in SITPN$, the input SITPN model; $d \in design$, the $\mathcal{H}$-VHDL design being generated; $\gamma \in WM(sitpn, d)$, the binder between $sitpn$ and d; $mmf \in P \to \mathbb{N}$, the function assigning a maximal marking value to each place. The procedure iterates over the set of places of the $sitpn$ parameter. For each place p in the set, the procedure produces a corresponding PCI id_p, and generates its generic map g_p, and its partially-built input and output port maps i_p and o_p. At the end of the procedure (Lines 28 to 30), a fresh and unique component identifier id_p is generated, and a new component instantiation statement, corresponding to the instantiation of the PCI id_p, is composed with the current behavior of design d. Finally, the γ binder receives a new couple corresponding to binding of place p to identifier id_p.

Algorithm 4: generate_PCIs($sitpn, d, \gamma, mmf$)

```
1  foreach $p \in P$ do
2      if input($p$) = $\emptyset$ and output($p$) = $\emptyset$ then err("p is an isolated place")
3      $g_p \leftarrow \{(\texttt{mm}, mmf(p))\}; i_p \leftarrow \emptyset; o_p \leftarrow \emptyset$
4      if input($p$) = $\emptyset$ then
5          $g_p \leftarrow g_p \cup \{(\texttt{ian}, 1)\}$
6          $i_p \leftarrow i_p \cup \{(\texttt{iaw}(0), 0), (\texttt{itf}(0), \texttt{false})\}$
7      else
8          $g_p \leftarrow g_p \cup \{(\texttt{ian}, |\texttt{input}(p)|)\}$
9          $i \leftarrow 0$
10         foreach $t \in \texttt{input}(p)$ do
11             $i_p \leftarrow i_p \cup \{(\texttt{iaw}(i), post(t, p))\}$
12             $i \leftarrow i + 1$
13     if output($p$) = $\emptyset$ then
14         $g_p \leftarrow g_p \cup \{(\texttt{oan}, 1)\}$
15         $i_p \leftarrow i_p \cup \{(\texttt{oaw}(0), 0), (\texttt{oat}(0), \texttt{basic}), (\texttt{otf}(0), \texttt{false})\}$
16         $o_p \leftarrow o_p \cup \{(\texttt{oav}, \texttt{open}), (\texttt{pauths}, \texttt{open}), (\texttt{rtt}, \texttt{open})\}$
17     else
18         $i \leftarrow 0$
19         foreach $t \in \texttt{output}_c(p) \cup \texttt{output}_{nc}(p)$ do
20             $(\omega, a) \leftarrow pre(p, t)$
21             $i_p \leftarrow i_p \cup \{(\texttt{oaw}(i), \omega), (\texttt{oat}(i), a)\}$
22             $i \leftarrow i + 1$
23     if acts($p$) = $\emptyset$ then $o_p \leftarrow o_p \cup \{(\texttt{marked}, \texttt{open})\}$
24     else
25         $id_s \leftarrow \texttt{genid}()$
26         $d.sigs \leftarrow d.sigs \cup \{(id_s, \texttt{boolean})\}$
27         $o_p \leftarrow o_p \cup \{(\texttt{marked}, id_s)\}$
28     $id_p \leftarrow \texttt{genid}()$
29     $d.cs \leftarrow d.cs \mid\mid \texttt{comp}(\texttt{id}_p, \texttt{place}, g_p, i_p, o_p)$
30     $\gamma \leftarrow \gamma \cup \{(p, \texttt{id}_p)\}$
```

From Line 2 to Line 27, the procedure generates the generic map, the input port map, and the output port map of the PCI that implements place p. First, the procedure checks if the current place p is isolated, i.e. without input nor output transitions. An error, with an associated message, is raised with the `err` primitive if the test succeeds. The HILECOP transformation raises errors in the presence of an input SITPN model that does not meet the well-definition property (see Definition **??**). One part of the well-definition property pertains to the absence of isolated place in the input model. Line 3 initializes the variables g_p, i_p and o_p, respectively holding the generic map, the input port map and the output port map of the PCI being generated. The generic map g_p initially takes a single association that binds the `mm` constant to the maximal marking value returned by the mmf function for place p. The input port map i_p and the output port map o_p are initialized with empty sets.

Line 4 tests if the set of input transitions of p is empty. If the test succeeds, the `ian` constant is associated to 1 in the generic map g_p. The size of the `iaw` and `itf` input ports, which are of the array type, is equal to the value of the `ian` constant minus one. Thus, in the case where the `ian` constant is associated to 1 in the generic map g_p, the `iaw` and `itf` input ports are composed of one subelement with index 0. At Line 6, the sole subelement of the `iaw` port is associated with 0, and the sole subelement of the `itf` port is associated with `false` in the input port map i_p. If the set of input transitions of p is not empty, the `ian` constant is associated with the size of the set in the generic map g_p. Then, each subelement of the `iaw` port is associated with the weight of the arc between place p and an input transition t. Note that, in that case, the procedure does not deal with the connection of the `itf` port. As the set of input transitions of p is not empty, the connection of the `itf` port will be performed by the `generate_interconnections` described in Algorithm 8.

Line 13 tests if the set of output transitions of p is empty. If the test succeeds, the `oan` constant is associated to 1 in the generic map g_p. The size of the `oaw`, `oat` and `otf` input ports, which are of the array type, is equal to the value of the `oan` constant minus one. Thus, in the case where the `oan` constant is associated to 1 in the generic map g_p, the `oaw`, `oat` and `otf` input ports are composed of one subelement with index 0. At Line 15, the sole subelement of the `oaw` port is associated with 0, the sole subelement of the `oat` port is associated with `basic`, and the sole subelement of the `otf` port is associated with `false` in the input port map i_p. Also, in the abscence of output transitions, the `oav`, `pauths` and `rtt` output ports are left unconnected, i.e. they are associated with the `open` keyword of output port map o_p.

If the set of output transitions of p is not empty, the `oan` constant is associated with the size of this set in the generic map g_p. Then, each subelement of the `oaw` (resp. the `oat`) port is associated with the weight (resp. the type) of the arc between place p and an output transition t. Note that, in that case, the procedure does not handle the connection of the `otf` input port, nor the connection of the `oav`, `pauths` and `rtt` output ports. As the set of output transitions of p is not empty, these connections will be performed by the `generate_interconnections` described in Algorithm 8.

From Line 23 to Line 27, the `generate_PCIs` procedure connects the `marked` output port in the output port map o_p. If the place p is not associated with any action, the `marked` output port is left unconnected, i.e. connected to the `open` keyword. Otherwise, the `marked` output port is connected to a newly generated internal signal of the Boolean type. This generated signal joins the internal signal declaration list of design d. The connection between the `marked` output port and the internal signal will be used later, during the generation of the `action` process (see

Section 1.3.5).

The `generate_TCIs` procedure, presented in Algorithm 5, iterates over the set of transitions T of the $sitpn$ parameter. For each transition t in the set, the procedure produces a corresponding TCI id_t, and generates its generic map g_t, and its partially-built input and output port maps i_t and o_t. At the end of the procedure (Lines 18 to 20), a fresh and unique component identifier id_t is generated, and a new component instantiation statement, corresponding to the instantiation of the TCI id_t, is composed with the current behavior of design d. Finally, the γ binder receives a new couple corresponding to binding of transition t to identifier id_t.

Algorithm 5: `generate_TCIs`$(sitpn, d, \gamma)$

```
1  foreach $t \in T$ do
2      if $\texttt{input}(t) = \emptyset$ and $\texttt{output}(t) = \emptyset$ then $\texttt{err}(\text{"}t\ \textit{is an isolated transition}\text{"})$
3      $g_t \leftarrow \{(\texttt{tt}, \texttt{get\_ttype}(t)), (\texttt{mtc}, \texttt{get\_mtc}(t))\}$
4      $i_t \leftarrow \{(\texttt{A}, \begin{cases} 0 \text{ if } t \notin \texttt{dom}(I_s) \\ lower(I_s(t))\ otherwise \end{cases}), (\texttt{B}, \begin{cases} 0 \text{ if } t \notin \texttt{dom}(I_s) \vee upper(I_s(t)) = \infty \\ upper(I_s(t))\ otherwise \end{cases})\}$
5      $id_s \leftarrow \texttt{genid}()$
6      $d.sigs \leftarrow d.sigs \cup \{(id_s, boolean)\}$
7      $o_t \leftarrow \{(\texttt{fired}, id_s)\}$
8      if $\texttt{input}(t) = \emptyset$ then
9          $g_t \leftarrow g_t \cup \{(\texttt{ian}, 1)\}$
10         $i_t \leftarrow i_t \cup \{(\texttt{iav}(0), \texttt{true}), (\texttt{pauths}(0), \texttt{true}), (\texttt{rt}(0), id_s)\}$
11     else
12         $g_t \leftarrow g_t \cup \{(\texttt{ian}, |\texttt{input}(t)|)\}$
13     if $\texttt{conds}(t) = \emptyset$ then
14         $g_t \leftarrow g_t \cup \{(\texttt{cn}, 1)\}$
15         $i_t \leftarrow i_t \cup \{(\texttt{ic}(0), \texttt{true})\}$
16     else
17         $g_t \leftarrow g_t \cup \{(\texttt{cn}, |\texttt{conds}(t)|)\}$
18     $id_t \leftarrow \texttt{genid}()$
19     $d.cs \leftarrow d.cs\ ||\ \texttt{comp}(\texttt{id}_t, \texttt{transition}, g_t, i_t, o_t)$
20     $\gamma \leftarrow \gamma \cup \{(t, \texttt{id}_t)\}$
```

At Line 2, the procedure checks if transition t is isolated, and raises an error accordingly. Lines 3 to 7 initialize the variables g_t, i_t and o_t, respectively holding the generic map, the input port map and the output port map of the TCI being-generated. The generic map g_t initially takes two associations: the one between the `tt` constant and the result of the function call `get_-ttype`(t), and the one between the `mtc` constant and the result of the function call `get_mtc`(t). The `get_ttype` function returns the type of transition t, i.e. either `NOT_TEMPORAL`, `TEMPORAL_-A_A`, `TEMPORAL_A_B` or `TEMPORAL_A_INFINITE`, based on the form of the time interval associated with t. Algorithm 6 describes the `get_ttype` function.

Algorithm 6: `get_ttype`(t)

```
if $t \notin$ dom($I_s$) then return $NOT\_TEMPORAL$
else if $I_s(t) = [a, a]$ then return $TEMPORAL\_A\_A$
else if $I_s(t) = [a, b]$ then return $TEMPORAL\_A\_B$
else if $I_s(t) = [a, \infty]$ then return $TEMPORAL\_A\_INFINITE$
```

The `get_mtc` function determines the maximal value for the time counter of t based on the form of the time interval associated with transition t. Algorithm 7 describes the `get_mtc` function.

Algorithm 7: `get_mtc`(t)

```
if $t \notin$ dom($I_s$) then return 1
else if $I_s(t) = [a, b]$ then return $b$
else if $I_s(t) = [a, \infty]$ then return $a$
```

In the `generate_TCIs` procedure, Line 4 sets the value of the `A` and `B` input ports while initializing the input port map i_t. The `A` port is associated with 0 if the transition t is not a time transition (i.e. t has no associated time interval, it is not in the domain of function I_s); otherwise, the `A` port is associated with the lower bound of the time interval of t. The `B` input port is associated with 0 if transition t is not a time transition or if its time interval has an infinite upper bound; otherwise, the `B` port is associated with the upper bound of the time interval of t. From Lines 5 to 7, the `generate_TCIs` procedure connects the `fired` output port to a newly generated internal signal in the output port map o_p. This internal signal will then be connected to the input port map of PCIs during the interconnection phase of the transformation (see Section 1.3.4).

Line 8 checks if the set of input places of t is empty. If the test succeeds, the `ian` constant is associated with 1 in the generic map g_t. The size of the `iav`, `pauths` and `rt` input ports, which are of the array type, is equal to the value of the `ian` constant minus one. Thus, in the case where the set of input places of t is empty, the `iav`, `pauths` and `rt` input ports are composed of one subelement with index 0. At Line 10, the sole subelements of the `iav` and the `pauths` ports are associated with `true`, and the sole subelement of the `rt` port is associated with the signal identifier id_s. Remember that the `fired` output port has been previously connected to the internal signal id_s in the output port map o_t. Thus, the `fired` output port is connected to the subelement of the `rt` input port with index 0 through the id_s signal. This connection is mandatory to reset the value of the `s_time_counter` signal (which is an internal signal of the `transition` design) in the abscence of input places. If the set of input places of t is not empty, then the `ian` constant is associated with the size of the set in the generic map g_t.

Line 13 checks if the set of conditions attached to t is empty. If the test succeeds, the `cn` constant is associated with 1 in the generic map g_t. The size of the `ic` input port, which is of the array type, is equal to the value of the `cn` constant minus one. Thus, in the case where the set of conditions attached to t is empty, the `ic` input port is composed of one subelement with index 0. Then, the sole subelement of the `ic` port is associated with `true` in the input port map i_t. If the set of conditions attached to t is not empty, the `cn` constant is associated with the size of the set in the generic map g_t. In that case, the `generate_conds` procedure, presented in Algorithm 10, will handle the connection of the subelements of the `ic` input port.

1.3.4 Interconnection of the place and transition component instances

After the generation of PCIs and TCIs, and of all constant associations in their generic and input port maps, the next step of the transformation performs the interconnections between the interfaces of PCIs and TCIs. The `generate_interconnections` procedure, presented in Algorithm 8, produces these interconnections.

Algorithm 8: `generate_interconnections`$(sitpn, d, \gamma)$

```
1  foreach $p \in P$ do
2      $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \leftarrow \texttt{get\_comp}(\gamma(p), d.cs)$
3      $i \leftarrow 0$
4      foreach $t \in \texttt{input}(p)$ do
5          $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \leftarrow \texttt{get\_comp}(\gamma(t), d.cs)$
6          $i_p \leftarrow i_p \cup \{(\texttt{itf}(i), \texttt{actual}(\texttt{fired}, o_t))\}$
7          $i \leftarrow i + 1$
8      $i \leftarrow 0$
9      foreach $t \in \texttt{output}_c(p)$ do
10         $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \leftarrow \texttt{get\_comp}(\gamma(t), d.cs)$
11         $i_p \leftarrow i_p \cup \{(\texttt{otf}(i), \texttt{actual}(\texttt{fired}, o_t))\}$
12         $\texttt{connect}(o_p, i_t, \texttt{oav}(i), \texttt{iav}, d)$
13         $\texttt{connect}(o_p, i_t, \texttt{rtt}(i), \texttt{rt}, d)$
14         $\texttt{connect}(o_p, i_t, \texttt{pauths}(i), \texttt{pauths}, d)$
15         $\texttt{put\_comp}(id_t, \texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t), d.cs)$
16         $i \leftarrow i + 1$
17     foreach $t \in \texttt{output}_{nc}(p)$ do
18         $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \leftarrow \texttt{get\_comp}(\gamma(t), d.cs)$
19         $i_p \leftarrow i_p \cup \{(\texttt{otf}(i), \texttt{actual}(\texttt{fired}, o_t))\}$
20         $\texttt{connect}(o_p, i_t, \texttt{oav}(i), \texttt{iav}, d)$
21         $\texttt{connect}(o_p, i_t, \texttt{rtt}(i), \texttt{rt}, d)$
22         $id_s \leftarrow \texttt{genid}()$
23         $d.sigs \leftarrow d.sigs \cup (id_s, \texttt{boolean})$
24         $o_p \leftarrow o_p \cup \{(\texttt{pauths}(i), id_s)\}$
25         $\texttt{cassoc}(i_t, \texttt{pauths}, \texttt{true})$
26         $\texttt{put\_comp}(id_t, \texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t), d.cs)$
27         $i \leftarrow i + 1$
28     $\texttt{put\_comp}(id_p, \texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p), d.cs)$
```

The `generate_interconnections` procedure iterates over the set of places of the $sitpn$ parameter. For each place p, the procedure generates the interconnections between the PCI id_p and the TCIs that implement the input and output transitions of p; we will refer to them as the input and output TCIs of PCI id_p.

At Line 2, the `get_comp` function returns the PCI associated with the identifier $\gamma(p)$ (i.e. the PCI identifier associated with place p in γ) by looking up the behavior of the design d. At this step, we assume that all PCIs and TCIs, and all bindings pertaining to places and transitions in the γ binder, have been previously generated by the `generate_architecture` procedure.

Otherwise, the `get_comp` function raises an error if it is not able to find the PCI id_p in the behavior of design d.

Then, from Line 3 to Line 27, the procedure modifies the input and output port map of PCI id_p and the input port map of its input and output TCIs. Finally, Line 28 replaces the old PCI id_p by the modified one in the behavior of design d.

From Line 3 to Line 7, the procedure iterates over the input transitions of place p. Note that the iteration is performed in the same order than the iteration performed by the **foreach** loop at Line 10 of the `generate_PCIs` procedure; this is mandatory to preserve a consistency between the index i and the connection to a given transition (see Remark 2). For each input transition t of p, the corresponding TCI id_t is retrieved from the behavior of design d. Then, the internal signal associated with the `fired` output port in the output port map of TCI id_t is retrieved (i.e. $\texttt{actual}(\texttt{fired}, o_t)$), and the signal is associated with the subelement of the `itf` input port with index i. We know that the `generate_TCIs` function has generated the association between the `fired` output port and an internal signal in the output port map of all TCIs. Thus, the `actual` function never raises an error.

Remark 2 (Connections consistency)**.** *In the behavior of the* `place` *design, some processes access to the subelements of composite ports through the use of indices. For instance, the* `input_tokens_sum` *process (see Appendix* **??***) increments a local variable i in range* 0 *to* `input_arcs_number` $- 1$ *in a for loop. The process tests the value of the* `itf` *port's subelement with index i. If the test suceeds, the process adds the value of the* `iaw` *port's subelement with index i to the local variable* `v_internal_input_token_sum`*. Thus, the subelement with index i of the* `itf` *and* `iaw` *ports must refer to the connection to the same transition. Otherwise, the process does not compute a correct input tokens sum. Figure 1.6 illustrates the correct connection of the* `itf` *and* `iaw` *ports in the input port map of PCI id_p w.r.t. to the connection between transitions t_a, t_b, t_c and place p.*

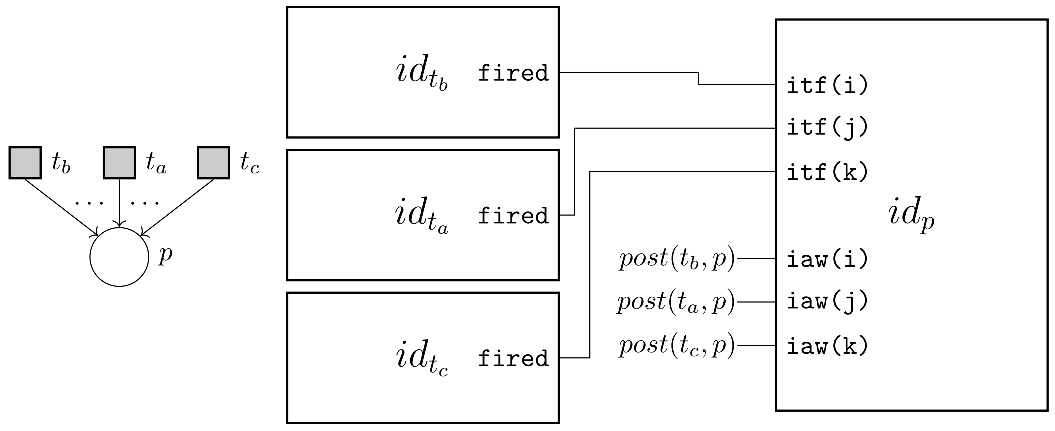

FIGURE 1.6: An example of correct connections between the PCI id_p and TCIs id_{t_a}, id_{t_b} and id_{t_c}. On the left, the input SITPN model showing the connections of the transitions t_a, t_b and t_c to the place p. The dots indicate that the place p possibly has other input transitions. On the right, the TCIs and the PCI generated by the transformation. In the input port map of PCI id_p, the subelements of the `itf` input port are connected to the `fired` port of TCIs; the subelements of the `iaw` port are connected to constant values, i.e. the weight of the arcs between place p and the input transitions of p.

*It is the rôle of the transformation function to ensure the consistency of the connections of the subelements in the input and output port maps of PCIs. The input and output port maps of TCIs are not subject to such a constraint. The fact that a **foreach** loop always iterates in the same order over the elements of a set ensures the consistency of the connections.*

From Line 9 to Line 16, the procedure connects the PCI id_p to the TCIs implementing the conflicting output transitions of place p. For each conflicting output transition t of p, the corresponding TCI id_t is retrieved from the behavior of design d. The function call $\texttt{actual}(\texttt{fired}, o_t)$ returns the internal signal associated with the `fired` output port in the output port map of TCI id_t. This internal signal is then connected to the subelement of the `otf` input port with index i in the input port map of PCI id_p. At Line 12, the `connect` function generates an internal signal id and adds it to the internal signal declaration list of design d. Then, the function associates the subelement $\texttt{oav}(i)$ (i.e. the subelement of the `oav` input port with index i) with the internal signal id in the output port map o_p, and it associates one subelement of the `iav` input port to the internal signal id in the input port map i_t. The `connect` function operates similarly on the `rtt` output port and the `rt` input port at Line 13, and on the `pauths` input port and the `pauths` output port at Line 14. Finally, at Line 15, the old TCI id_t is replaced by the modified one in the behavior of design d.

From Line 18 to Line 26, the procedure connects the TCIs implementing to the output transitions of p that are not in conflict. Note that the variable i is not reset between the two **foreach** loops to preserve the continuity of indices. For each non-conflicting output transition t of p, the corresponding TCI id_t is retrieved from the behavior of design d. Then, the interconnections between PCI id_p and TCI id_t are similar to the ones that have been performed for the conflicting transitions of p. The difference lies in the connection the `pauths` ports. Between the PCI id_p and its *non-conflicting* TCIs, the `pauths` are not connected together; this to reflect the independence of non-conflicting output transitions regarding the priority authorizations. Instead, the subelement of the `pauths` output port with index i is connected to a newly generated internal signal id_s in the output port map o_p (Line 22 to Line 24); the internal signal id_s is not connected to anything, and it will be removed by the industrial compiler at the time of the synthesis. Also, one subelement of the `pauths` input port is associated with `true` in the input port map i_t (Line 25); this connection represents the fact that, since the transition t is not a conflicting transition of place p, then, transition t always has the authorization to be fired, given that it is firable.

1.3.5 Generation of ports, the `action` and the `function` process

The last part of the transformation pertains to the generation of the input and output ports of the $\mathcal{H}$-VHDL design. The input ports implement the conditions declared in the input SITPN model. Each input port is associated with a condition through the γ binder. This binding is built during the transformation. The output ports of the $\mathcal{H}$-VHDL design implement the action and function of the input SITPN. Each output port is associated with an action or a function through the γ binder. During the simulation of a $\mathcal{H}$-VHDL design, the value of an output port represent the activation/execution status of the associated action/function. Algorithm 9 presents the `generate_ports` procedure. This procedure calls three procedures, namely: the `generate_condition_ports` procedure, responsible for the generation and the connection of input ports implementing conditions; the `generate_action_ports` procedure, responsible for the generation of output ports implementing actions, and for the generation of the `action` process; the `generate_function_ports` procedure, responsible for the generation of output ports implementing functions, and for the generation of the `function` process. These three procedures are detailled in Algorithms 10, 11 and 12.

Algorithm 9: `generate_ports`$(sitpn, d, \gamma)$

```
1 generate_condition_ports($sitpn, d, \gamma$)
2 generate_action_ports($sitpn, d, \gamma$)
3 generate_function_ports($sitpn, d, \gamma$)
```

Algorithm 10 describes the `generate_condition_ports` procedure.

Algorithm 10: `generate_condition_ports`$(sitpn, d, \gamma)$

```
1 foreach $c \in \mathcal{C}$ do
2     $id_c \leftarrow$ genid()
3     $d.ports \leftarrow d.ports \cup \{($in$, id_c,$ boolean$)\}$
4     $\gamma \leftarrow \gamma \cup \{(c, id_c)\}$
5     foreach $t \in$ trs$(c)$ do
6         comp$(id_t,$ transition$, g_t, i_t, o_t) \leftarrow$ get_comp$(\gamma(t), d.cs)$
7         if $\mathbf{C}(t,c) = 1$ then cassoc$(i_t,$ ic$, id_c)$
8         else if $\mathbf{C}(t,c) = -1$ then cassoc$(i_t,$ ic$,$ not $id_c)$
9         put_comp$(id_t,$ comp$(id_t,$ transition$, g_t, i_t, o_t), d.cs)$
```

The `generate_condition_ports` procedure iterates over the set of conditions of the *sitpn* parameter. For each condition of the set, the `generate_condition_ports` procedure produces a corresponding input port identifier id_c, and adds an input port declaration entry in the port declaration list of design d. The declared input port is of the Boolean type. Also, a binding between condition c and identifier id_c is added to γ. Then, the procedure performs the connection between the input port id_c and the `ic` input port present in the input interface of TCIs. The `ic` input port is an array composed of Boolean subelements. Indeed, as multiple conditions can be attached to a given transition, a given TCI is possibly connected to multiple input ports implementing conditions through its `ic` port. At Line 5, the **foreach** loop iterates over the set of transitions attached to condition c. For each such transition t, the corresponding TCI id_t is retrieved from the behavior of design d. Then, depending on the relation that exists between condition c and transition t, an association between id_c and one subelement of the `ic` input port is added to the input port map i_t. At the end of the loop, the old TCI id_t is replaced by a new TCI, with an updated input port map, in the behavior of design d.

Algorithm 11 describes the `generate_action_ports` procedure.

Algorithm 11: `generate_action_ports`$(sitpn, d, \gamma)$

```
1  $rstss \leftarrow$ null
2  $fss \leftarrow$ null
3  foreach $a \in \mathcal{A}$ do
4      $id_a \leftarrow$ genid()
5      $d.ports \leftarrow d.ports \cup \{(\texttt{out}, id_a, \texttt{boolean})\}$
6      $\gamma \leftarrow \gamma \cup \{(a, id_a)\}$
7      $e_{id_a} \leftarrow$ false
8      foreach $p \in \texttt{pls}(a)$ do
9          $\texttt{comp}(id_p, \texttt{place}, g_p, i_p, o_p) \leftarrow \texttt{get\_comp}(\gamma(p), d.cs)$
10         $id_s \leftarrow \texttt{actual}(\texttt{marked}, o_p)$
11         $e_{id_a} \leftarrow id_s$ or $e_{id_a}$
12     $rstss \leftarrow rstss;\ id_a \Leftarrow$ false
13     $fss \leftarrow fss;\ id_a \Leftarrow e_{id_a}$
14 $d.cs \leftarrow d.cs \mid\mid \texttt{process}(\texttt{action}, \{\texttt{clk}\}, \emptyset, \texttt{rst}\ (rstss)\ (\texttt{falling}\ fss))$
```

The `generate_action_ports` procedure does two things. First, it generates an output port for each action of the input SITPN; second, it builds the `action` process that is responsible for the assignment of the value of *action* ports depending on the value of the `marked` output ports of PCIs. The `action` process is a synchronous process; its statement body is composed of a single `rst` block. A `rst` block is composed of two blocks of sequential statements; the first block is executed only during an initialization phase, otherwise, the second block is executed. Here, the second block corresponds to a `falling` block, i.e. a block that is only executed during a falling edge phase. Thus, the `generate_action_ports` procedure builds two blocks of sequential statements: the first one, hold in the *rstss* variable, corresponds to the first part of the `rst` block (i.e. the one executed during the initialization phase); the second one, hold in the *fss* variable, corresponds to the second part of the `rst` block, i.e. a falling edge block. The first two lines of the procedure initialize the $rstss$ and fss with the `null` sequential statement. Then, in the abscence of actions defined in the input SITPN, the statement body of the `action` process is composed of `null` statements; the execution of `null` statements has no effect on the state of design during a simulation. At Line 3, the procedure iterates over the set of actions of the $sitpn$ parameter. For each action a in the set, an output port identifier id_a is generated, an output port declaration entry is added to the port declaration list of design d, the binding between action a and identifier id_a joins the γ binder.

An action is activated at given state if one of its attached place is marked, i.e. its marking is greater than zero. An output port identifier that implements the activation status of a given action is assigned in the falling block of the `action` process. The expression assigned to the output port id_a corresponds to the `or` sum between each `marked` port of the PCIs implementing the places attached to the action a. From Line 7 to Line 13, the `generate_action_ports` procedure builds this `or` sum expression. For each place p associated with the action a, the corresponding PCI id_p is retrieved from the behavior of design d. The internal signal id_s associated with the `marked` port is looked up in the output port map of PCI id_p. Then, the signal identifier id_s is composed with the expression e_{id_a} with the `or` operator. At the end of the loop started at Line 3, the procedure adds a new signal assignment statement to the $rstss$ and to the fss variables by composition with the ; operator. In the $rstss$ variable, i.e. in the part of the `action` process executed during an initialization phase, the id_a output port is assigned to `false`. In the fss variable, i.e. the part of the `action` process executed during a falling edge phase, the id_a output port is assigned to the previously built `or` sum expression e_{id_a}. The last line of the procedure builds and adds the `action` process to the behavior of design d. The `action` process is a synchronous process, thus, it declares the `clk` signal in its sensitivity list. The `action` process has an empty set of local variables. Finally, its statement body is composed of a `rst` block with $rstss$ as a first block, and a `falling` edge block wrapping fss as a second block.

Algorithm 12 describes the `generate_function_ports` procedure.

Algorithm 12: `generate_function_ports`$(sitpn, d, \gamma)$

```
1  $rstss \leftarrow$ null
2  $rss \leftarrow$ null
3  foreach $f \in \mathcal{F}$ do
4      $id_f \leftarrow$ genid()
5      $d.ports \leftarrow d.ports \cup \{(\texttt{out}, id_f, \texttt{boolean})\}$
6      $\gamma \leftarrow \gamma \cup \{(f, id_f)\}$
7      $e_{id_f} \leftarrow$ false
8      foreach $t \in \texttt{trs}(f)$ do
9          $\texttt{comp}(id_t, \texttt{transition}, g_t, i_t, o_t) \leftarrow \texttt{get\_comp}(\gamma(t), d.cs)$
10         $id_s \leftarrow \texttt{actual}(\texttt{fired}, o_t)$
11         $e_{id_f} \leftarrow id_s$ or $e_{id_f}$
12     $rstss \leftarrow rstss;\ id_f \Leftarrow$ false
13     $rss \leftarrow rss;\ id_f \Leftarrow e_{id_f}$
14 $d.cs \leftarrow d.cs \mid\mid \texttt{process}(\texttt{function}, \{\texttt{clk}\}, \emptyset, \texttt{rst}\ (rstss)\ (\texttt{rising}\ rss))$
```

The `generate_function_ports` procedure does two things. First, it generates an output port for each function of the input SITPN; second, it builds the `function` process that is responsible for the assignment of the value of *function* ports depending on the value of the `fired` output ports of PCIs. Similarly to the `action` process, the `function` process is a synchronous process with a statement body composed of a single `rst` block. The second part of the `rst` block is a `rising` block, i.e. a block that is only executed during a rising edge phase. Thus, the `generate_function_ports` procedure builds two blocks of sequential statements: the first one, hold in the *rstss* variable, corresponds to the first part of the `rst` block (i.e. the one executed during the initialization phase); the second one, hold in the *rss* variable, corresponds to the second part of the `rst` block, i.e. a rising edge block. The first two lines of the procedure initialize the *rstss* and *rss* with the `null` sequential statement. At Line 3, the procedure iterates over the set of functions of the *sitpn* parameter. For each function f in the set, an output port identifier id_f is generated, an output port declaration entry is added to the port declaration list of design d, the binding between function f and identifier id_f joins the γ binder.

An function is executed at given state if one of its attached transition is fired. An output port identifier that implements the execution status of a given function is assigned in the rising block of the `function` process. The expression assigned to the output port id_f corresponds to the `or` sum between each `fired` port of the TCIs implementing the transitions attached to the function f. From Line 7 to Line 13, the `generate_function_ports` procedure builds this `or` sum expression. For each transition t associated with the function f, the corresponding TCI id_t is retrieved from the behavior of design d. The internal signal id_s associated with the `fired` port is looked up in the output port map of TCI id_t. Then, the signal identifier id_s is composed with the expression e_{id_f} with the `or` operator. At the end of the loop started at Line 3, the procedure adds a new signal assignment statement to the *rstss* and to the *rss* variables by composition with the

; operator. In the *rstss* variable, i.e. in the part of the `function` process executed during an initialization phase, the id_f output port is assigned to `false`. In the *rss* variable, i.e. the part of the `function` process executed during a rising edge phase, the id_f output port is assigned to the previously built `or` sum expression e_{id_f}. The last line of the procedure builds and adds the `function` process to the behavior of design d. The `function` process is a synchronous process, thus, it declares the `clk` signal in its sensitivity list. The `function` process has an empty set of local variables. Finally, its statement body is composed of a `rst` block with *rstss* as a first block, and a `rising` edge block wrapping *rss*.

1.4 Coq implementation of the HILECOP model-to-text transformation

This section presents the implementation of the HILECOP model-to-text transformation with the Coq proof assistant. The full implementation is available under the `sitpn2hvhdl` folder of the following Git repository: `https://github.com/viampietro/ver-hilecop`

Listing 1.1 gives the Coq implementation of the `sitpn_to_hvhdl` function presented in an imperative pseudo-code version in Algorithm 1.

```
Definition sitpn_to_hvhdl (sitpn : Sitpn)
   (decpr : forall x y :  T sitpn, {pr x y} + {~pr x y})
   (id_e id_a : ident) (mmf : P sitpn → nat) :
   (design * Sitpn2HVhdlMap sitpn) + string :=
  RedV
    (( do _ ← generate_sitpn_infos sitpn decpr;
      do _ ← generate_architecture sitpn mmf;
      do _ ← generate_ports sitpn;
      do _ ← generate_comp_insts sitpn;
      generate_design_and_binder id_e id_a)
      (InitS2HState sitpn Petri.ffid)).
```

LISTING 1.1: The Coq implementation of the `sitpn_to_hvhdl` function presented in Algorithm 1.

In Listing 1.1, the `sitpn_to_hvhdl` function has five parameter: `sitpn`, the input SITPN model; `decpr`, a proof that the `pr` relation (i.e. the implementation of the firing priority relation) is decidable over the set of transitions of `sitpn` (i.e. `T sitpn`); id_e and id_a, the entity and architecture idenfifiers for the generated $\mathcal{H}$-VHDL design; the `mmf` function that maps the places of the `sitpn` parameter to a maximal marking value, i.e. a natural number. The `sitpn_to_hvhdl` function returns a couple composed of the generated $\mathcal{H}$-VHDL design, of type `design`, and the generated γ binder, of type `Sitpn2HVhdlMap sitpn`; or, the `sitpn_to_hvhdl` function returns a `string` corresponding to an error message.

In the body of the `sitpn_to_hvhdl` function, the `RedV` is a notation that reduces a monadic function call to a value. Our implementation of the HILECOP transformation function relies on the state-and-error monad [17]. Each function that implements a part of the transformation function takes a *compile-time* state as a parameter, and returns either a value and a new compile-time state, or an error message. The bind construct of the state-and-error monad permits to pipeline multiple function calls, and, combined with the do notation, it permits to write functional programs in the style of imperative languages. The sequence defined in the body of the sitpn_to_hvhdl function gives an example of what can be achieved with the combination of the state-and-error monad and the do notation. This sequence constitutes a single monadic function that takes a state of the Sitpn2HVhdlState type (see Listing 1.2) as input, and yields a value with a new state, or an error message. Here, the RedV notation retrieves only the value returned by the application of the monadic function to the parameter (InitS2HState sitpn Petri.ffid) (i.e. the initial compile-time state), or it retrieves the error message.

In the do sequence of Listing 1.1, the four first function calls do not return values that are relevant; thus, we use the underscore notation to notify that we are not interested in the returned values. Indeed, the generate_sitpn_infos, generate_architecture, generate_ports and generate_comp_insts functions directly modify the compile-time state without returning a value. They are the functional implementation of the procedures described in the previous section.

Now, let us present the content of the compile-time state. As said above, the compile-time state is carried from function to function and modified all along the transformation. Listing 1.2 gives the implementation of the compile-time state structure.

```
1  Record Sitpn2HVhdlState (sitpn : Sitpn) : Type :=
2    MkS2HState {
3        lofPs : list (P sitpn);
4        lofTs : list (T sitpn);
5        lofCs : list (C sitpn);
6        lofAs : list (A sitpn);
7        lofFs : list (F sitpn);
8        nextid : ident;
9        sitpninfos : SitpnInfos sitpn;
10       iports : list pdecl;
11       oports : list pdecl;
12       arch : Architecture sitpn;
13       beh : cs;
14       γ : Sitpn2HVhdlMap sitpn;
15
16   }.
```

LISTING 1.2: The compile-time state structure defined as the Coq Sitpn2HVhdlState record type.

The compile-time state structure is implemented by the Sitpn2HVhdlState record type. This type depends on a given sitpn passed as a parameter. It is composed of eleven fields. The first five fields (Line 3 to 7) are the list versions of the finite sets of places, transitions, conditions, actions and functions of the sitpn parameter. These fields are filled at the very beginning of the transformation by the generate_sitpn_infos function, and are convenient to write functions in the context of dependent types. The nextid field (Line 8) permits to generate fresh and unique identifiers all along the transformation. The sitpinfos field (Line 9) is an instance of the SitpnInfos type that depends on the sitpn parameter. The sitpninfos field is filled up by the `generate_sitpn_infos` function. It is a convenient way to represent the information associated with the places, transitions, conditions, actions and functions of the `sitpn` parameter. The `iports` (resp. `oports`) field, at Line 10 (resp. at Line 11), gathers the list of input (resp. output) port declarations of the generated $\mathcal{H}$-VHDL design. The `arch` field (Line 12) is an intermediary representation of the behavior of the generated $\mathcal{H}$-VHDL design. This representation is easier to modify and to handle than a $\mathcal{H}$-VHDL concurrent statement. The `beh` field (Line 13) is the behavior of the generated $\mathcal{H}$-VHDL design; it is an instance of the `cs` type, i.e. the type of concurrent statements defined in the abstract syntax of $\mathcal{H}$-VHDL. The γ field (Line 14) is the SITPN-to-$\mathcal{H}$-VHDL binder generated alongside the $\mathcal{H}$-VHDL design, and returned at the end of the transformation.

At the beginning of the transformation, an initial compile-time state is built with the `InitS2HState` function. The `InitS2HState` function gives a initial value to the fields of the state structure; mostly, the fields are initialized with empty lists, and the `beh` field is initialized with the `null` statement. The `InitS2HState` function takes an `Sitpn` instance and an identifier as inputs. The identifier parameter represents the initial value of the `nextid` field. In Listing 1.1, the second parameter of the `InitS2HState` function is `Petri.ffid`. It corresponds to the *first fresh* identifier that the transformation can use to produce a $\mathcal{H}$-VHDL design that respects the uniqueness of identifiers.

Let us now present the functions composing the `do` sequence of the `sitpn_to_hvhdl` function, and how they modify the compile-time state to produce the final $\mathcal{H}$-VHDL design and the γ binder.

1.4.1 The `generate_sitpn_infos` function

Listing 1.3 presents a part of the `generate_sitpn_infos`. The part that is let aside, represented by little dots, pertains to the creation of the dependently-typed lists constituting the first fields of the compile-time state structure (Line 3 to 7 in Listing 1.2).

```
Definition generate_sitpn_infos
          (sitpn : Sitpn)
          (decpr : forall x y : T sitpn, {pr x y} + {~pr x y}) :
  Mon (Sitpn2HVhdlState sitpn) unit :=
  ...
  do _ ← check_wd_sitpn sitpn decpr;
  do _ ← generate_trans_infos sitpn;
  do _ ← generate_place_infos sitpn decpr;
  do _ ← generate_cond_infos sitpn;
  do _ ← generate_action_infos sitpn;
  generate_fun_infos sitpn.
```

LISTING 1.3: A part of the `generate_sitpn_infos` function.

The `generate_sitpn_infos` function takes an `Sitpn` instance and a proof of decidability for the `pr` relation as parameters. It returns a value of type `Mon (Sitpn2HVhdlState sitpn) unit`. A value of this type can either be a couple $(state, value)$, where $state$ is of type `(Sitpn2HVhdlState sitpn)` and $value$ is of type `unit`, or an error message. The `unit` type as only one possible value tt. The unit type is used here to represent a function that modifies the compile-time state without returning a value.

The aim of the generate_sitpn_infos function is to fill the sitpninfos field of the compile-time state; the sitpninfos field is an instance of the SitpnInfos record type. Listing 1.4 presents the definition of the SitpnInfos record type, along with the definition of the PlaceInfo and TransInfo record types.

```
Record PlaceInfo (sitpn : Sitpn) : Type :=
  MkPlaceInfo { tinputs : list (T sitpn);
                tconflict : list (T sitpn);
                toutputs : list (T sitpn) }.

Record TransInfo (sitpn : Sitpn) : Type :=
  MkTransInfo { pinputs : list (P sitpn); conds : list (C sitpn) }.

Record SitpnInfos (sitpn : Sitpn) : Type :=
  MkSitpnInfos {
      pinfos : list (P sitpn * PlaceInfo);
      tinfos : list (T sitpn * TransInfo);
      cinfos : list (C sitpn * list (T sitpn));
      ainfos : list (A sitpn * list (P sitpn));
      finfos : list (F sitpn * list (T sitpn));
    }.
```

LISTING 1.4: The PlaceInfo, TransInfo and SitpnInfos record types.

The PlaceInfo record type is composed of three lists that represent the input transitions, tinputs, the conflicting output transitions, tconflict, and the non-conflicting output transitions, toutputs, of a place. In the SitpnInfos structure, the pinfos field maps the places of the sitpn parameter to their respective informations, i.e. an instance of the PlaceInfo type. This mapping is built by the generate_place_infos function called in the body of generate_sitpn_infos function. While building an instance of the PlaceInfo type for a given place p, the generate_place_infos function computes the list of output transitions of p that are conflict in the manner of the $\texttt{output}_c$ function described in Section 1.3.2. First, it computes the list of output transitions that are linked to the place p through a basic arc; then, the function checks if all conflicts between the transitions of this list are solved by means of mutual exclusion. If it is the case, the tconflict field is left empty, and all transitions of the list join the toutputs list. Otherwise, the function tries to establish a strict total order over the transitions of the list, by decreasing level of priority. If no such order can be established, the function raises an error; otherwise, the tconflict field is filled with the ordered list.

The TransInfo record type is composed of two lists that represent the input places, pinputs, and the output places, poutputs, of a transition. In the SitpnInfos structure, the tinfos field maps the transitions of the sitpn parameter to their respective informations, i.e. an instance of the TransInfo type. This mapping is built by the generate_trans_infos function called in the body of generate_sitpn_infos function.

In the SitpnInfos structure, the cinfos (resp. ainfos and finfos) field maps the conditions (resp. actions and functions) of the sitpn parameter to the list of transitions (resp. places and transitions) they are attached to. This mapping is built by the generate_cond_infos (resp. generate_action_infos and generate_fun_infos) function called in the body of generate_-sitpn_infos function.

At the beginning of the generate_sitpn_infos function, the check_wd_sitpn function partly checks the well-definition of the sitpn parameter. Precisely, it checks that the set of places and transitions of the sitpn parameter are not empty, and that the priority relation is a strict order, i.e. transitive and reflexive, over the set of transitions. The other parts of the well-definition checking are performed later during the transformation. For instance, the generate_place_in-fos function checks that, for each group of transitions in conflict, the conflicts are either solved by means of mutual exclusion or that the priority relation is a strict total order over this group. It also checks that there are no isolated places in the input sitpn parameter, etc.

1.4.2 The generate_architecture function

Listing 1.5 presents the generate_architecture function. The generate_architecture function implements the generate_architecture and the generate_interconnections procedures detailled in Algorithms 3 and 8. The composition of the generate_place_map and the generate_trans_map functions implements generate_architecture procedure of Algorithm 3. Precisely, the generate_place_map function implements the generate_PCIs procedure presented in Algorithm 4, and the generate_trans_map function implements the generate_TCIs procedure presented in Algorithm 5.

```
Definition generate_architecture (sitpn : Sitpn) (mmf : P sitpn → nat) :
  Mon (Sitpn2HVhdlState sitpn) unit :=
  do _ ← generate_place_map sitpn mmf;
  do _ ← generate_trans_map sitpn;
  generate_interconnections.
```

LISTING 1.5: The generate_architecture function that implements the generate_architecture procedure of Algorithm 3.

The generate_architecture function takes an Sitpn instance and the mmf function as inputs, and modifies the compile-time state. The generate_architecture function fills the arch field of the compile-time state; the arch field is an instance of the Architecture record type. Listing 1.4 presents the definition of the Architecture record type, along with the definition of the InputMap, OutputMap and HComponent type aliases.

```
Definition InputMap := list (ident ∗ (expr + list expr)).
Definition OutputMap := list (ident ∗ ((option name) + list name)).
Definition HComponent := (genmap ∗ InputMap ∗ OutputMap).

Record Architecture (sitpn : Sitpn) := MkArch {
  sigs  : list sdecl;
  plmap : list (P sitpn ∗ HComponent);
  trmap : list (T sitpn ∗ HComponent);
  fmap  : list (F sitpn ∗ list expr);
  amap  : list (A sitpn ∗ list expr) }.
```

LISTING 1.6: The `Architecture` record type, and the `InputMap`, `OutputMap` and `HComponent` subsidiary types.

The `HComponent` type is an intermediate representation of an $\mathcal{H}$-VHDL component instantiation statement. This type has been devised to ease the construction of PCIs and TCIs, and of their generic, input port and output port maps all along the transformation. The `HComponent` type is a triplet composed of a generic map as defined in the $\mathcal{H}$-VHDL abstract syntax, an instance of the `InputMap` type, and an instance of the `OutputMap` type. The `InputMap` type maps an input port identifier to a either a simple expression or to a list of expressions, where the `expr` type is the type of expressions defined in the $\mathcal{H}$-VHDL abstract syntax. In an `InputMap` instance, an input port identifier of a scalar type (i.e. Boolean or constrained natural) is mapped to a simple expression, whereas an input port identifier of the array type is mapped to a list of expressions. Each expression of the list represents the actual part associated with one subelement of the input port. Similarly to the `InputMap` type, the `OutputMap` type maps an output port identifier to either an option to a signal (the `None` value representing the connection to the `open` keyword) name, or to a list of signal names. In the definition of the `OutputMap` type, the `name` type represents the type of simple identifiers or indexed identifiers defined in the $\mathcal{H}$-VHDL abstract syntax.

The `Architecture` record type is an intermediary representation of the behavioral and declarative part of an $\mathcal{H}$-VHDL design's architecture. The `sigs` field of the `Architecture` type represents the internal signal declaration list constituting the declarative part of an $\mathcal{H}$-VHDL design's architecture. The transformation adds a new signal declaration entry to the `sigs` field every time a internal signal must be generated, for example, during the generation of interconnections between PCIs and TCIs. The `plmap` (resp. the `trmap`) field maps the places (resp. transitions) of the `sitpn` parameter to their corresponding PCI (resp. TCI) implemented in an intermediate format, i.e. an instance of the `HComponent` type. The `fmap` field of the `Architecture` type maps the functions of the `sitpn` parameter to a list of expressions. For a given function f, the associated list of expressions corresponds to the list of internal signals associated with the `fired` port of the TCIs implementing the transitions of the `trs`(f) set (i.e. the set of transitions associated with function f). The `fmap` field is filled by the `generate_ports` function described in Listing 1.7. The `amap` field is the twin of the `fmap` field but on the side of the actions of the `sitpn` parameter. Thus, in the `amap` field, the list of expressions associated with an action a corresponds to the list of internal signals connected to the `marked` port of the PCIs implementing the places of a.

In the body of the `generate_architecture` function, the `generate_place_map` function implements the `generate_PCIs` procedure described in Algorithm 4. For each place of the `sitpn` parameter, the `generate_place_map` function builds an instance of the `HComponent` type, and adds an association between place and `HComponent` instance in the `plmap` field. The `generate_place_map` function fills the generic, input port and output port map of the `HComponent` instances as described in the `generate_PCIs` procedure. Following the `generate_place_map` function, the `generate_trans_map` function implements the `generate_TCIs` procedure described in Algorithm 5. For each transition of the `sitpn` parameter, the `generate_trans_map` function builds an instance of the HComponent type, and adds an association between transition and HComponent instance in the trmap field. The generate_trans_map function fills the generic, input port and output port map of the HComponent instances as described in the generate_TCIs procedure. Finally, the generate_interconnections function modifies the input and output port maps of the HComponent instances in the plmap and trmap fields, and thus, implements the interconnections described in the generate_interconnections procedure of Algorithm 8.

1.4.3 The generate_ports function

Listing 1.7 presents the generate_ports function called in the body of the sitpn_to_hvhdl function (see Listing 1.1). The generate_ports function implements the generate_ports procedure described in Algorithm 9. The generate_ports function calls three functions: the generate_action_ports_and_ps function that implements the generate_action_ports procedure of Algorithm 11, the generate_fun_ports_and_ps function that implements the generate_function_ports procedure of Algorithm 12, and the generate_and_connect_cond_ports that implements the generate_condition_ports procedure of Algorithm 10.

```
Definition generate_ports (sitpn : Sitpn) : Mon (Sitpn2HVhdlState sitpn) unit :=
  do _ ← generate_action_ports_and_ps;
  do _ ← generate_fun_ports_and_ps;
  generate_and_connect_cond_ports.
```

LISTING 1.7: The generate_ports function implementing the generate_ports procedure presented in Algorithm 9.

For every action of the sitpn parameter, the generate_action_ports_and_ps function adds a port declaration entry to the oports field of the compile-time state, and adds a binding between action and output port identifier in the γ field. It also builds the action process as described in the generate_action_ports procedure, and adds the process to the beh field of the compile-time state. The generate_fun_ports_and_ps does the same for the functions of the sitpn parameter, and similarly builds the function process and adds it to the beh field. The generate_and_connect_cond_ports function add a port declaration entry for every condition of the sitpn parameter to the iports field of the compile-time state. Then, it modifies the input port map of HComponent instances in the trmap of the compile-time state's arch field. The modifications pertain to the connection of input ports to the ic input port of TCIs, as described in the generate_condition_ports procedure (see Algorithm 10).

1.4.4 The generate_comp_insts and generate_design_and_binder functions

At the end of the sitpn_to_hvhdl function (see Listing 1.1), the generate_comp_insts function transforms the HComponent instances, associated with places and transitions in the compile-time state's arch field, into real component instantiation statements as defined in the $\mathcal{H}$-VHDL abstract syntax. Then, the generate_design_and_binder builds up the final $\mathcal{H}$-VHDL design and the γ binder, and returns the couple. Listing 1.8 presents the generate_comp_insts function and the generate_design_and_binder function.

```
Definition generate_comp_insts (sitpn : Sitpn) : Mon (Sitpn2HVhdlstate sitpn) unit :=
  do _ ← generate_place_comp_insts sitpn; generate_trans_comp_insts sitpn.

Definition generate_design_and_binder (sitpn : Sitpn) (ide ida : ident) :
    Mon (Sitpn2HVhdlstate sitpn) (design ∗ Sitpn2HVhdlMap sitpn) :=
  do s ← Get;
  Ret (( design_ ide ida [] ((iports s) ++ (oports s)) (sigs (arch s)) (beh s)), (γ s)).
```

LISTING 1.8: The `generate_comp_insts` and the `generate_design_and_binder` function.

The `generate_comp_insts` function is needed because we are using an intermediary representation for the component instantiation statements. Even though this representation is convenient to manipulate data during the different phases of the transformation, it also implies an extra generation step to complete the generation of the $\mathcal{H}$-VHDL design and the γ binder. The `generate_comp_insts` function calls the `generate_place_comp_insts` and the `generate_trans_comp_insts` functions. These two functions being similar in all points, except for their type of the inputs, we are only presenting the `generate_place_comp_insts` function here. The `generate_place_comp_insts` function calls the `generate_place_comp_inst` function for each place defined in the set of places of the `sitpn` parameter. Listing 1.9 presents the code the `generate_place_comp_inst` function.

```
Definition generate_place_comp_inst (sitpn : Sitpn) (p : P sitpn) :
      Mon (Sitpn2HVhdlstate sitpn) unit :=

      do idp ← get_nextid;
      do _      ← bind_place p idp;
      do pcomp  ← get_pcomp p;
      do pci    ← HComponent_to_comp_inst idp place_entid pcomp;
      add_cs pci.
```

LISTING 1.9: The `generate_place_comp_inst` function.

The `generate_place_comp_inst` function generates a fresh and unique PCI identifier by appealing to the `get_nextid` function. The `get_nextid` function returns and increments the current value of the `nextid` field, defined in the compile-time state. Then, the `bind_place` function adds a binding between the place `p` and the identifier id_p in the γ field of the compile-time state. The `get_pcomp` function looks up the `plmap` field (defined under the `arch` field of the compile-time state) and returns the `HComponent` instance associated with the place `p`, i.e. `pcomp`. The `HComponent_to_comp_inst` function translates the `HComponent` instance `pcomp` into a PCI with the identifier id_p. Finally, the `add_cs` function composes the returned PCI with the current $\mathcal{H}$-VHDL design behavior, hold in the `beh` field of the compile-time state.

The transformation of a `HComponent` instance into a PCI implies the translation of the input and output port map, which are instances of the `InputMap` and `OutputMap` types, into their equivalent representation in $\mathcal{H}$-VHDL abstract syntax. The translation especially concerns the association between a port identifier of the array type and a list of expressions, or names. For instance, let us consider an instance of `InputMap` that is an intermediary representation of the input port map of a PCI id_p. In this `InputMap` instance, the `itf` port, which is a composite input port of the `place` design, is associated with the list $[id_a, id_b, id_c]$. Then, based on the previous association, the `HComponent_to_comp_inst` function generates the following associations is the concrete input port map of PCI id_p: $(\texttt{rt}(0), id_a)$, $(\texttt{rt}(1), id_b)$ and $(\texttt{rt}(2), id_c)$.

Getting back to Listing 1.8, the `generate_design_and_binder` function retrieves the current compile-time state `s` with the `Get` function. Then, based on the value of the different fields of the compile-time state, the function builds an $\mathcal{H}$-VHDL design and returns it along with the γ binder. The $\mathcal{H}$-VHDL design receives the id_e and id_a identifiers passed as parameters as its entity and architecture identifiers. The generic constant declaration list of the $\mathcal{H}$-VHDL design is empty, i.e. it receives the empty list value. The port declaration list of the $\mathcal{H}$-VHDL design is built by concatenating the content of the `iports` and `oports` fields defined in state `s`. The internal signal declaration list is filled by the `sigs` field, defined under the `arch` field of state `s`. Finally, the `beh` field fills the behavior of the $\mathcal{H}$-VHDL design.

1.5 Conclusion

The purpose of this chapter was to give to the reader a complete understanding of the HILECOP model-to-text transformation function, and of what makes it a very specific transformation case. We first gave an informal presentation of the transformation function with a high-level view of the transformation principles. Then, we presented our literature review pertaining to transformation functions in the context of formal verification, with a particular focus on the expression and the implementation of transformation functions. Two points, drawn out from the literature review, are of particular interest. First, the review showed that it is important, during a transformation, to keep the binding between the elements of the source representation and their corresponding versions in the target representation. This binding is the base of the comparison of the run-time state of the source and target representation that permits to express the theorems of semantic preservation. Second, if the distance between the source and the target representation is too important, it is easier, while aiming at proving a semantic preservation property, to split the transformation into multiple simple transformation steps. Then, to each transformation step will correspond an intermediary representation, and a theorem of semantic preservation will be laid out and proved for each one of them. In the case of the HILECOP model-to-text transformation, even though the transformation as a lot of tricky aspects pertaining to particular cases of input models, there is no need to split the transformation into simple steps with intermediary representations. Even though the verification task is quite close, the HILECOP transformation is quite different from the certified GPL or the HDL compilers presented in the literature review. Indeed, the source representation is an input model not a programming language. Moreover, due to the interconnection of the component instances generated by the transformation function, devising an transformation algorithm that generates modular and independently executable code is impossible. As everything is connected, one has to reason over the entire transformation process to get the overall behavior of the generated $\mathcal{H}$-VHDL design. This is also one of the main difference between the HILECOP transformation and compilers for programming languages. Despite at all that, the transformation algorithm, presented in this chapter, gets as close as possible to a completed modular expression of the HILECOP transformation. Our implementation of the transformation algorithm uses an intermediate format to represent the component instantiation statements. Even though this intermediate representation is convenient to handle datas during the transformation, it requires an extra transformation step to generate the behavior of the resulting $\mathcal{H}$-VHDL design. Thus, this intermediate format complexifies the transformation function. In the future, we will implement the transformation function closer to the expression of the transformation algorithm. This will help reasoning over the `sitpn_to_hvhdl` function during the mechanization of the proof of semantic preservation.

Bibliography

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

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

[3] Thomas Bourgeat et al. "The Essence of Bluespec: A Core Language for Rule-Based Hardware Design". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. New York, NY, USA: Association for Computing Machinery, June 11, 2020, pp. 243–257. ISBN: 978-1-4503-7613-6. DOI: 10.1145/3385412.3385965. URL: https://doi.org/10.1145/3385412.3385965 (visited on 05/05/2021).

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

[5] Thomas Braibant and Adam Chlipala. "Formal Verification of Hardware Synthesis". In: *Computer Aided Verification*. Ed. by Natasha Sharygina and Helmut Veith. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 213–228. ISBN: 978-3-642-39799-8. DOI: 10.1007/978-3-642-39799-8_14.

[6] Daniel Calegari et al. "A Type-Theoretic Framework for Certified Model Transformations". In: *Formal Methods: Foundations and Applications*. Ed. by Jim Davies, Leila Silva, and Adenilso Simao. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 112–127. ISBN: 978-3-642-19829-8. DOI: 10.1007/978-3-642-19829-8_8.

[7] Adam Chlipala. "A Verified Compiler for an Impure Functional Language". In: *ACM SIGPLAN Notices* 45.1 (Jan. 17, 2010), pp. 93–106. ISSN: 0362-1340. DOI: 10.1145/1707801.1706312. URL: https://doi.org/10.1145/1707801.1706312 (visited on 05/22/2020).

[8] Benoît Combemale et al. "Essay on Semantics Definition in MDE. An Instrumented Approach for Model Verification". In: *Journal of Software* 4 (Nov. 1, 2009). DOI: 10.4304/jsw.4.9.943-958.

[9] Johannes Dyck, Holger Giese, and Leen Lambers. “Automatic Verification of Behavior Preservation at the Transformation Level for Relational Model Transformation”. In: *Software & Systems Modeling* 18.5 (5 Oct. 1, 2019), pp. 2937–2972. ISSN: 1619-1374. DOI: 10.1007/s10270-018-00706-9. URL: https://link.springer.com/article/10.1007/s10270-018-00706-9 (visited on 05/22/2020).

[10] Lukasz Fronc and Franck Pommereau. “Towards a Certified Petri Net Model-Checker”. In: *Programming Languages and Systems*. Ed. by Hongseok Yang. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 322–336. ISBN: 978-3-642-25318-8. DOI: 10.1007/978-3-642-25318-8_24.

[11] Xavier Leroy. “A Formally Verified Compiler Back-End”. In: *Journal of Automated Reasoning* 43.4 (Nov. 4, 2009), p. 363. ISSN: 1573-0670. DOI: 10.1007/s10817-009-9155-4. URL: https://doi.org/10.1007/s10817-009-9155-4 (visited on 01/21/2020).

[12] Andreas Lööw. “Lutsig: A Verified Verilog Compiler for Verified Circuit Development”. In: *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2021. New York, NY, USA: Association for Computing Machinery, Jan. 17, 2021, pp. 46–60. ISBN: 978-1-4503-8299-1. DOI: 10.1145/3437992.3439916. URL: https://doi.org/10.1145/3437992.3439916 (visited on 05/04/2021).

[13] Said Meghzili et al. “On the Verification of UML State Machine Diagrams to Colored Petri Nets Transformation Using Isabelle/HOL”. In: *2017 IEEE International Conference on Information Reuse and Integration (IRI)*. 2017 IEEE International Conference on Information Reuse and Integration (IRI). Aug. 2017, pp. 419–426. DOI: 10.1109/IRI.2017.63.

[14] Martin Strecker. “Formal Verification of a Java Compiler in Isabelle”. In: *Automated Deduction—CADE 18*. International Conference on Automated Deduction. Springer, Berlin, Heidelberg, July 27, 2002, pp. 63–77. DOI: 10.1007/3-540-45620-1_5. URL: https://link.springer.com/chapter/10.1007/3-540-45620-1_5 (visited on 06/08/2020).

[15] Yong Kiam Tan et al. “A New Verified Compiler Backend for CakeML”. In: (Sept. 4, 2016). DOI: 10.17863/CAM.6525.

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

[17] Philip Wadler. “The essence of functional programming”. In: *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL ’92. Association for Computing Machinery, 1992, 1–14. ISBN: 978-0-89791-453-6. DOI: 10.1145/143165.143169. URL: https://doi.org/10.1145/143165.143169.

[18] Zhibin Yang et al. “From AADL to Timed Abstract State Machines: A Verified Model Transformation”. In: *Journal of Systems and Software* 93 (July 1, 2014), pp. 42–68. ISSN: 0164-1212. DOI: 10.1016/j.jss.2014.02.058. URL: http://www.sciencedirect.com/science/article/pii/S0164121214000727 (visited on 01/16/2020).