

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

Acknowledgements	iii
1 The HILECOP model-to-text transformation	1
1.1 Informal presentation of the HILECOP transformation	1
1.2 Expressing transformation functions	7
1.2.1 Building transformation functions	7
1.3 The transformation algorithm	12
1.3.1 The <code>sitpn_to_hvhd1</code> function	12
1.3.2 Primitive functions and sets	14
1.3.3 Generation of component instances and constant parts	16
1.3.4 Interconnection of the place and transition component instances	21
1.3.5 Generation of ports, the action and the function process	24
1.4 Coq implementation of the HILECOP model-to-text transformation	27
Bibliography	31

List of Figures

1.1	Transformation of an input SITPN model into a top-level \mathcal{H} -VHDL design.	2
1.2	Generation of the place and transition component instances.	3
1.3	Generation of the interconnections between the place and transition component instances.	4
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.	6
1.5	Translation from Java expressions to Java bytecode expressions	8
1.6	An example of correct connections between several TCIs and a PCI.	23

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 ntantiation 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

For/Dedicated to/To my...

Chapter 1

The HILECOP model-to-text transformation

- Motivate the interest of the transformation in the introduction. Why the models are not hand-coded in VHDL directly? Because graphics help the design and communication, automatic generation because places and transitions interconnections to complex...
- Principles of SITPNs implementation in VHDL: place pivot, keep place and transition design as simple as possible, interconnections via Boolean signals only

The aim of this chapter is to present in details the HILECOP model-to-text transformation that we propose to verify as semantic preserving. The chapter is structured as follows. First, 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 detail the HILECOP transformation function in the form of an algorithm. Finally, in Section 1.4, we describe the Coq implementation of the algorithm.

1.1 Informal presentation of the HILECOP transformation

Here, we give an overview of the HILECOP transformation function. The goal is to give to the reader the means to appreciate the difference 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 an 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.

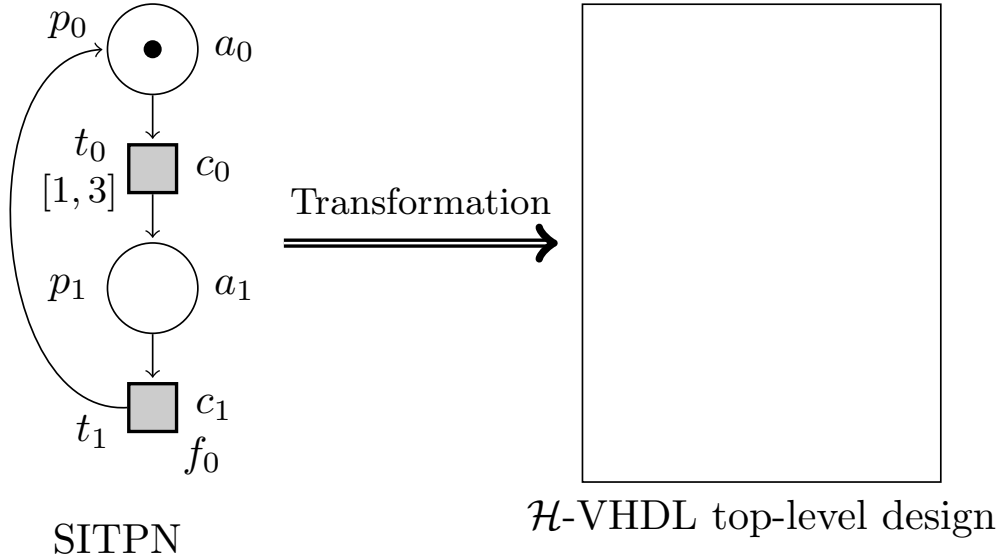


FIGURE 1.1: Transformation of an input SITPN model into a top-level \mathcal{H} -VHDL design. The input model is composed of two places, p_0 and p_1 , and two transitions, t_0 and t_1 . Transition t_0 is associated with the time interval $[1, 3]$ and the condition c_0 . Transition t_1 is associated with condition c_1 , and its firing triggers the execution of function f_0 . Action a_0 is activated when place p_0 is marked, and action a_1 is activated when 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 content of the behavior of the top-level design after this first generation step. In addition to the generation of PCIs and TCIs, all constant values are produced in the generic map and the input port map of component instances. The constant values pertain to all the information related to the structure of the input SITPN. The generic map of TCIs receive the number of conditions, the type of time interval, and the maximal value for the time counter associated with their corresponding transitions. The generic map of PCIs receive the number of input arcs, the number of output arcs, and the maximal marking associated with their corresponding places. The maximal marking associated with each place 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. This analysis takes place before the transformation into a \mathcal{H} -VHDL design in the proceeding of the HILECOP methodology. For now, a global maximal marking is passed as a parameter to the transformation function; therefore, all PCIs receive the same value for the `maximal_marking` generic constant in the first phase of the transformation. However, we can easily convert the global maximal marking into a function mapping the places of the input SITPN to a specific maximal marking value. Thus, each PCI would be associated with their own maximal marking value at the generation of their generic map. The input port map of

PCIs receive the weight and type of input arcs, and the weight of output arcs for each input and output arc of their corresponding places.

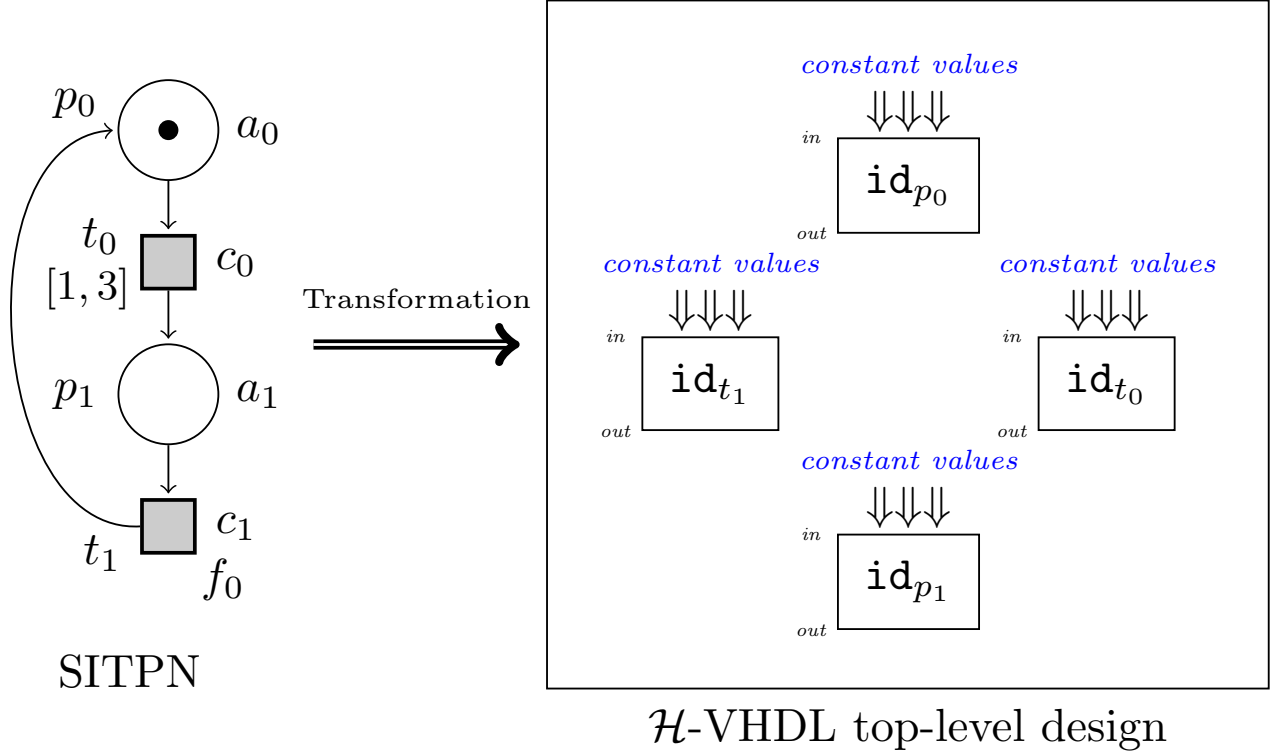


FIGURE 1.2: Generation of the place and transition component instances based on the set of places and transitions of the input SITPN. The PCI id_{p_0} implements the place p_0 , TCI id_{t_0} the transition t_0 ,...

After the generation of the PCIs, the TCIs, and of the constant parts of the generic and input port maps, the component instances are interconnected through their port interfaces. The PCIs and TCIs interact through their interfaces to exchange informations. Figure 1.3 illustrates the behavior of the top-level design after the interconnection of PCIs and TCIs. For instance, a PCI, 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`. The 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 about their sensitization status. To perform this exchange of information, the transformation generates an internal signal to connect the subelements of the `output_arcs_valid` output port¹, where `output_arcs_valid` is defined in the output port map of the PCI. Each subelement of the `output_arcs_valid` port is connected to one subelement of the `input_arcs_valid` input port, where `input_arcs_valid` is defined in the input port map of TCIS. 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, defined in the output port map of the TCI, to the

¹The `output_arcs_valid` output port is a composite port, i.e. of the *array* type.

input_transitions_fired and output_transitions_fired input ports, defined in the input port map of the input and output PCIS. 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 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.

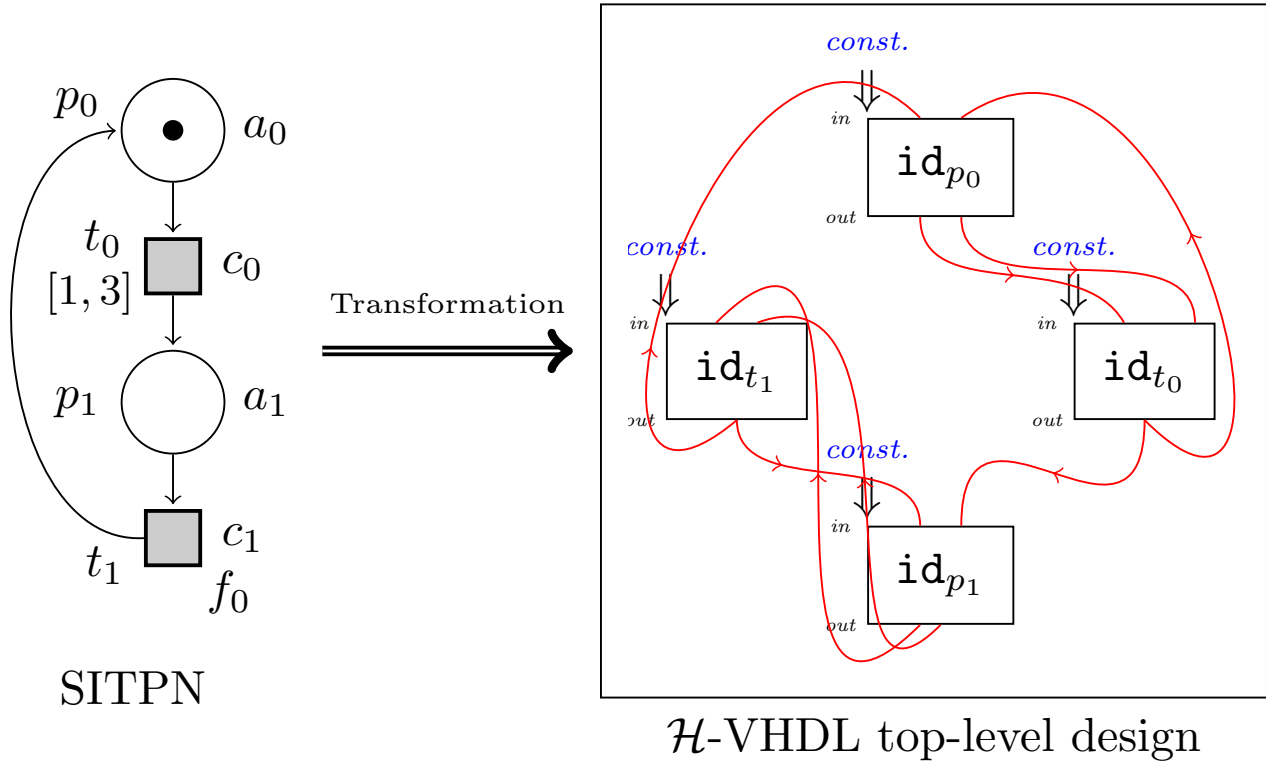


FIGURE 1.3: Generation of the interconnections between the place and transition component instances. In **red**, the internal signals interconnecting the PCIs and the TCIs. These signals are generated by the transformation. The arrows indicate the sense of propagation of the information. In **blue**, the constant values produced at the previous transformation step.

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 is connected to the input_conditions input port of TCIs. The interconnection of an input port of the top-level design to the input_conditions 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 the

action 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 output port belongs to the output port map of PCIs. Through the Boolean marked output port, the PCIs inform the outside about their marking status, i.e. if they possess at least one token or not. 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 output port belongs to the output port map of TCIs. Through the Boolean fired output port, the TCIs inform the outside about their firing status, i.e. if they are fired or not. Figure gives the top-level \mathcal{H} -VHDL design at the end of the transformation.

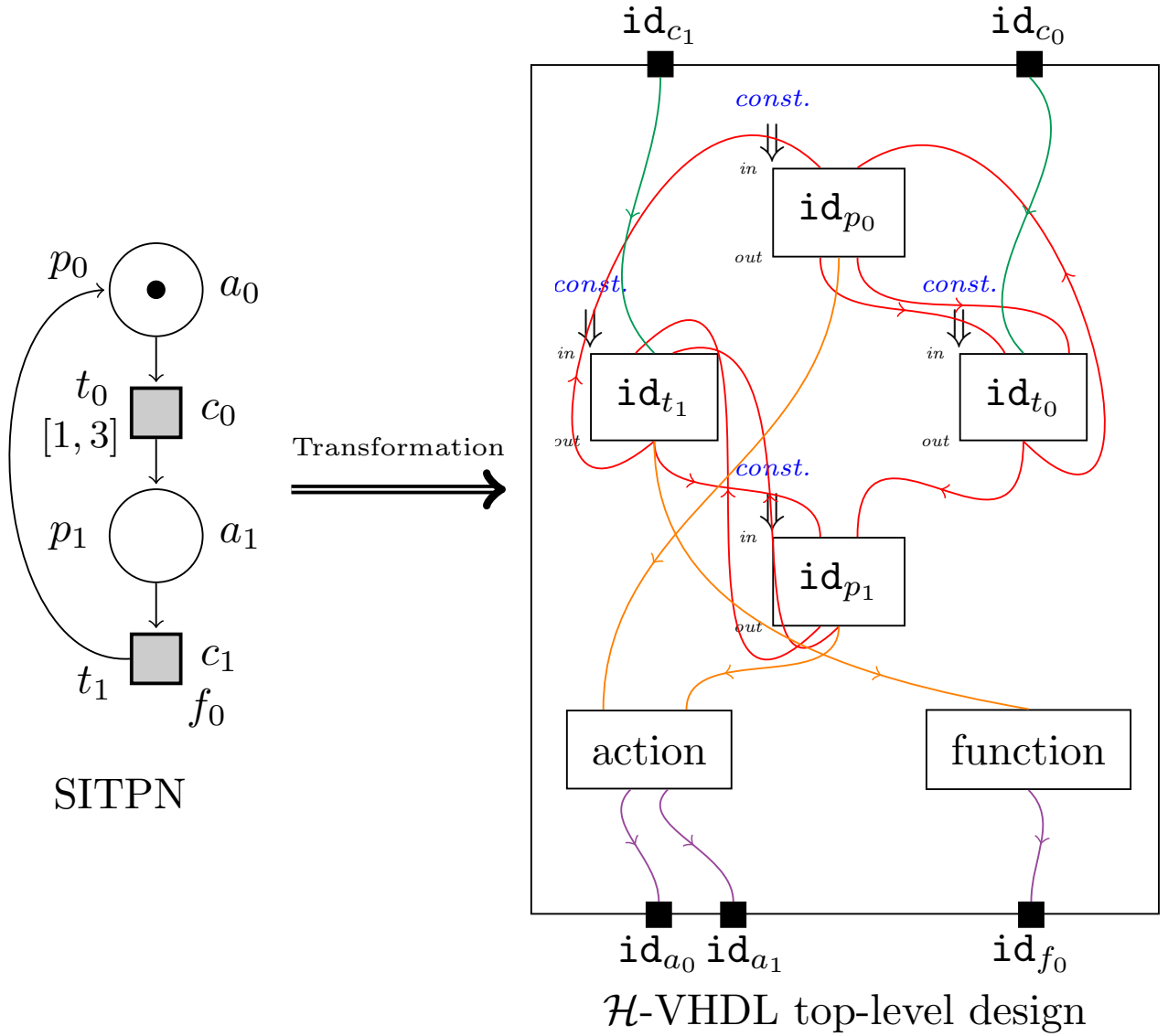


FIGURE 1.4: Generation of the input and output ports, and of the action and the function process in the \mathcal{H} -VHDL top-level design. The *primary* input port id_{c_1} (resp. id_{c_0}) implements the condition c_1 (resp. c_0). In **green**, the internal signals, generated by the transformation, connecting the input ports of the top-level design to the `input_conditions` input port of TCIs. The id_{a_0} and id_{a_1} output ports reflect the activation status of actions a_0 and a_1 . The id_{f_0} output port reflect the activation status of 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 assignment performed by the action and function processes 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). Especially, we are interested in two things:

1. Is there a proper way to build a transformation function? Do standards exist to do this 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?
2. In the context of formal verification, how are expressed the semantic preservation theorems? Are there usual proof strategies?

Here, the goal is to inspire ourselves with the work of the literature, and to see how far the correspondence holds between our specific case of transformation, and other cases of transformations. The results of the literature review are presented in two parts. The two parts have been prepared based on the same material. The first part will be focusing on the expression of the transformation functions in the literature, and the second part will be focusing on the proof that these transformations are semantic preserving ones.

The material we used for the literature review is divided in three categories. Each category covers a specific case of transformation function, always taken in the context of formal verification. Thus, the three categories are:

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

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? We open these to the more general context of transformation functions that map a source representation to target one.

Compilers for generic programming languages

In the context of formally verified compilers for generic programming languages, translation from a source program to a target program is mostly 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 CompCertproject [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. This is all the more natural, since the languages like Coq, Isabelle, HOLand other interactive theorem provers permit to perform pattern matching on the abstract syntax tree of the source programs.

The cases of optimizing compilers, like [11] and [16], show that to avoid to write too complex functions when passing from source to target program, that would too difficult to handle during the semantic preservation proof, 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 translation functions simple 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 is the expression of translation function is the necessity to keep a binding between source and 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 found in C programs to the registers generated in the RTL representation. The binding is important for the translation (replacing variables by their corresponding registers in the RTL code), and for the proof when values from the source program will be compared to values in the target program. This is a thing that should be anticipated when writing a translation function, i.e what is the correspondence between source and target elements.

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.

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 hardware 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 translation from VHDL to 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 describe the definition of the language FeSi (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 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 Koïka, which is also a simplification of the BlueSpec language. A Koïka 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 Koïka program is accompanied by a scheduler that specifies an execution order for the rules. The described compiler transforms Koïka 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 having two methods: *reset*, for the initialization of the streams, and *step*, for the update of values resulting of a synchronous step.

In [12], the authors describe a compiler that transforms Verilog programs into netlists targeting 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 in the HILECOP methodology, 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. Consequently, the 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. While giving an operational semantics to the DSL models, the aim of the transformation is to be able to compare the execution of the target models with the execution of the source models. The authors illustrate their approach with a source DSL named xSPeM, a process description language, and the model domain is timed PNs. The translation 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 $\langle s, c, t \rangle$ where the “correspondence model c explicitly stores correspondence relationships between source model s and target model t ”.

The work described in [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 describe 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 present 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.

Also, while translating 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 the 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 of the form of the input and the output representation. Compilers for programming languages tend to be a lot more compositional than model transformations. Compositional meaning 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, ... 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.

1.3 The transformation algorithm

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 embeds the generation of VHDL code from an 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_hvhd` 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). As it is generated along the transformation, the binder 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 $\langle PMap, TMap, CMap, AMap, FMap \rangle$ where:*

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

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 $e \in P \sqcup T \sqcup C \sqcup A \sqcup F$, we write $\gamma(e)$ where e is looked up in the appropriate function. For instance, for a given $f \in F$, $\gamma(f)$ is a shorthand for $FMap(f)$ where $\gamma = \langle \dots, FMap \rangle$.

Algorithm 1 is the algorithm of the HILECOP model-to-text transformation. The algorithm has 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; $mmf \in P \rightarrow \mathbb{N}$ 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, a function associated the places with a maximal marking value can be computed. Thus, the presence of the mmf function as a parameter of the $sitpn_to_hvhd1$ 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 the 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: $sitpn_to_hvhd1(sitpn, id_e, id_a, mmf)$

```

1  $d \leftarrow \text{design } id_e \ id_a \ \emptyset \ \emptyset \ \text{null}$ 
2  $\gamma \leftarrow \emptyset$ 
3  $\text{generate\_architecture}(sitpn, d, \gamma, mmf)$ 
4  $\text{generate\_interconnections}(sitpn, d, \gamma)$ 
5  $\text{generate\_ports}(sitpn, d, \gamma)$ 
6 return  $(d, \gamma)$ 
```

In Algorithm 1, Line 1 creates the \mathcal{H} -VHDL design. 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 $sitpn_to_hvhd1$ function has an empty set of generic constant, 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 detailed in Algorithms, the content of the `generate_interconnections` function is detailed in Algorithms, and the content of the `generate_ports` function is detailed in Algorithms.

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.

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

Every above set 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.

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

- $\text{output}_c(p)$. The function operates the following sequence:
 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 returns an empty set.
 2. otherwise, tries to establish a total ordering over the set of conflicting transitions of p w.r.t the firing priority relation:
 - raises an error 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).
 - returns the ordered set, with the top-level priority transition at the head.

- $\text{output}_{nc}(p)$. 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. $\text{output}(p)$ as defined above. Otherwise, the function returns the set of output transitions of p connected through a test or an inhib arc, i.e. $\{t \mid \exists \omega \text{ s.t. } \text{pre}(p, t) = (\omega, \text{test}) \vee \text{pre}(p, t) = (\omega, \text{inhib})\}$.
- $\text{cassoc}(\text{map}, \text{id}, x)$ 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 $(\text{id}(i), x)$ to the map structure. The index i is computed as follows based on the content of map :

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

Examples:

- $\text{cassoc}(((s(0), \text{true}), (s(1), \text{false})), s, \text{true})$ yields the resulting map $((s(0), \text{true}), (s(1), \text{false}), (s(2), \text{true}))$.
- $\text{cassoc}(((s(0), \text{true}), (s(1), \text{false})), a, \text{open})$ yields the resulting map $((s(0), \text{true}), (s(1), \text{false}), (a(0), \text{open}))$.
- $\text{get_comp}(\text{id}_c, \text{cstmt})$ where id_c is an identifier, and $\text{cstmt} \in \text{cs}$ 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 .
- $\text{put_comp}(\text{id}_c, \text{cistmt}, \text{cstmt})$ where id_c is an identifier, cistmt is a component instantiation statement, and $\text{cstmt} \in \text{cs}$ 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 composed with cstmt with the $||$ operator. The put_comp procedure throws an error if multiple CIS with identifier id_c exist in cstmt .
- $\text{actual}(\text{id}, \text{map})$ 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 $(\text{id}, a) \in \text{map}$. 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 .
- $\text{genid}()$. 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. The procedure takes an output port map o , an input port map i , two identifiers id_1, id_2 and a \mathcal{H} -VHDL design d as parameters. The procedure generates an internal signal id_s and adds it to the internal signal declaration list of design d . Then, the procedure connects a subelement of port id_1 (resp. id_2) to the internal signal id_s in the output port map o (resp. input port map i). As the result, a subelement of id_1 is connected to a subelement of id_2 through the internal signal id_s .

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

```

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

```

Complementary notations

When there is no ambiguity, id_p (resp. id_t) denotes the PCI (resp. TCI) identifier associated to 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 to 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. This is when 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 in the output port map of PCIs and TCIs during this first generation step. 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( $sitpn, d, \gamma$ )

```

Note that, in the following algorithms and explanations, we refer to the generic constant, internal signal and port identifiers defined in the place and transition designs through their

abbreviated names (see Table ??).

Algorithm 4: generate_PCIs(*sitpn*, *d*, γ , *mmf*)

```

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

```

The generate_PCIs procedure, presented in Algorithm 4, has four parameters: *sitpn* $\in \text{SITPN}$, the input SITPN model; *d* $\in \text{design}$, the \mathcal{H} -VHDL design being generated; $\gamma \in \text{WM}(\text{sitpn}, d)$, the binder between *sitpn* and *d*; *mmf* $\in P \rightarrow \mathbb{N}$, 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 . As said before, the only associations generated in the input and output port maps during this phase pertain to the association of ports to constant values. At the end of the procedure (Lines 28 to 30), a fresh and unique component identifier id_p is generated, and a new CI statement, corresponding to the instantiation of 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 .

From Line 2 to Line 27, the procedure generates the generic map, the input port map, and the output port map of the PCI. 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 generated PCI. 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, 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 `connect_input_trs` procedure during the generation of the interconnections between PCIs and TCIs.

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, 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 absence 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 `connect_output_trs` procedure during the generation of the interconnections between PCIs and TCIs.

From Line 23 to Line 27, the 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).

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

```

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

```

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 CI statement, corresponding to the instantiation of 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 .

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 generated TCI. The generic map g_t initially takes two associations: the one between the tt constant and the result of the function call $\text{get_ttype}(t)$, and the one between the mtc constant and the result of the function call $\text{get_mtc}(t)$. 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. 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 6: `get_ttype(t)`

```

1 if  $t \notin \text{dom}(I_s)$  then return NOT_TEMPORAL
2 else if  $I_s(t) = [a, a]$  then return TEMPORAL_A_A
3 else if  $I_s(t) = [a, b]$  then return TEMPORAL_A_B
4 else if  $I_s(t) = [a, \infty]$  then return TEMPORAL_A_INFINITE

```

Algorithm 7: `get_mtc(t)`

```

1 if  $t \notin \text{dom}(I_s)$  then return 1
2 else if  $I_s(t) = [a, b]$  then return  $b$ 
3 else if  $I_s(t) = [a, \infty]$  then return  $a$ 

```

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 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 absence 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, γ)`

```

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

```

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. `actual(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 succeeds, 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 .*

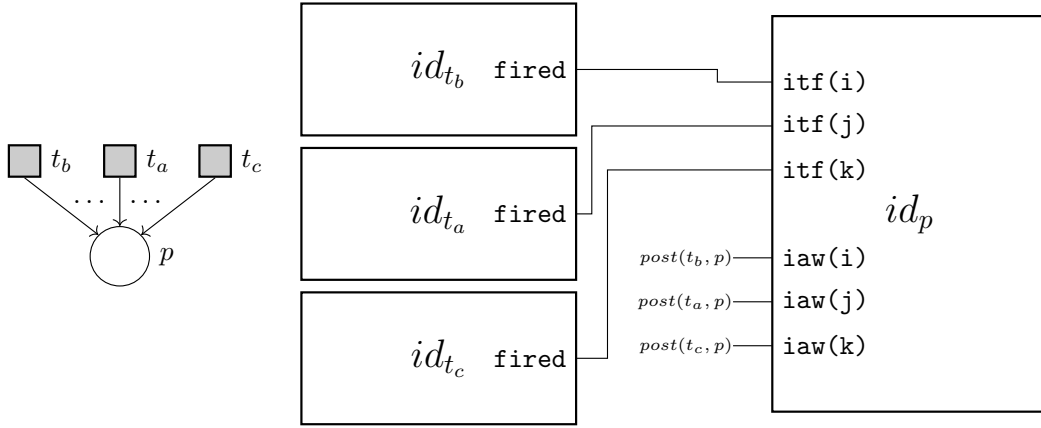


FIGURE 1.6: An example of correct connections between the PCI id_p and TCIs id_{t_a} , id_{t_b} and id_{t_c} . On the left, the input SITPN model showing the connections of the transitions t_a , t_b and t_c to the place p . The dots indicate that the place p possibly has other input transitions. On the 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 `actual(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 `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 similarly 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). Also, one subelement of the pauths input port is associated with true in the input port map i_t (Line 25).

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 presents the generate_ports procedure. This procedure calls three procedures, namely: the generate_conditions 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 detailed in Algorithms 10, 11 and 12.

Algorithm 9: generate_ports(*sitpn*, *d*, γ)

```

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

```

Algorithm 10 describes the generate_conditions procedure.

Algorithm 10: generate_conditions(*sitpn*, *d*, γ)

```

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

```

The generate_conditions procedure iterates over the set of conditions of the *sitpn* parameter. For each condition of the set, the generate_conditions 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, γ)`

```

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

```

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 statements. Then, in the absence 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, γ)`

```

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

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 function process, the fired 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_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 `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 statements. 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 `sitpn_to_hvhd1` function presented in Algorithm 1 has been implemented in Coq by the function `sitpn_to_hvhd1` function presented in Listing 1.1.

```

1 Definition sitpn_to_hvhd1 (sitpn : Sitpn)
2   (decpr : forall x y : T sitpn, {pr x y} + {~pr x y})
3   (id_e id_a : ident) (mmf : P sitpn → nat) :
4   (design * Sitpn2HVhdlMap sitpn) + string :=
5   RedV
6   ((do _ ← generate_sitpn_infos sitpn decpr;
7     do _ ← generate_architecture sitpn mmf;
```

```

8      do _ ← generate_ports sitpn;
9      do _ ← generate_comp_insts sitpn;
10     generate_design_and_binder  $id_e$   $id_a$ )
11     (InitS2HState sitpn Petri.ffid)).

```

LISTING 1.1: The Coq implementation of the `sitpn_to_hvhd1` function.

In Listing 1.1, the `sitpn_to_hvhd1` 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 identifiers for the generated \mathcal{H} -VHDL; the `mmf` function that maps places of `sitpn` to a maximal marking value. The `sitpn_to_hvhd1` function returns a couple composed of the generated \mathcal{H} -VHDL design, of type `design`, and the generated γ binder, of type `Sitpn2HVhd1Map sitpn`; or, the `sitpn_to_hvhd1` function returns a string corresponding to an error message.

In the body of the `sitpn_to_hvhd1` 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. Thus, the `do` sequence defined in the body of the `sitpn_to_hvhd1` function constitutes a single monadic function. Here, the `RedV` notation reduces the result of the application of the monadic function to the parameter `(InitS2HState sitpn Petri.ffid)`, i.e. the initial compile-time state. 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 value returned by these function calls. 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 compile-time state that is carried and modified through the multiple function calls composing the transformation. Listing 1.2 gives the implementation of the compile-time state structure.

```

1  Record Sitpn2HVhd1State (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      $\gamma$  : Sitpn2HVhd1Map sitpn;

```

```

15
16   }.

```

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

The compile-time 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 are the list versions of the finite sets of places, transitions, conditions, actions and functions of the `sitpn` parameter. The fields are built 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 permits to generate fresh and unique identifiers all along the transformation. The `sitpninfos` field 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 gathers the list of input port declarations of the generated \mathcal{H} -VHDL design. The `arch` 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 is the behavior of the generated \mathcal{H} -VHDL design. The γ field is the SITPN-to- \mathcal{H} -VHDL binder also generated alongside the \mathcal{H} -VHDL design, and returned at the end of the transformation.

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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/10.1016/j.jss.2014.02.058). URL: <http://www.sciencedirect.com/science/article/pii/S0164121214000727> (visited on 01/16/2020).