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

En Informatique

École doctorale : Information, Structures, Systèmes

Unité de recherche LIRMM

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

Présenté par Vincent IAMPIETRO

Le Date de la soutenance

Sous la direction de David Delahaye
et David Andreu

Devant le jury composé de

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

Acknowledgements

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

Contents

List of Figures

List of Tables

List of Abbreviations

SITPN	**S**ynchronously executed **I**nterpreted **T**ime **P**etri **N**et with priorities
VHDL	**V**ery high speed integrated circuit **H**ardware **D**escription **L**anguage
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

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...
- Pciples 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.

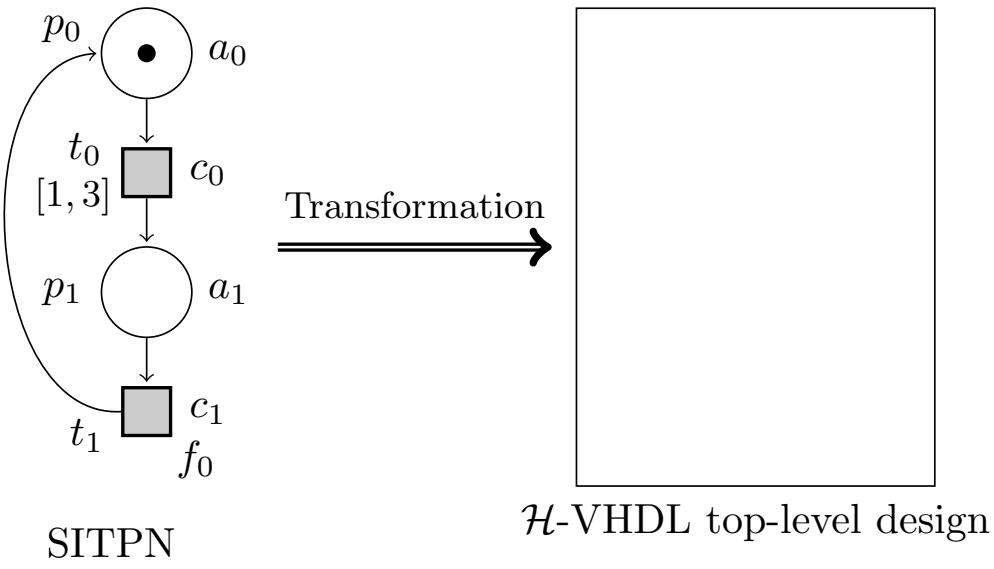

FIGURE 1.1: Transformation of an input SITPN model into a top-level $\mathcal{H}$-VHDL design. The input model is composed of two places, p_0 and p_1, and two transitions, t_0 and t_1. Transition t_0 is associated with the time interval $[1,3]$ and the condition c_0. Transition t_1 is associated with 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.

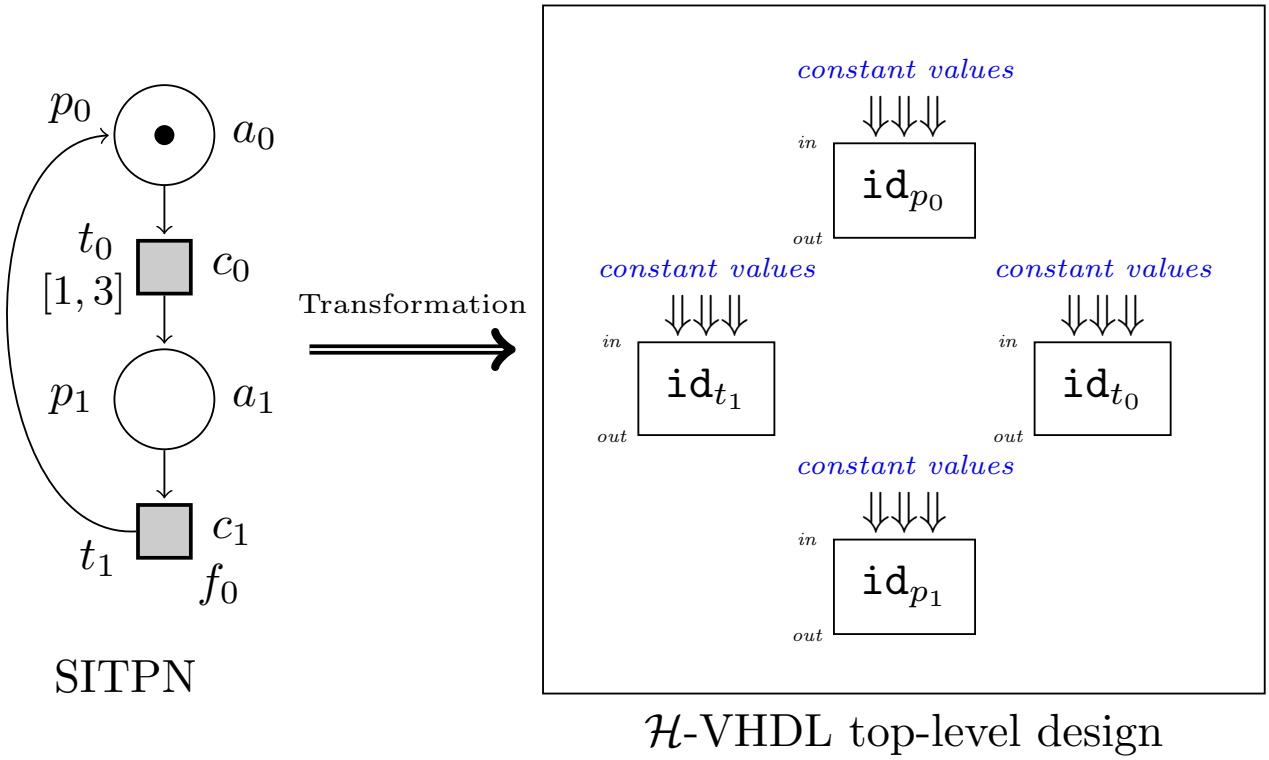

FIGURE 1.2: Generation of the `place` and `transition` component instances based on the set of places and transitions of the input SITPN. The `PCI` $\texttt{id}_{p_0}$ implements the place p_0, TCI $\texttt{id}_{t_0}$ the transition t_0,...

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 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 port[1], 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

[1]The `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.

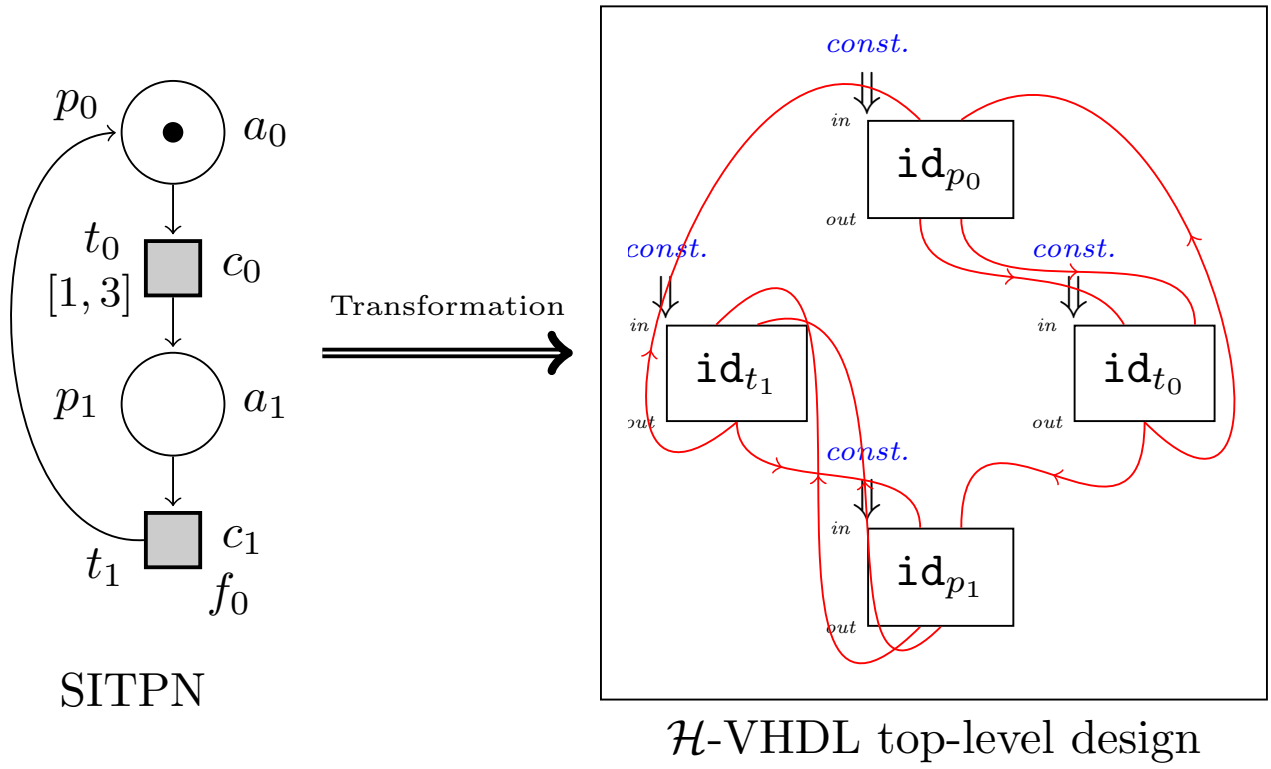

FIGURE 1.3: Generation of the interconnections between the `place` and `transition` component instances. In red, the internal signals interconnecting the PCIs and the TCIs. These signals are generated by the transformation. The arrows indicate the sense of propagation of the information. In blue, the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

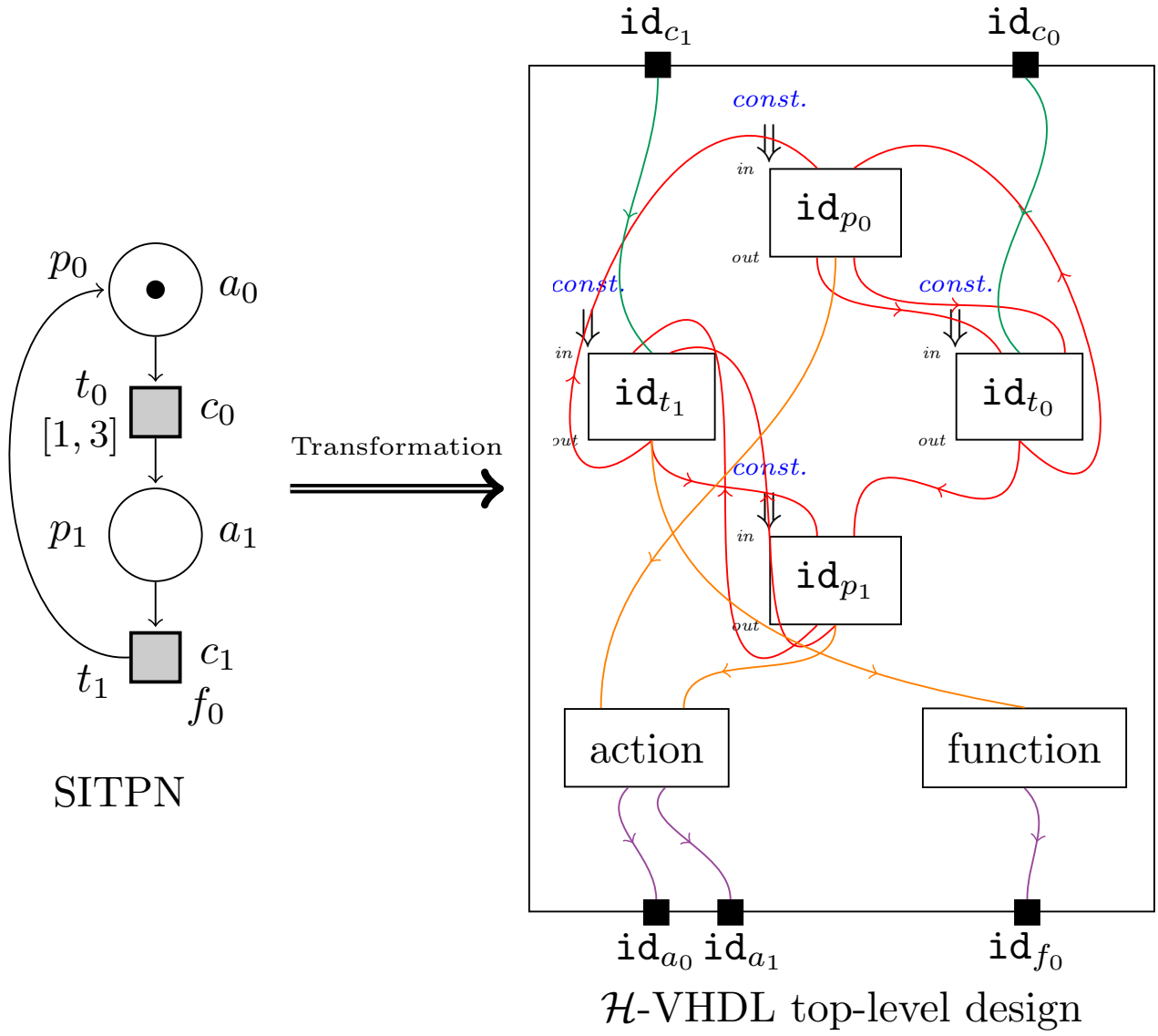

FIGURE 1.4: Generation of the input and output ports, and of the `action` and the `function` process in the $\mathcal{H}$-VHDL top-level design. The *primary* input port $\mathtt{id}_{c_1}$ (resp. $\mathtt{id}_{c_0}$) implements the condition c_1 (resp. c_0). In green, the internal signals, generated 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 actions a_0 and a_1. The $\mathtt{id}_{f_0}$ 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 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 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 Koîka, which is also a simplification 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 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 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 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 $<s, c, t>$ 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 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 [17], 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 is 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_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). 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 $<PMap, TMap, CMap, AMap, FMap>$ where:*

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

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, 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_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 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_hvhdl`$(sitpn, id_e, id_a, mmf)$

```
1 d ← design id_e id_a ∅ ∅ ∅ null
2 γ ← ∅
3 generate_architecture(sitpn, d, γ, mmf)
4 generate_interconnections(sitpn, d, γ)
5 generate_ports(sitpn, d, γ)
6 return (d,γ)
```

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_hvhdl` 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 detailled in Algorithms, the content of the `generate_interconnections` function is detailled in Algorithms, and the content of the `generate_ports` function is detailled 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.

- $\texttt{input}(p) = \{t \mid \exists \omega \text{ s.t. } post(t,p) = \omega\}$, the set of input transitions of a place p.
- $\texttt{output}(p) = \{t \mid \exists \omega, a \text{ s.t. } pre(p,t) = (\omega, a)\}$, 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.
- $\texttt{input}(t) = \{p \mid \exists \omega, a \text{ s.t. } pre(p,t) = (\omega, a)\}$, the set of input places of a transition t.
- $\texttt{output}(t) = \{p \mid \exists \omega \text{ s.t. } post(t,p) = \omega\}$, 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 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.

- $\texttt{output}_c(p)$. 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.

- $\texttt{output}_{nc}(p)$. 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 $\texttt{output}(p)$ as defined above. Otherwise, the function returns the set of output transitions of p connected through a `test` or an `inhib` arc, i.e. $\{t \mid \exists \omega \; s.t. \; pre(p,t) = (\omega, \texttt{test}) \vee pre(p,t) = (\omega, \texttt{inhib})\}$.

- $\texttt{cassoc}(map, id, x)$ 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:

  - `cassoc( ((s(0),true),(s(1),false)), s, true)` yields the resulting map `((s(0),true),(s(1),false),(s(2),true))`.
  - `cassoc( ((s(0),true),(s(1),false)), a, open)` yields the resulting map `((s(0),true),(s(1),false),(a(0),open))`.

- $\texttt{sassoc}(map, id, x)$ 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 `sassoc` procedure adds an association of the form (id, x) to the map structure.

- $\texttt{get\_comp}(id_c, cstmt)$ where id_c is an identifier, and $cstmt \in$ 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 ecist 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 $cstmt \in$ 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 $cstmt$.

- $\texttt{actual}(id, map)$ 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.

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 2 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: `generate_PCIs` generates the PCIs in the behavior of d, and binds the PCI identifiers to places in the γ binder; `generate_TCIs` generates the TCIs in the behavior of d, and binds the TCI identifiers to transitions in the γ binder. Note that in the following algorithms, the generic constant, internal signal and port identifiers defined in the `place` and `transition` designs are referred to by their alias names (see Table **??**).

Algorithm 2: `generate_architecture`$(sitpn, d, \gamma, mmf)$

```
1 generate_PCIs(sitpn, d, γ, mmf)
2 generate_TCIs(sitpn, d, γ)
```

Algorithm 3 presents the content of the `generate_PCIs` procedure. The procedure 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 P of the $sitpn$ parameter. At Line 2, the procedure checks if place p is isolated, i.e. without input nor output transitions. An error, with an associated message, is raised with the `err` primitive if one place of $sitpn$ is isolated. 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. Then, for each place in set P, the procedure generates a corresponding PCI.

Line 3 builds the generic map of the PCI. The three generic constants defined in the generic clause of the `place` design receive a value. In the generated generic map, the `input_arcs_number` (resp. `output_arcs_number`) generic constant is associated with 1 if the set of input (resp. output) transitions of p is empty, or with the size of the set otherwise. The `maximal_marking` constant is associated with the value returned by the mmf function for the place p.

From Lines 4 to 16, the procedure partially builds the input port map of the PCI; the generated associations only pertain to the input ports associated with constant values. At Line 7, each subelement of the `input_arcs_weights` input port is associated with the weight of the arc between place p and one of its input transition. For example, let us take a place p, and $\{t_0, t_1, t_2\}$, the set of input transitions of p; the **foreach** loop of Line 7 generates the following input port map: $((\texttt{iaw}(0), post(t_0, p)), (\texttt{iaw}(1), post(t_1, p)), (\texttt{iaw}(2), post(t_2, p)))$. Line 5 deals with the particular case where the set of input transitions of p is empty. From Lines 13 to 16,

each subelement of the `output_arcs_weights` (resp. the `output_arcs_types`) input port is associated with the weight (resp. type) of the arc between place p and one of its output transition. Line 9 deals with the case where the set of output transitions of p is empty.

From Lines 17 to 22, the procedure connects the `marked` output port in the output port map o_p. In the case where place p is not associated with any action (Line 18), the `marked` output port is connected to the `open` keyword, i.e. not connected. Otherwise, the `marked` output port is connected to a newly generated internal signal of the Boolean type. This connection will be used later, during the generation of the `action` process (see Section 1.3.5).

Finally, from Lines 23 to 25, the procedure generates a fresh component identifier for place p, builds a new PCI that holds the previously generated generic map, input and output port maps, and adds the PCI to the behavior of design d by composition with the `||` operator. Line 25 adds the binding between place p and the PCI identifier id_p to the γ binder.

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

1 **foreach** $p \in P$ **do**
2 **if** $\texttt{input}(p) = \emptyset$ *and* $\texttt{output}(p) = \emptyset$ **then** `err`("p *is an isolated place*")
3 $g_p \leftarrow ((\texttt{ian}, \begin{cases} 1 \text{ if } \texttt{input}(p) = \emptyset \\ |\texttt{input}(p)| \text{ otherwise} \end{cases}), (\texttt{oan}, \begin{cases} 1 \text{ if } \texttt{output}(p) = \emptyset \\ |\texttt{output}(p)| \text{ otherwise} \end{cases}), (\texttt{mm}, mmf(p)))$
4 $i_p \leftarrow ()$
5 **if** $\texttt{input}(p) = \emptyset$ **then** $\texttt{cassoc}(i_p, \texttt{iaw}, 0)$
6 **else**
7 **foreach** $t \in \texttt{input}(p)$ **do**
8 $\texttt{cassoc}(i_p, \texttt{iaw}, post(t, p))$
9 **if** $\texttt{output}(p) = \emptyset$ **then**
10 $\texttt{cassoc}(i_p, \texttt{oaw}, 0)$
11 $\texttt{cassoc}(i_p, \texttt{oat}, \texttt{basic})$
12 **else**
13 **foreach** $t \in \texttt{output}_c(p) \cup \texttt{output}_{nc}(p)$ **do**
14 $(\omega, a) \leftarrow pre(p, t)$
15 $\texttt{cassoc}(i_p, \texttt{oaw}, \omega)$
16 $\texttt{cassoc}(i_p, \texttt{oat}, a)$
17 $o_p \leftarrow ()$
18 **if** $\texttt{acts}(p) = \emptyset$ **then** $\texttt{sassoc}(o_p, \texttt{marked}, \texttt{open})$
19 **else**
20 $id_s \leftarrow \texttt{genid}()$
21 $d.sigs \leftarrow d.sigs \cup (id_s, \texttt{boolean})$
22 $\texttt{sassoc}(o_p, \texttt{marked}, id_s)$
23 $id_p \leftarrow \texttt{genid}()$
24 $d.cs \leftarrow d.cs \,||\, \texttt{comp}(\texttt{id}_p, \texttt{place}, g_p, i_p, o_p)$
25 $\gamma \leftarrow \gamma \cup (p, \texttt{id}_p)$

Algorithm 6 presents the content of the `generate_TCIs` procedure. The procedure iterates over the set of transitions T of the $sitpn$ parameter, and generates a corresponding TCI for

every transition $t \in T$. At Line 2, the procedure checks if transition t is isolated, i.e. without input nor output places.

Line 3 builds the generic map of the TCI. The `transition_type` generic constant is associated with the result of the function call `get_ttype`(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 to t. Algorithm 4 describes function `get_ttype`.

Algorithm 4: `get_ttype`(t)

```
1 if t ∉ 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, ∞] then return TEMPORAL_A_INFINITE
```

The `maximal_time_counter` generic constant is associated with the result of the function call `get_mtc`(t). The `get_mtc` function determines the maximal value for the time counter based on the form of the time interval associated with transition t. Algorithm 5 describes function `get_mtc`.

Algorithm 5: `get_mtc`(t)

```
1 if t ∉ dom(I_s) then return 1
2 else if I_s(t) = [a, b] then return b
3 else if I_s(t) = [a, ∞] then return a
```

The `input_arcs_number` (resp. `conditions_number`) generic constant is associated with 1 if the set of input places (resp. conditions) of t is empty, or with the size of the set otherwise.

From Lines 4 to 7, the procedure builds the input port map of the TCI. Line 5 deals with the particular case where the set of conditions of t is empty; in that case, the $(\texttt{ic}(0), 0)$ association is appended to the i_t map. In the case where the set of conditions of t is not empty, the `genarate_conds` procedure, presented in Algorithm 7, will handle the connection of the `input_conditions` input port in the input port map of TCIs. Line 6 adds the association $(\texttt{A}, 0)$ to the i_t map if t is not a time transition (i.e. t has no associated time interval, it is not in the domain of function I_s), or $(\texttt{A}, upper(I_s(t)))$ otherwise. Line 7 adds the association $(\texttt{B}, 0)$ to the i_t map if t is not a time transition or has an infinite upper bound. Otherwise, the association $(\texttt{B}, upper(I_s(t)))$ joins the i_t map.

From Lines 9 to 11, the procedure connects the `fired` input port to a newly generated internal signal in the output port of the TCI. 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). At Line 12, if the set of input places of t is empty, then the subelement of index 0 of the `reinit_time` input port must be connected to the `fired` output port of the same TCI through the newly generated internal signal id_s. This connection represents the fact that, in the absence of input places, resetting the value of a transition's time counter only depends on the firing status of the transition.

Finally, from Lines 13 to 15, the procedure generates a fresh component identifier for transition t, builds a new TCI that holds the previously generated generic map, input and output port maps, and adds the TCI to the behavior of design d by composition with the `||` operator.

Line 15 adds the binding transition t and the TCI identifier id_t to the γ binder.

Algorithm 6: `generate_TCIs`$(sitpn, d, \gamma)$

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

1.3.4 Interconnection of the component instances

Remark 2 (Indices consistency). *We remind the reader of the fact that a* ***foreach*** *loop always iterates in the same order over the elements of a set. When generating the input and output port maps of PCIs, there must be some kind of correspondence between the subelements of two ports, when the subelements share the same index. If the subelement of index 0 in the* `input_arcs_weights` *input port (i.e.* $\texttt{iaw}(0)$*) is associated with the weight of the arc* (t_0, p)*, then, during the generation of interconnections, the subelement* $\texttt{itf}(0)$ *will be connected to the* `fired` *port of the TCI implementing transition* t_0*. The subelement of index 0 in the* `iaw` *and* `itf` *input ports both refer to transition* t_0*. This is why a* ***foreach*** *loop must always iterates in the same order over the elements of a set to preserve the consistency between the indices of input and output ports.*

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

Algorithm 7: `generate_conds`$(sitpn, d, \gamma)$

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

➔ talk about the state-and-error monad, the γ binder, the use of generic list functions (fold, map, filter, iter…)

CONCLUSION OF THE CHAPTER: Our transformation function is not really expressible in a compositional way as P and T instances are bound together. It's a specific case of model-to-text transformation. However, as much as the comparison with other works of transformation verification holds, we have been trying to inspire ourselves from the literature to implement the HILECOP model-to-text transformation function in Coq.

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] 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).