UNIVERSITY NAME

DOCTORAL THESIS

Thesis Title

Author:
John SMITH

Supervisor:
Dr. James SMITH

A thesis submitted in fulfillment of the requirements
for the degree of Doctor of Philosophy

in the

Research Group Name
Department or School Name

May 10, 2021

"Thanks to my solid academic training, today I can write hundreds of words on virtually any topic without possessing a shred of information, which is how I got a good job in journalism."

Dave Barry

UNIVERSITY NAME

Abstract

Faculty Name
Department or School Name

Doctor of Philosophy

Thesis Title

by John SMITH

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

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

For/Dedicated to/To my…

Chapter 1

Proving semantic preservation in HILECOP

- Change σ_{injr} and σ_{injf} into σ_i.
- Define the $\texttt{Inject}_{\downarrow}$ and $\texttt{Inject}_{\uparrow}$ relations.
- Keep the $sitpn$ argument in the SITPN full execution relation, but remove it from the SITPN execution, cycle and state transition relations.
- Make a remark on the differentiation of boolean operators and intuitionistic logic operators
- Explain and illustrate the equivalence relation between SITPN and VHDL.
- Talk about the correspondence between combinational signal value and there assignment expression deduced from the code. Explain that this is where the $\mathcal{H}$-VHDL semantics plays it part in the proof; although we are not detailling how assignment expressions are deduced from running the semantics of the $\mathcal{H}$-VHDL code. Give some examples of correspondence between combinational signal value and assignment expressions.

1.1 Semantic preserving transformations in the literature

In this section, we present the review of the work done in the literature pertaining to transformation functions with a focus 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 representation 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

1.1.1 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 1.1 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 1.1: 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-to-model and model-to-text 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.

1.1.2 Transformations and proofs of semantic preservation

In this section, we are interested in how to prove that transformation functions are semantic preserving. Especially, we are interested in the expression of semantic preservation theorems, i.e, what does one mean by semantic preservation, and in seeking usual proof strategies.

In the introduction of his article about CompCert [11], X.Leroy presents the two points of major importance to express semantic preservation theorems for GPL compilers, and more generally to get the meaning of semantic preservation.

The first point is to clearly state how things are compared between the source and the target programs. It is to describe the runtime state of the source and the target, and to draw a correspondence between two. This is expressed through a state comparison relation.

The second point is to relate the execution of the source program to the execution of the target program through a simulation diagram. Figure shows the different kind of simulation diagrams possibly relating two programs. Choosing an adequate simulation diagram to express a semantic preservation theorem depends on the kind of possible behaviors that can exhibit a given program. In the case of GPL programs, X.Leroy lists three kinds of possible behaviors: either the program execution succeeds and returns a value, or the program execution fails and returns an error, or the program execution diverges.

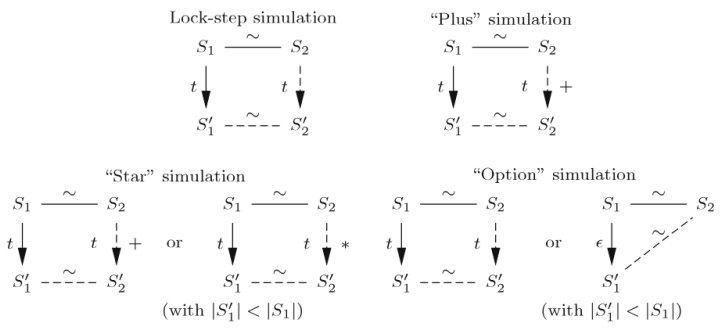

FIGURE 1.2: Simulation diagrams between source and target programs

Anyway, in the case where the source program execution succeeds, the theorem of semantic preservation takes this general form:

Consider a source program P_1 compiled into a target program P_2, a starting state S_1 for program P_1 and a starting state S_2 for program P_2 such that S_1 and S_2 are similar states w.r.t. the exhibited state comparison relation. If the execution of P_1 leads from state S_1 to state S'_1, then there exists a state S'_2 resulting of the execution of program P_2 from state S_2 such that S'_1 and S'_2 are similar w.r.t. the exhibited state comparison relation.

Compiler verification tasks tend to prove the kind of theorem stated above. The other kind of task that can be applied to certify a compiler is to perform compiler validation. Compiler validation is interested in generating a proof of behavior preservation (or a counter-example showing that behaviors diverge) for a given input program alongside the compilation process. Thus, for a given input program, the compiler yields a target program and the proof that the input and target have the same behavior. Exhibiting a theorem of semantic preservation is stronger than building a proof of semantic preservation for each input program. Therefore, compiler verification is stronger than compiler validation. In the aim of the thesis is to perform compiler *verification* over the HILECOP methodology. A few works presented in this section afterwards perform compiler validation, but are presented here for the sake of coverage.

1.2 Preliminary Definitions

1.3 Behavior Preservation Theorem

1.4 Initial States

1.5 First Rising Edge

1.6 Rising Edge

1.7 Falling Edge

1.8 A detailled proof: equivalence of fired transitions

Appendix A

Reminder on natural semantics

Appendix B

Reminder on induction principles

- Present all the material that will be used in the proof, and that needs clarifying for people who do not come from the field (e.g, automaticians and electronicians)
  - structural induction
  - induction on relations
  - ...

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).