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 12, 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 $\sigma_{injr}$ and $\sigma_{injf}$ into $\sigma_i$.
- 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 its 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 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

### 1.1.1 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 [12], 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, or bisimulation, 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.

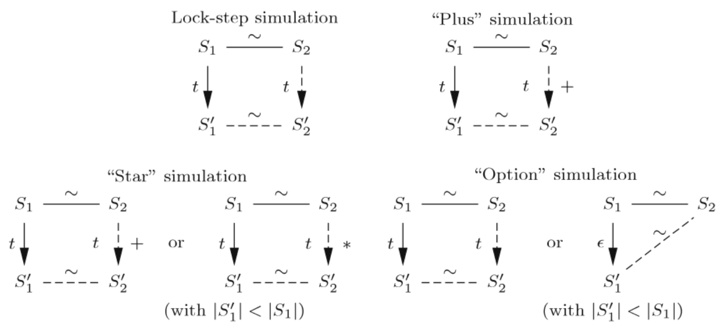

FIGURE 1.1: Simulation diagrams between source and target programs

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 aims at proving 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. The aim of the thesis is to perform compiler *verification* over the HILECOP methodology. Some of the works, cited afterwards, are more interested in compiler or transformation validation techniques than in verification. They are presented here for the sake of coverage.

Now that we have clarified the meaning of semantic preservation for GPL compilers, we state that this definition of semantic preservation holds also for more general case of transformation from a source representation to a target representation. The only condition to be able to verify that a transformation is semantic preserving is that the source and target representation must have an execution semantics (i.e, the instances of the source and target representations must be executable).

For each article used in the literature review and presenting a specific case of transformation, the following questions have been asked:

- What are the similarities/differences between source and target representations?
- How are described the runtime state for the source and target representations?
- How is expressed the state comparison relation?
- How is stated the semantic preservation theorem?
- What is the employed proof strategy?

**Compilers for generic programming languages**

Taking the CompCert compiler as an example, the compilation pass from Clight programs to Cminor programs is described in [2, 12]. Clight is a subset of the C language, and Cminor is a low-level imperative language. The two languages are endowed with a big-step operational semantics. Here, the execution state of the source and target languages are memory models (of course, we are dealing with programming languages). The memory model is the same for all intermediate language involved in the CompCert compiler. The memory model consists in block references; each block has a lower and an upper bound. To access a data, one has to specify the block reference along with the size of the accessed data (i.e, the data type) and the offset from the start of the block reference (i.e, where to begin the data reading). About the proof of semantic preservation, the most difficult point is to relate the memory state sof the source program to the memory states of the target program. To do so, the authors define a *memory injection* relation that binds the values of source and target together. They also establish a relation to compare execution environments, i.e, the environments holding the declaration of functions, global variables. . . The proof of semantic preservation is built incrementally: the authors prove a simulation lemma for the Clight expressions, then for the Clight statements, and finally for the entire Clight program. The proof strategy is to reason by induction over the evaluation relation of the Clight programs, and to perform case analysis on the translation function.

The pattern to compiler verification for GPLs is more or less the same as presented above. May it be compilers for imperative languages [12, 15], or compilers for functional languages [7, 17], compiler verfication proceeds as follows:

1. establish a relation between the memory models of the source and target languages, and between the global execution environments

2. prove simulation lemmas starting from simple constructs, and building up incrementally to consider entire programs

3. reason by induction over the evaluation relation of the source language, and the translation function

Relating memory models is more difficult when the gap between the source and target languages is important (for instance, the translation of Cminor programs into RTL programs in [12]). As a consequence, the complexity of the relation for memory model comparison increases.

**Compilers for hardware description languages**

In the case of HDL compilers, proving semantic preservation is very similar to the case of GPL compilers. Of course, the difference lies in the semantics of HDL languages, and in the description of execution states. The semantics of HDLs is intrinsically related to the notion of execution over time, or over multiple clock cycles; indeed, we are dealing with reactive systems. Therefore, the semantic preservation theorems are formulated w.r.t. the synchronous or time-related semantics of the considered languages.

In [3, 5], the source languages are a subset of the BlueSpec specification language for hardware synthesis , and the target is an RTL representation of the circuit. The execution states of the source and target are based on registers. In [3], the execution state also hold a log of the read and write operations of the input program, and this log is compared to the log of the RTL representation. The semantic preservation theorem states that the registers hold the same values after the execution of source program and the resulting RTL circuit after one clock cycle.

In [4], the source language is a subset of Lustre and the target language is imperative language called Obc. A Lustre program is composed of nodes; each node treats a set of input streams and publishes output streams after the computation of its statement body. In its statement body, a Lustre node possibly refer to instances of other nodes. In the compilation process, each Lustre node is translated into an Obc class. An Obc class hold a vector of variables composing its internal memory and a vector of other Obc class instances. The authors define a data flow semantics for the Lustre language; judgments of the semantics describe how output streams are computed based on input streams. Also, as we are dealing with hardwares, the judgments treat synchronous statements and combinational ones. On the side of the Obc language, the semantics define a function *step* that computes the execution the Obc classes over one clock cycle. To prove the semantic preservation theorem, the state comparison relation binds the values of input and output streams on one side to the values of variables and Obc class instances on the other side. The semantic preservation theorem is as follows: if a Lustre node yields output streams $o$ from input streams $i$, then the iterative execution of the *step* function for the corresponding Obc class builds every step of output streams $o$ given the values of input streams $i$. The proof is done by induction over the clock step count, and by induction over the evaluation derivation of the nody instruction body.

In [13], the HDL compiler translates Verilog modules into netlists. The execution state of Verilog module holds the value of the variables declared in the module. The execution state of a netlist circuit holds the value of the registers declared in the circuit. Therefore, the state comparison relation used to state the semantic preservation theorem binds the values of variables on one side to the values of registers on the other side. The semantics of Verilog resembles the one of VHDL; the set of processes composing a module are executed w.r.t. the simulation semantics of the language, i.e, composed of synchronous and combinational execution steps. The semantics of netlists is set as a big-step operational semantics by means of an interpreter that runs a netlist list over n clock cycles. The semantic preservation theorem is as follows: Assuming that a module is transformed into a circuit, and that some well-formation hypotheses hold on the module, if the module executes without error, and yields a final state *venv*, then there exists a final state *cenv* yielded by the execution of the circuit over n clock cycles s.t. *venv* and *cenv* are similar according to the relation *verilog_netlist_rel*. Here, the *verilog_netlist_rel* is the state comparison relation.

In [19], the compiler transforms programs of the synchronous language SIGNAL into Synchronous Clock Guarded Actions programs (S-CGA programs). A SIGNAL program describes a set of processes; each process holds a set of equations describing the relation between signals. The equations can be synchronous equations (refering to a clock) or combinational ones. An S-CGA program defines a set of actions to be applied to some variables when some conditions (the guards) are met. The SIGNAL (resp. the S-CGA) language has been endowed with a trace semantics describing the computation of signal values (resp. variable values) over time. The authors describe a function to translate the traces of SIGNAL and S-CGA programs into a common trace model. Thus, the semantic preservation theorem is stated by comparing two traces of execution defined through the same model. The proof of the semantic preservation theorem is built incrementally. For each statement of a SIGNAL process, the authors exhibit a lemma proving that the trace resulting from the execution of the statement is equivalent to the trace resulting of the execution of the corresponding guarded actions (obtained through the compilation). The proof is fully mechanized within the Coq proof assistant.

In [11], the authors verify a methodology to design hardware models with SystemC models. SystemC models describe hardware with modules; a module is a C++ class with ports, data members and methods. The methodology describes a transformation from SystemC into Abstract State Machine (ASM) thus enabling to model-check the hardware models. ASMs are described in the language AsmL; in AsmL, an ASM is implemented by a class with data members and methods. A denotational (fixpoint) semantics for SystemC modules is defined along with a denotational semantics for AsmL. The semantics is another variant of simulation cycle, similar to all other synchronous languages. There are two phases: evaluate and update and the gap between the two is called a delta-delay. The execution state of a SystemC module is divided into a signal store, mapping signal to value, and a variable store, mapping variable to value. The execution state of an AsmL class is only composed of a variable store. The theorem of semantic preservation states that, after translation, a SystemC model has the same *observational* behavior than its corresponding AsmL class. What is compared between a SystemC model and its corresponding AsmL class through their observational behavior is the activity of the processes of the first one and the activity of the methods of the second one. Processes and methods must be active at the same delta cycles. Therefore, what is compared here are not the values that the execution states hold, but rather the activity of the source and target programs.

### Model transformations

Regarding model transformations, a lot of works consider semantic preservation as the preservation of structural properties in the transformed model [1, 6, 14].

Still, there are many cases where the source model and th target one have both an execution semantics. In these cases, the authors are interested in proving that the transformation is semantic preserving by showing that the computation of the source model and the target model follow a simulation relation (see Figure 1.1).

In [8] and [18], the authors are interested in giving a translational semantics to a given model having itself a reference execution semantics. In [8], the source models are called xSpem models; they describe a set of activities exchanging resources and an holding an internal state. The target models are PNs. Both xSpem models and PNs have a state transition semantics. The state comparison is performed by checking the correspondence between each current status of the activities describe in an xSpem model and the marking of the PN. Then, the authors prove a bisimulation theorem, illustrated in Figure 1.2.

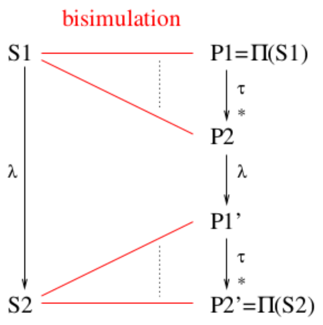

FIGURE 1.2: Bisimulation diagram relating an xSpem model execution and a Petri net execution

In Figure 1.2, one the right side of the diagram, i.e, the Petri net side, one can see that a Petri net possibly performs many internal actions (represented the arrow $\xrightarrow{\tau}^*$) before and after executing the computation step that is interest for the proof (i.e, action $\lambda$). Referring to the diagrams of Figure 1.1, this is a case of "star" simulation. The proof is performed by reasoning by induction on the structure of the xSpem model, and then by reasoning of the state transition semantics of xSpem models and PNs.

In [18], the authors describe a transformation from a model of the AADL formalism (Architecture Analysis and Design Language) to a particular kind of Abstract State Machine (ASM) called Timed Abstract State Machines (TASM). To verify that the transformation is semantic preserving, the authors define the semantics of AADL models and TASMs through Timed Transition Systems (TTSs). Thus, the execution state of an AADL model is the execution state of the corresponding TTS, and the same holds for a TASM. Comparing the state of two TTSs is easier than comparing the state of two different models. Then, the authors prove a strong bisimulation theorem to verify that the transformation is semantic preserving. The whole proof is mechanized within the Coq proof assistant.

In [10], the authors describe a transformation from LLVM-labelled Petri nets to LLVM programs, where LLVM is low-level assembly language. Precisely, the generated LLVM program implements the state space of the source Petri net (i.e, the graph of reachable markings). The authors want to verify if an LLVM program trully implements the PN state space, i.e if each marking present in the PN state space can be reached by running a specific $fire_t$ function on the generated LLVM program. The state of an LLVM program is defined by a memory model composed of a heap and a stack. The marking of an LLVM-labelled PN is defined in such a manner that the correspondence with the LLVM program memory model is straight-forward. The PN model has a classical firing semantics, and LLVM programs follow a small-step operational semantics. The semantic preservation theorem states that for all transition $t$ being fired, leading from marking $M$ to marking $M'$, then applying running the $fire_t$ function over the generated LLVM program at state $LM$ (such that $LM$ implements marking $M$) leads to a new state $LM'$, such that $LM'$ implements marking $M'$. To prove this theorem, the authors proceed by induction on the number of places of the Petri net.

#### Discussions on transformations and proof strategies

In this thesis, we are interested in the verification of a semantic preservation property for a given transformation by proving a bisimulation theorem. To achieve this kind of proof task, the proceedings are quite similar, at least in the three cases of transformation presented above (i.e, GPLs compilation, HDLs compilation and model transformations). Even though the source and target languages or models are different from one case of transformation to the other, however, bisimulation theorems carry the same structure. The state comparison relation and the choice of the bisimulation diagram are the two angular stones of the process.

One can notice that when verifying the transformation of HDL programs, the bisimulation theorems are expressed around a time-related computational step. It can either be a clock cycle, or another kind of time step. The state equivalence checking is made at the end this time-related computational step. This differs from the expression of bisimulation theorems for GPLs, where a computational step is not related to time, but rather expresses the one-time computation of programs.

Concerning proof strategies, in the case of programming languages, proving the bisimulation theorems are systematically done by induction over the semantics relation of the source languages. The semantics relation are themselves defined by following the inductive structure of the language ASTs. In the case of model transformations, when the source model permits it, the proofs are performed similarly by applying inductive reasoning over the structure of the input model. This enable compositional reasoning, i.e: to split the difficulty of proving the bisimulation theorem into simpler lemmas about the execution of simpler programs or simple model structures.

## 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] A. Habibi and S. Tahar. “Design and Verification of SystemC Transaction-Level Models”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 14.1 (Jan. 2006), pp. 57–68. ISSN: 1557-9999. DOI: 10.1109/TVLSI.2005.863187.

[12] 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).

[13] 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).

[14] 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.

[15] 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).

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

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

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

[19] Zhibin Yang et al. “Towards a Verified Compiler Prototype for the Synchronous Language SIGNAL”. In: *Frontiers of Computer Science* 10.1 (Feb. 1, 2016), pp. 37–53. ISSN: 2095-2236. DOI: 10.1007/s11704-015-4364-y. URL: https://doi.org/10.1007/s11704-015-4364-y (visited on 01/21/2020).