ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

**Mary Sheeran, Chalmers Technical University, Sweden, and Satnam Singh,
University of Glasgow, UK (Eds)**

Designing Correct Circuits

Proceedings of the 3rd Workshop on Designing Correct Circuits
(DCC96), Båstad, Sweden, 2-4 September 1996

ISBN: 3-540-76102-0

Paper:

Evaluating Possibilities for Formally Sound Simulation and Verification of VHDL

Ralf Reetz and Thomas Kropf

Published in collaboration with the
British Computer Society

Evaluating Possibilities for Formally Sound Simulation and Verification of VHDL

Ralf Reetz

Institut für Rechnerentwurf und Fehlertoleranz (Prof. D. Schmid), Universität Karlsruhe

Postfach 6980, 76128 Karlsruhe, Germany

email: Ralf.Reetz@informatik.uni-karlsruhe.de

www: <http://goethe.ira.uka.de/people/reetz/>

Thomas Kropf

Institut für Rechnerentwurf und Fehlertoleranz (Prof. D. Schmid), Universität Karlsruhe

Postfach 6980, 76128 Karlsruhe, Germany

email: Thomas.Kropf@informatik.uni-karlsruhe.de

www: <http://goethe.ira.uka.de/people/kropf/>

Abstract

Formal verification is a promising way to ensure correctness of digital circuits. VHDL is an important standard in describing digital circuits. This paper gives a survey about the state of the art in bringing formal verification and VHDL together. Up to now, there is no unique and best solution for the formal verification of arbitrary descriptions. The survey notes several aspects, which has been traded off against each other: “degree of automation”, “supported VHDL subset”, “practical usability”, “confidence of the approach”, to name just some. The existing approaches are compared with regard to these aspects. The reader gets an overview of the possibilities and limitations of different approaches with regard to simulation, symbolic simulation and formal verification. Finally, the formal VHDL environment created by the authors is presented.

1 Introduction

VHDL has become an important standard for specifying and designing digital circuits and systems. A wide variety of design tools using VHDL as their way of describing circuits have been developed and used in practice. However, guaranteeing correctness with the existing tools is still a critical moment in the design process. In the last years, the use of formal methods in circuit design was shown to be a new, promising way leading to tools for the design of correct circuits [1]. This paper has two aims: giving an overview of possibilities and limitations of different formal approaches toward formal design tools using VHDL and presenting the formal tool prototype for VHDL developed by the authors.

The base of formal methods is to use the language of a formal system (e.g. some sort of logics) to describe circuits. If VHDL is used for describing circuits, there must exist a formal, precise relation between VHDL and the formal system in order to get a formal description of the circuits. This is usually named as giving a formal semantics to VHDL. Many research groups have tackled this task [3, 4, 22, 30]. However, VHDL has been originally designed for simulation only, which makes it hard to establish a complete formal semantics well suited for formal methods, especially for (semi-)automated verification. In section 2, we discuss the different strategies and compromises people have chosen.

Once a formal semantics is given to VHDL, how can someone ensure that this semantics itself is correct, i.e. that it reflects the meaning given informally in the IEEE standard [2]? If the relation between the original design description written in VHDL and the formal description as the base of guaranteeing correctness is erroneous, a succeeding formal

verification is useless (and will probably fail anyway). In section 3, we discuss several approaches to this problem. It will be shown that these solutions have an influence on giving formal semantics and again, compromises have to be made. We compare the approaches with our solutions. Our aim was to explore other, new compromises or even develop methods to overcome the need of making compromises. Our efforts have led to a prototype of a formal VHDL environment for simulation, symbolic simulation and formal verification, all based on the same formal semantics, which is presented in section 4. The paper ends with a conclusion.

2 Formal VHDL semantics

The formal verification of a circuit is the proof in a formal system that a correctness relation holds between a specification and an implementation. Specifications and implementations are certain sorts of descriptions of a circuit in the language of the formal system. Implementations are usually complete circuit descriptions. A specification may also be a complete circuit description and the relation is then the equivalence between these two descriptions. Or the specification may be a description of safety and liveness properties, which the implementation should fulfil. Surprisingly, research has been focused on closing the gap between implementations written in VHDL and language of formal system as discussed in section 2.1, but only a few works have thought about how to write formal property specifications with VHDL, too. This is discussed in section 2.4. Abstraction levels and hierarchy in formal implementation descriptions with VHDL are discussed in section 2.2 and 2.3.

2.1 Formal languages and proof methods for VHDL

Giving a semantics to VHDL in terms of a formal language does not immediately results in having proof methods for the formal verification of VHDL descriptions at hand. The trade-off is the ease of describing the semantics against the availability of proof methods.

One approach is to use an operational and executable formalism well-suited for VHDL [31, 23]. Being close to the simulation model of VHDL, it is easily possible to formalize arbitrary parts of VHDL. However, the result is mainly usable for theoretical considerations or simulation and usually even lacks an underlying formal system with proof mechanisms needed for formal verification. Then it is not possible to perform proofs of any kind. The only possibility would be to develop specific reasoning systems for the used operational and executable formalisms. The converse approach is to use the formalism of an existent verification tool, resulting in a provable semantics and the possibility to directly use the respective tool. As the given formalism may not be well suited for VHDL, only a very restricted subset of VHDL may be treated. This especially holds for approaches based on automated verification tools [11, 5, 7, 10]. These tools are usually based on a state space traversal of a finite state machine. Although symbolic traversal techniques have led to a tremendous increase of the processable state space, the overall circuit size remains limited. Because of the finiteness of the state space, data types of VHDL have usually been restricted to simple, finite datatypes aimed at the register transfer level, i.e. to boolean, bits and bit vectors. Herrmann and Pargmann have enhanced this to all finite datatypes [26]. Additional assumptions about the overall behavior of a given VHDL description have to be made to ensure the static finiteness of the semantic model. Certainly one of the most important restrictions is the finite, pregiven length of the signal drivers [11].

The solution to overcome the restrictions imposed by the static finiteness of the formal model is to use a formal system powerful enough to drop the finiteness condition. However, this usually leads to interactive verification tools and thus to an expensive verification. One system used is the Nqthm system, using the logic of Boyer and Moore [32, 8]. The used datatypes are still aimed at the register transfer level, but the condition of the finite, pregiven length of the signal drivers can be dropped. A more general solution is to use an Theorem prover for higher-order-logic (hol) [13]. hol is powerful enough to give a formal semantics to VHDL without any restrictions. However, an interactive hol system is usually even more difficult to use than using Nqthm. Another approach is to use PROLOG [24]. This work has its own restrictions in subset and in possible verification goals.

In an ideal formal environment for VHDL one would like to have it both ways: a simple way to formalize all constructs of VHDL and access to existing verification systems. An early step in that direction was the Prevail environment [6], where different VHDL subsets were given different formal semantics, each fitting for a certain verification system.

The advantage of this approach is to have different verification techniques at hand, each well suited for a certain class of circuits (e.g. model checking for controllers and Nqthm for large, inductively structured data paths). However, the formal connection between these different semantics remains unclear and thus a complete verification of a circuit, consisting of subcircuits belonging to different classes, remains unsolved.

In our approach we used two key ideas in order to make a compromise possible. First, instead of using a VHDL-fitting model or of exploiting just the basic terms of a logic from a verification system, we used a “meet-in-the-middle” approach: we set up a new formal model (hierarchical composition of flow graphs), which is on the one hand based on the hol logic but which is on the other hand independent, but much closer to hardware description languages than the pure hol logic. The powerful extension methods of the hol logic itself ensures that our model is in fact a sound part of the logic itself. The second idea comes from the observation that the datatypes to be formalized decide which of the two main classes of verification systems has to be used: automatic model checking or interactive theorem proving [1]. Using the polymorphism of the hol logic, we made our formal model for VHDL parameterizable and thus independent from the actual formalization of the datatypes (including a datatype structure needed for expressing drivers, i.e. lists). Thus dependent on the kind of formalization of the datatypes, which is plugged into our parametrized semantics, the resulting non-parametrized semantics is equally well suited for either model checking or theorem proving. The main advantage is to have one semantics for different verification methods with a separate datatype semantics. The price we had to pay was a higher amount of formalization effort using the hol system HOL90 [14], compared to a model directly fitting for VHDL. Furthermore, having a very detailed formal semantics, verification with our approach is more complicated than using fully automated verification methods.

Two approaches form another category: they are not mainly interested in formal verification of VHDL description, but in formal proofs of properties of VHDL itself [15, 9]. Their used formalisms are still “paper-and-pencil” work and thus fall outside of the scope of this section.

2.2 Levels of time abstraction

A formal VHDL semantics can be given at different levels of abstraction. Informally spoken, an having an abstraction means “something is getting lost”. What is getting lost by describing a VHDL semantics at higher levels of time abstraction is the detail of the notion of progress in the simulation of a VHDL description. Thus the trade-off is the abstraction level of the semantics versus the supported notion of progress.

The lowest level of a semantics is the notion of progress of a VHDL simulator. Here an atomic operation executed “in one step” is the execution of one sequential statement or the execution of one certain operation during the update phase of one simulation cycle. As the semantics of VHDL is defined on that level [2], it has the possibility of giving the complete formal semantics. Many approaches use this level [9, 11, 15, 31, 30, 13]. However, there are two different classes. In the first class, there are those approaches that try to give a semantics based on a formalism, which normally is not powerful enough to describe that level completely. Especially approaches based on formalisms using state space traversal techniques fall in that class (see the previous section), since they can not capture dynamically changing drivers of arbitrary length. This is only possible with formalisms capable of describing infinite datatypes. The second class comprises those approaches, whose formalisms are appropriate for that level, namely higher order logics (again, see the previous section).

On the next level of abstraction, an atomic operation executed in one step is one complete simulation cycle, i.e. all sequential statements between two wait statements are executed simultaneously. Delta- and real-time-delay can still be supported, but the allowed forms of sequences of sequential statements have to be restricted, i.e. some progress on the step-by-step execution of sequential statements have to be abstracted away in order to squeeze this progress into one step. Giving a semantics on that level, these restrictions have been examined by Dharbe and Borriore [7].

The next level of abstraction is the complete abstraction from the simulation cycle. In this case the notion of progress is the notion of value changes in signal drivers. Here, one of both time dimensions of VHDL is usually dropped. Where almost all approaches eliminate the delta-delay and thus support only real-time delay (e.g. [24]), Fuchs and Mendler do the opposite by dropping the real-time delay and supporting delta-delay only [20].

Finally, on the highest level of abstraction, drivers are also abstracted away and the notion of progress is the notion of signal value changes. This level has been used in the early version of the Prevail environment [6]. As formal

verification do not need to deal with drivers here, this is very well suited for model checking. But as drivers are missing, delay mechanisms like transport and inertial delay can not be described.

2.3 Preserving hierarchy

Complex circuits are usually designed hierarchically. Hierarchy allows the repeated use of already designed submodules and speeds up the design process. The given design hierarchy may be useful in partitioning also the verification process by successively verifying circuit modules and using their proven specification on the next higher abstraction levels to reduce the proof complexity.

Unfortunately, although VHDL allows hierarchical descriptions, the design hierarchy is flattened out into a “sea-of-processes” during the elaboration phase, preceding all further analysis [2]. These processes are then executed in parallel and guarded by a dedicated update-process. The latter determines e.g. the new simulation time and new signal values. Hence all approaches, which use an abstraction level including the original simulation cycle, are not able to preserve hierarchy.

One method to preserve hierarchy is to go to an abstraction level where the simulation cycle is no longer represented. This has been realized in [8]. Note that this is again a compromise as going to that abstraction level, other things have to be left out.

In our approach, we have realized another strategy to preserve hierarchy [30]. We have given a semantics of VHDL on the lowest abstraction level including the simulation cycle by keeping the hierarchical process structure, leading to a different, but semantically equivalent semantics of VHDL. The basic idea is to retain each design entity (entity and architecture body) as a functional unit. All processes of that design entity are controlled by a separate update process for this design entity. This process controls all signals, whose driver are changed inside this design entity. Furthermore, all update processes of the respective design entities communicate with each other in a treelike structure, reflecting the original design hierarchy. This is necessary to determine the next simulation time and for synchronization purposes. Thus our formalization may serve as the basis for a hierarchical verification methodology. The compromise here lies in the treelike communication between the update processes, which needs additional simulation time and proving effort. However, as hierarchical verification is needed for large circuits, we feel the compromise has a point.

2.4 Property specifications in VHDL

An early approach was to use VHDL’s assert statement to specify properties [12]. A more complex early approach was to extend VHDL with new language constructs similar to the idea of using the assert statement [18]. The key idea here was to “hide” these new constructs in comments, so these annotated VHDL programs are still correct VHDL programs.

In our approach proposed in [29], the test bench concept usually used in VHDL simulations is adapted for property specifications, which we call a “verification bench”. A verification bench contains two parts: the implementation on the one hand and watchdogs on the other hand. The watchdogs continuously check that safety and liveness properties hold. In more detail, they drive two boolean signals: one signal is responsible for all safety properties and is always true if the safety properties always hold and the other signal is responsible for the all liveness properties and becomes true when the liveness properties become true. The formal semantics of the verification bench is given to the formal prover as the formal implementation and the formal specification is just the formal description of the property of these two signals as described in the previous sentence.

3 Validating the correctness of a formal VHDL semantics

A key issue to treating VHDL in a formal way is the trustworthiness of the implemented formal semantics. Hence ways have to be found to get confidence in it. As the standardized language reference manual of VHDL is given in plain English, there is no way to formally prove the total correctness of the given formal semantics.

3.1 Simulation

One way to validate the given formal semantics is to simulate the formal semantics of VHDL descriptions from a VHDL validation suite and compare these results with the results from the simulation of these VHDL descriptions. Here, two problems arise: first, there is no “golden simulator” or no “golden validation suite” which can be carelessly trusted. E.g. in our own practical experiences with the simulators we had at hand, different VHDL simulators sometimes do compute different results, i.e. they disagree on how the standard [2] should be interpreted. Second, it may be difficult to simulate the formal semantics because the language of the used formal system may not directly support simulation.

There are different ways to solve the second problem. One approach is using a formal semantics, which is simulatable without any “highly” probable erroneous translation. Purely functional, executable approaches do not need any additional translation. However, this is not really a solution because, as already mentioned in the previous section, these approaches usually lack an underlying formal system.

Another approach has been used by Rusinoff [8] and is also used in our formal environment for VHDL: one can use the semantical resemblance of programming languages and the logics of certain theorem provers to translate the symbolic representation of the formal semantics given in the logic into an executable representation very easily without the use of any error-prone significant changes. [8] has used the similarities between the logic used in Nqthm and LISP, we have used the similarities between HOL90 and SML [17] (see next section).

Finally, if the functional model is given in a deductive rule-based proof system as e.g. in [30, 8], then also a symbolic execution is possible, leading to a symbolic simulation, where results are given in terms of abstract signal and time values. However, one should note that a symbolic simulation is extremely space and time consuming and thus restricted to only small test examples.

3.2 Formal proof of properties

Another possibility for gaining confidence in the formal semantics is to prove consistency properties of the formal semantics itself, i.e. properties, which hold for all VHDL descriptions. To achieve this, it is not sufficient to have an abstract formal model on which the semantics is based on, but the semantics as well as the VHDL syntax and the formal relation between syntax and semantics have to be formalized in a formal system suited for formal proofs. Formalizing a language in this way is called “deep embedding” [25]. Only few approaches have formalized VHDL in this way (e.g. [24, 8, 30, 13]). In our approach as well as in [8] this advantage of a deep embedding has been used to verify simple properties. As already mentioned in the previous section, [15] has proven very general properties of his semantics, but this proofs are not machine-checked and this work was not aimed at the verification of concrete circuits.

Moreover, one has to pay a price to get a deep embedded, formally validated semantics: the formalization effort is much higher than a “shallow embedding”: there only the abstract formal model is represented formally. This explains why there are not many approaches which have formally verified properties of their semantics and have formalized a significant subset of VHDL at the same time. To ease this process, we have developed a formalization tool for automatically deep embedding arbitrary languages in higher order logic [27, 30]. This tool realizes a formal compiler-compiler and thus reduces the deep embedding effort to writing a VHDL compiler. This enabled us to do a deep embedding and to realize a significant subset of VHDL at the same time. However our formalization style makes formal proofs about properties more tricky than the approach presented by Gossens [15], thus up to now, we have only proved simple properties like Rusinoff [8].

4 A Formal Environment for VHDL

In our previous work [27, 28, 29, 30], we have set up a formal semantics of VHDL in higher order logic. Everything was realized in HOL90: the formalization of VHDL by deep embedding as well as first formal proofs. Our new contribution is a reimplementing of our formal semantics using a shallow embedding instead of the previously used deep embedding for efficiency reasons. Furthermore, we have added a simulator and several interfaces based on the shallow embedding.

To be able to explain our formal environment, some results from [30] are shortly summarized. Giving a formal semantics to VHDL has been done in three steps: first, a model capable of describing VHDL's simulation cycle has been deep embedded in hol. Second, a certain formalization style realizing deep embedding has been implemented in a formalization tool. Third, using this formalization tool, the formal relation between VHDL syntax and the model has been set up.

Figure 1: Parameterized models

Fig. 1 shows that we have not formalized one, but three parametrized models fitting to each other. For structural descriptions, there is the model of the hierarchical combination of transition systems. This structural model is parametrized with the transition systems, i.e. it works for all transition systems, which fulfill certain interface restrictions. For behavioral descriptions, there is the model of flowgraphs, whose semantics is given in terms of a transition system, fitting into the structural model. The behavior model is parametrized with datatypes, i.e. it works for all formalizations of datatypes. Finally, a formal model for scalar datatypes of VHDL has been formalized. In contrast to [26], we have not used a finite, but a more abstract, infinite formalization base, namely the integers in the sense of Peano axioms.

Our formalization tool for deep embedding is a compiler-compiler for formal compilers. Given a context-free grammar and a set of semantic rules, it constructs a formal representation of the syntax as derivation trees in terms of hol and a formal representation of the semantics as a compiler in terms of hol. Then formally proving properties of the defined compiler means formally validating the semantics. Formally deriving the result of applying the formal compiler to a formal derivation tree means formally compiling a program.

Fig. 2 summarizes the application of the formalization tool for VHDL and compares it to the standard semantics [2]. The parametrized models are used as a base for semantic rules for VHDL, which are formulated as terms of higher order logic. The compiler-compiler, which is implemented in SML, takes these semantic rules together with a context-free grammar and produces a representation of a formal VHDL compiler in higher order logic. Elaboration of a VHDL description in the standard means parsing the VHDL description into a formal derivation tree and then formally deriving the result of applying the formal compiler to a derivation tree representing the VHDL description. Executing the simulation cycle means formally deriving the state sequences of the previously derived hierarchical combined flowgraphs.

4.1 Similarity between higher order logic and SML

As discussed in section 3.1, we want to get an efficiently executable semantics of VHDL from our formal semantics with a “high” degree of confidence with regard to the correctness of the resulting simulator. The basic idea is to make use of the similarity between the logic hol and the programming language SML. This similarity even made the deep embedding of a major subset of SML in hol possible [21]. The “kernel” of both, hol and SML, is a typed lambda-calculus (a formal system to describe functions). Four basic constructs form the kernel: constants, variables, abstraction (function definition) and application (application of a function to some arguments). Their semantics are the same in hol and SML. Syntactically, only trivial changes have to be performed for a translation: Variables are identified by their names and can thus be trivially kept the same when going from hol to SML. We only needed simple constant

LRM semantics *our formal semantics*

Figure 2: Application of the formalization tool

definitions forms in hol: a single equation or a conjunction of equations. To define these constants in SML, only the addition of the keyword “val” or “fun” in front of an equation is needed and the conjunction symbol “/\” has to be changed to “|”. Abstraction in hol is written as “\x. E” (function with argument x and body E) and written in SML as “fn x => E”. Application is written the same way.

As already mentioned, both are typed lambda-calculi, meaning that every expression in the lambda-calculus has a type like types in well-known programming languages like PASCAL or C. Thus their type systems determines the describable objects and are an important subject here. Both contain the same core of a type system. It comprises standard data structures for storing informations like pairs, lists, trees, etc. The only problem we encountered here concerned tuples, which are expressed as multiple pairs. In hol, pairs are right-associative, i.e. $(x,y,z) = (x,(y,z))$, but in SML, it is just the other way around, i.e. $(x,y,z) = ((x,y),z)$. This subtle difference is not obvious at the first glance and we had to spend quite some time to get aware of this.

However, the type system of hol contains other type mechanisms, with enables formal proofs of consistency properties about the way how data elements are stored in the data structures. Going from hol to SML, these consistency properties get lost, although the semantics of the functions remain the same.

4.2 Software libraries

The first thing to do was to transform the hol formalization of the flowgraph model and the hierarchical combination of transition systems into an SML version. As discussed in the previous section, this could be done by changing the syntax and by taking care of pairs. The SML versions of these models realize programming libraries implemented in SML, which can be used for the implementation of own, specialized verification algorithms in the future.

The next thing to do was to add an SML program output to the compiler-compiler. Except the fact that expressions describing semantic rules had to be changed from hol terms to SML expressions, everything else in the interface could remain the same. In our experience, formally deriving the formal semantics of concrete VHDL descriptions of around 100 relevant lines needed 10-30 minutes on a SuperSparc 50Mhz. The SML version needs just a second or two.

Considering the significant speed-up, one might wonder why we did not use a SML version from the very beginning. First, the SML version realizes a shallow embedding, which can not be formally validated. Second, if the SML version of the compiler-compiler has a bug, the translation of the VHDL descriptions has a bug, too. But the

deep embedded hol version is validated by the consistency proofs mentioned above and the derivation result is always correct with respect to the supplied semantic rules. In fact, using this property, we could easily identify two bugs in the first SML version of our compiler-compiler by comparing the SML version to the hol version. This reflects exactly the message we have learned from doing it that way: first, formalize your objects in hol, formally validate them using the hol theorem prover and then easily create an SML version for the practical use because of efficiency. In [30], we did the formal part in hol and here we transformed the semantics to SML and added some tools.

Pages of textual formulas describing flowgraphs and hierarchical structures are difficult to read. On the other hand, drawing graphs nicely without any prior placement information is a whole research field. Thus instead of going into that complex subject in detail or relying on a trivial solution, we have implemented an interface from flowgraphs and hierarchical structures to the existing tool daVinci [19], a general graph visualization system. Thus we can graphically visualize the semantics of a VHDL description.

As additional interfaces from flowgraphs and hierarchical combinations of transition systems, we have implemented an interface to the theorem prover HOL90 based on the hol-version of our formal semantics and an interface to the model checker SMV [16]. As our models are parametrized with datatype formalizations, so are the interfaces, e.g. in order to use an interface from the flowgraphs to SMV, an interface from the used datatypes to SMV has to be given. Finally, a simulator for the hierarchical combination of transition systems has been implemented, which is parametrized with a simulator for the transition systems itself. Here, an implemented simulator for the flowgraphs is plugged in, which itself is parametrized with an executable version of the datatype functions.

Figure 3: Software libraries of the formal environment

To summarize, a whole collection of libraries for flowgraphs and hierarchy with simulators and interfaces for programming, for the theorem prover HOL90 and for the model checker SMV has been implemented. Fig. 3 gives an overview of the libraries. A functor in SML is a higher-order function, which takes a library and maps it to another library, thus realizing parametrization. Once libraries for the concrete used datatypes are set up, the functors are used

to create a new collection of libraries especially for these concrete datatypes. Thus it is possible to use the flowgraph and hierarchy libraries with different datatypes at the same time.

4.3 The formal environment for VHDL

The collection of libraries combined with the compiler-compiler forms an environment for formalizing a wide range of hardware description languages. We have used this environment for our formal VHDL semantics. The semantic rules previously written in HOL have been translated to SML. For our infinite VHDL datatype formalization, we have implemented the necessary SML libraries. But as this formalization is based on integers, an interface to theorem provers based on finite state space traversal like SMV is not possible in principle. However, alternatively one could plug in the finite datatype formalization from [26] in our semantics to get a trivial interface to SMV. Thus we are able to produce a hol term describing the semantics of a VHDL description. Alternatively, using the results from [26], we could produce a Kripke structure for SMV describing the semantics of a VHDL description.

Figure 4: Formal environment at work

We want to emphasize here that our formal environment for VHDL is still not a tool ready for use by a hardware designer. There is no graphical user interface nor an integration into some commercial design framework. However, research in formal semantics of VHDL and formal verification of VHDL descriptions can be done. Fig. 4 gives a screen shot from the environment at work: on the top right, there is an usual text editor containing the considered

VHDL description. On the top left, daVinci visualizes the hierarchy of the formal semantics of the VHDL description. The flowgraphs in the formal semantics can also be visualized. In the right middle, the commands used in the formal environment are managed in a text editor. In the left middle, there is the command shell of the formal environment. Here, verification with HOL90 can be done. Finally, on the bottom, there is the output of a simulation of one submodule. Arbitrary subparts of the hierarchy of the formal semantics can be simulated or verified.

5 Conclusion

Up to now, there is no unique and “best” solution for the formal verification of arbitrary VHDL description. Planning the implementation of a formal verification tool for VHDL, several questions have to be answered: What is the desired subset of VHDL? What is the desired degree of confidence in the tool? What is the desired degree of automation for the verification process? What is the desired size of circuits to be verified? With the current state of the art, there is no single method simultaneously satisfying all the requirements, given by the above questions, in an optimal way. A wide range of approaches, including ours, have thus explored different possible compromises.

In our on work, we have implemented a formal environment for VHDL. We have explored a way from a deep embedding with high confidence to a shallow embedding with practical execution times. However, the closeness of the latter to the original deep embedded version ensures a high confidence. In our formalization environment, VHDL descriptions can be translated into their formal semantics. This formal semantics can be simulated, graphically visualized and formally verified with the higher order logic theorem prover HOL90. In combination with e.g. [26], a formal verification with the model checker SMV would also be possible.

References

- [1] A. Gupta. Formal hardware verification methods: A survey. *Journal of Formal Methods in System Design*, 1:151–238, 1992.
- [2] ANSI/IEEE Std 1076–1993. *IEEE Standard VHDL Language Reference Manual*. IEEE, New York, USA, June 1994.
- [3] C. Delgado Kloos and P.T. Breuer, editors. *Formal Semantics for VHDL*, volume 307 of *The Kluwer international series in engineering and computer science*. Kluwer Academic Publishers, Madrid, Spain, March 1995.
- [4] D. Borriane. Special Issue on VHDL Semantics. *Formal Methods in System Design*, 7(1/2), August 1995.
- [5] D. Borriane and A. Salem. Denotational Semantics of a Synchronous VHDL Subset. In [4], pages 53–72.
- [6] D. Borriane, L. Pierre, and A.M. Salem. Formal Verification of VHDL Descriptions in the Prevail Environment. *IEEE Design & Test of Computers*, pages 42–55, June 1992.
- [7] D. Déharbe and D. Borriane. Semantics of a verification-oriented subset of VHDL. In [22], pages 293–310.
- [8] D.M. Russinoff. A Formalization of a Subset of VHDL in the Boyer-Moore Logic. In [4], pages 7–26.
- [9] E. Börger, U. Glässer, and W. Müller. A formal definition of an abstract VHDL’93 simulator by EA-machines. In [3], chapter 4, pages 107–139.
- [10] E. Encrenaz. A Symbolic Relation for a Subset of VHDL’87 Descriptions and its Application to symbolic Model Checking. In [22], pages 328–342.
- [11] G. Döhmen and R. Herrmann. A deterministic finite-state model for VHDL. In [3], chapter 6, pages 170–204.
- [12] J. Van Tassel and D. Hemmendinger. Toward formal verification of VHDL specifications. In *International Workshop on Applied Formal Methods for Correct VLSI Design*, pages 409–418. IFIP WG 10.2/WG 10.5, North-Holland, 1989.

- [13] J.P. Van Tassel. An operational semantics for a subset of vhdl. In [3], chapter 3, pages 71–106.
- [14] K. Slind. An Implementation of Higher Order Logic. Technical Report 91–419–03, University of Calgary, Computer Science Department, 1991.
- [15] K.G.W. Goossens. Reasoning about VHDL using operational and observational semantics. In [22], pages 311–327.
- [16] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
- [17] L.C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.
- [18] L.M. Augustin, B.A. Gennart, Y.Huh, D.C. Luckham, and A.G. Stanculescu. Verification of VHDL designs using VAL. In *25th Design Automation Conference*, pages 48–53. ACM/IEEE, 1988.
- [19] M. Fröhlich and M. Werner. Demonstration of the interactive graph visualiation system daVinci. In R. Tamassia and I. Tollis, editors, *DIMACS Workshop on Graph Drawing '94*, volume 894 of *Lecture Notes in Computer Science*, pages 266–269, Princeton, USA, 1995. Springer Verlag.
- [20] M. Fuchs and M. Mendler. A functional semantics for delta-delay VHDL based on focus. In [3], chapter 1, pages 9–42.
- [21] M. van Inwegen and E.L. Gunter. HOL-ML. In J.J. Joyce and C.-J.H. Seger, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 59–73, Vancouver, Canada, August 1993. University of British Columbia, Springer-Verlag, published 1994.
- [22] P.E. Camurati and H. Ekeking, editors. *Correct Hardware Design and Verification Methods*, volume 987 of *Lecture Notes in Computer Science*. IFIP WG10.5 Advanced Research Working Conference CHARME'95, Springer-Verlag, October 1995.
- [23] P.T. Breuer, L.S. Fernandez, and C. Delgado Kloos. A Functional Semantics for Unit-Delay VHDL. In [3], chapter 2, pages 43–70.
- [24] P.T. Breuer, L.S. Fernandez, and C. Delgado Kloos. A Simple Denotational Semantics, Proof Theory and a Validation Condition Generator for Unit-Delay VHDL. In [4], pages 27–52.
- [25] R. Boulton, A. Gordon, M.J.C. Gordon, J. Herbert, and J. van Tassel. Experience with embedding hardware description languages in HOL. In *International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 129–156, Nijmegen, June 1992. IFIP TC10/WG 10.2, North-Holland.
- [26] R. Herrmann and H. Pargmann. Computing Binary Decision Diagrams for VHDL Data Types. In *European Design Automation Conference*, pages 578–585, Grenoble, France, September 1994. IEEE Computer Society Press.
- [27] R. Reetz and T. Kropf. Simplifying Deep Embedding: A Formalised Code Generator. In T.F. Melham and J. Camilleri, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 859 of *Lecture Notes in Computer Science*, pages 378–390, Malta, September 1994. Springer-Verlag.
- [28] R. Reetz and T. Kropf. A Flowgraph Semantics of VHDL: A Basis for Hardware Verification with VHDL. In [3], chapter 7, pages 205–238.
- [29] R. Reetz and T. Kropf. A Flowgraph Semantics of VHDL: Toward a VHDL Verification Workbench in HOL. In [4], pages 73–100.
- [30] R. Reetz. Deep Embedding VHDL. In E.T. Schubert, P.J. Windley, and J. Alves-Foss, editors, *8th International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 277–292, Aspen Grove, Utah, USA, September 1995. Springer-Verlag.

- [31] S. Olcoz. A formal model of VHDL using colored petri nets. In [3], chapter 5, pages 140–169.
- [32] S. Read and M. Edwards. A Formal Semantics of VHDL in Boyer–Moore Logic. In *2nd International Conference on Concurrent Engineering and EDA*, San Diego, CA, 1994. SCSl.