Degrees of Formality in Shallow Embedding Hardware Description Languages in HOL*

Catia M. Angelo, Luc Claesen, Hugo De Man

IMEC vzw, Kapeldreef 75, B3001 Leuven, Belgium

Abstract. Theorem proving based techniques for formal hardware verification have been evolving constantly and researchers are getting able to reason about more complex issues than it was possible or practically feasible in the past. It is often the case that a model of a system is built in a formal logic and then reasoning about this model is carried out in the logic. Concern is growing on how to consistently interface a model built in a formal logic with an informal CAD environment. Researchers have been investigating how to define the formal semantics of hardware description languages so that one can formally reason about models informally dealt with in a CAD environment. At the University of Cambridge, the embedding of hardware description languages in a logic is classified in two categories: deep embedding and shallow embedding. In this paper we argue that there are degrees of formality in shallow embedding a language in a logic. The choice of the degree of formality is a trade-off between the security of the embedding and the amount and complexity of the proof effort in the logic. We also argue that the design of a language could consider this verifiability issue. There are choices in the design of a language that can make it easier to improve the degree of formality, without implying serious drawbacks for the CAD environment.

1 Introduction

To interface theorem proving frameworks with CAD environments, researchers have been formalizing the semantics of hardware description languages. The two approaches for embedding HDLs in the HOL system [1], *deep embedding* and *shallow embedding*, are described in [2]. In *deep embedding*, the abstract syntax of an HDL is represented in the HOL logic and, within the logic, semantic functions assign meanings to the programs. The syntax of the HDL is represented by a HOL type. For instance, one could define in the HOL logic a type for the syntactic class of expressions of the HDL and then define in the logic a function giving meaning to the expressions. Deep embedding allows one to reason about classes of programs because it is possible to quantify over syntactic structures [3]. For instance, one could quantify over expressions of the HDL. The type definition package developed by Tom Melham [4] provides automation and support for the user to define a class of recursive types and to make proofs involving them.

* This research was sponsored by CNPq (Brazilian Government) and CHEOPS Project (ESPRIT BRA "3215").

However, defining the types to represent the abstract syntax of a language and semantic functions in the logic can be very complex and time-consuming. In *shallow embedding*, semantic operators are defined in the HOL logic and an ML [5] interface interprets programs into semantic structures in the logic. The abstract syntax of the HDL is represented by a ML type. Since the production rules of the language are not modeled in the logic, one cannot reason directly about the language in the logic. However, shallow embedding a language in the logic still allows one to model and reason about a specification written in the embedded language. One can also prove theorems about the semantic operators defined in the logic. Shallow embedding is less secure than deep embedding because the interpretation of programs is outside of the logic and therefore one cannot reason about this interpretation in the logic. However, shallow embedding is simpler than deep embedding. In general, it is much easier to define ML types than HOL types[2]. The shallow embedding method in HOL has been used in [7, 8, 9, 10, 11] and the deep embedding method in HOL has been used in [12, 13].

In the shallow embedding style, the semantics of a language is needed to consistently map a specification written in that language into its meaning in the HOL logic and, once in the logic, to reason about it. The reasoning in the HOL logic is safe but the parsing of programs written in the HDL and the semantic interpretation (the assigning of meanings to the programs) are possible sources of errors because they are implemented in ML and not in the logic. The simpler the parsing and the semantic interpretation, the safer the whole system. However, the more elaborate the semantic interpretation is, the simpler the model in the logic can be and, therefore, reasoning about it can be easier.

The shallow embedding style has two informal components (the parsing and the semantic interpretation) and a formal component (the reasoning about the meaning of programs in the logic). The degree of formality of the system is defined by the relative "weights" of these components in the whole system. We mean that a component A of the embedding has a heavier "weight" than a component B of the embedding if "more issues" about the embedding are addressed by A than by B and/or if the issues addressed by A are "more complex" (according to a certain criteria) than the issues addressed by B. The choice of the degree of formality is a trade-off between the security of the embedding and the amount and complexity of the proof effort in the logic. The development of infra-structure to build models and reason about them in the HOL logic can reduce the proof effort to reason about complex issues. This kind of infra-structure makes it easier to formally address complex issues that would otherwise be addressed by an informal component of the embedding. Therefore, this kind of infra-structure makes it easier to improve the degree of formality of language embeddings, which means to move in the scale of formality closer to the logic.

The deep embedding style has one informal component (the parsing) and two formal components (the semantic interpretation and the reasoning about the meaning of programs in the logic). One might think of deep embedding as a

[2] In [6], standard ML types and HOL types are discussed.

particular case of embedding style that has one less informal component than the shallow embedding style has[3]. On the other hand, one might think of informal programs (based on algorithmical approaches for the verification of aspects of a specification described in a language) as an extreme case of embedding style that has no formal component (no reasoning in a logic).

The embedding of languages in a logic also has to address well-formedness issues of specifications. Whether these issues are addressed informally or in the logic also has an impact on the security of the whole system and on the complexity of the proof effort in the logic. In the shallow embedding style, well-formedness issues can be addressed informally by implementing an algorithm in ML to check properties of specifications described in the embedded language. These issues can also be addressed formally by implementing proof procedures in ML that can prove in the logic properties of specific programs described in the embedded language. These proof procedures can be based on theorems about the semantic operators defined in the logic. In contrast with the shallow embedding style, in the deep embedding style one has additionally the possibility of using predicates defined over the types representing the syntax in order to address well-formedness issues in the logic.

In this paper we present a discussion about degrees of formality in the shallow embedding style, based on the definition of the multi-rate semantics of Silage [11]. In the next section we give an informal overview of the Silage language. Then we outline the basic principles of the semantics of Silage, focusing on a particular aspect. Next we discuss the semantic model and the idea of degrees of formality. Finally, some conclusions are drawn.

2 An Overview of Silage

The Silage language [14, 15, 16, 17, 18] used as input of the CATHEDRAL [19] silicon compilers is an applicative language suitable for describing DSP algorithms represented as a data-flow graph. No assignment operators are allowed. A basic concept of the language is the notion of a signal. A signal is an infinite stream of data indexed by discrete and periodic time instants. Signals are defined by function applications. The single assignment principle allows signals to be defined only once. A definition involving signals is a set of equations about the samples of the signals involved. The operations are vectorial and the indexing is implicit. Previous samples of the signals can be referred to in a signal definition. Arrays of samples can be manipulated. There are iterators and an **if** construct in Silage. No statements are made about the order or concurrency of the operations and the parallelism of computation can be exploited by the synthesis tool by analyzing the data flow dependencies.

There are operators in Silage that can generate signals with periods that are different from the periods of the signals they are derived from. The period of

[3] This does not mean that the deep embedding style is a particular case of the shallow embedding style, which is not true.

signals are implicit in Silage programs and they are derived by the synthesis tool. One cannot explicitly define the periods of the signals.

In Silage definitions, signals or samples on the left-hand side of the equality symbol are defined in terms of the expressions on the right-hand side. The order of the definitions is irrelevant. For instance, consider the piece of Silage code below. The second definition is recursive. The first definition initializes the second one. The operator @ is used to access previous samples of a signal and the operator @@ is used to define the values of the *initialisation samples*. The second definition means that the n-th sample of the signal **x** is defined by the addition of the n-th sample of a signal **in** with the previous sample of the signal **x**. The period of the signal **x** must be equal to the period of the signal **in**.

```
x@@1 = 0;
x = in + x@1;
```

Tuples of the form (a,...,z) in Silage aggregate signals that may have different periods. Tuples cannot be nested. On the other hand, the aggregate constructor of the form {a,...,z} aggregates signals with the same period. Functions in Silage returning multiple signals can be called using tuples. The two pieces of code below (where the initialisations are omitted) illustrate Silage functions and mutually recursive definitions in Silage. Both functions take as argument a signal **in** and return a signal **out**. The signals **a** and **b** are internal signals. When a Silage function has more than one argument signal, it is implicit that they have the same period. Unlike the first piece of code, the second one is illformed because of cyclic data dependencies between samples.

```
func Ok (in:word) out:word =
a:word; b:word;
begin
   b = a - b@2;
   out = b + a@1;
   a = in + a@2 + b@1;
end;
```

```
func notOk (in:word) out:word =
a:word; b:word;
begin
   b = a - b@2;
   out = b + a@1;
   a = in + a@2 + b;
end;
```

So far, except for the tuples and function definitions, only the Silage operators that handle signals with identical periods have been considered. When a signal is defined using these operators, it is implicit that the periods of all the signals involved are the same. Next we consider the **decimate**, the **interpolate** and the **switch** functions, that handle signals with different periods.

The **decimate** function takes as argument a source signal **sourceS** and returns a list of **n** signals **listS**. The signal **sourceS** is decomposed into **n** signals with a period **n** times greater than the period of **sourceS**. The samples of **sourceS** are distributed among the output signals in the order they appear in **listS**.

The **interpolate** function takes as argument a list **listS** of **n** signals with equal periods and returns a destination signal **destS**. The samples of the signals

in **listS** are interleaved generating the signal **destS** with a period **n** times smaller than the period of the signals in **listS**.

The following is an example of illformed Silage program because of inconsistencies with the periods. The first definition defines the period of the signal **out1** as half of the period of the signal **in1** and the definition of **out2** is only possible if the periods of **in1** and **out1** are equal.

```
func notOkToo (in1:word; in2:word) out1:word; out2:word =
    begin
        out1 = interpolate (in1, in2);
        out2 = in1 + out1;
    end;
```

The **switch** function takes as arguments **n** signals and returns **m** signals. The **n** argument signals are interpolated generating a temporary signal and this is decimated into **m** signals. In particular, if (**m** = 1), then **switch** is equivalent to **interpolate** and if (**n** = 1), then **switch** is equivalent to **decimate**.

In a Silage program, the periods of all signals depend on each other. One Silage definition alone does not define the period of the signals involved, but it defines a relation between them. One has to keep track of the period information in *every* relation modeling a Silage definition. Even for the operations that handle signals with equal periods, this information has to be explicit in the semantic interpretation. Most operations are only meaningful if the periods of the signals involved have some property. For instance, only signals with equal periods can be added or interpolated. Each Silage definition relates the samples and the periods of the involved signals.

3 Modeling the Period Semantics of Silage

In [11] the formal multi-rate semantics of Silage is defined relationally in the shallow embedding style. Each Silage definition is interpreted separately stating relations between the samples, the periods, and the phases of the signals. No distinction is made between the input and output signals of the DSP algorithm. In this section, we show informally the basic principles of the multi-rate semantics of Silage, with reference to the periods of the signals. We will refer to this view of Silage programs as the *period semantics* of Silage programs. This view of Silage programs will be used to illustrate the idea of degrees of formality in the shallow embedding style.

Next we introduce the basic ideas of the model of the periods of the signals in the multi-rate semantics of Silage, by refining the interpretation of some examples. In [11], Silage signals are modeled by a tuple of three components: samples (a function from time to sample values), period (the period of the signal), and the phase of the signal (the time instant when the first sample happens). Consider the following Silage definition *def*: $\boxed{y = x}$, where y and x are signal variables. Suppose also that **PER** is a selector defined in the HOL logic that applied to a

signal returns the period component of the signal. Then the period semantics of the definition *def* should be:

$$[\![y = x]\!]_p = ((\text{PER y}) = (\text{PER x}))$$

A signal variable is a particular case of Silage expression. Silage expressions are typed in ML. In [11], the period of every kind of Silage expression is defined. In particular, the period of a signal can be a multiple of the periods of other signals when the multi-rate Silage constructs **decimate**, **interpolate** and **switch** are involved. Suppose that the period of a generic Silage expression is $[\![e]\!]_p$. Then a first attempt to define the period semantics of the Silage equality between two expressions $e_1 = e_2$ could be:

$$[\![e_1 = e_2]\!]_p = ([\![e_1]\!]_p = [\![e_2]\!]_p)$$

However, there is a problem with this definition of $[\![e_1 = e_2]\!]_p$. In Silage, there might be constant signals and signals obtained by interpolating constants. These signals do not have period information. We have defined $[\![e]\!]_p$ such that, whenever an expression does not have period information, $[\![e]\!]_p$ is zero. However, the interpretation of a Silage definition such as y = 2 should not define the period of y as zero. No signal in a Silage program has period equal to zero. So, we improve the definition of $[\![e_1 = e_2]\!]_p$ as below, where T means true in the logic.

$$[\![e_1 = e_2]\!]_p = (\text{if } ([\![e_2]\!]_p = 0) \text{ then T else } ([\![e_1]\!]_p = [\![e_2]\!]_p))$$

There is still a problem with the definition of $[\![e_1 = e_2]\!]_p$. Suppose that we have a definition such as $z = x + y$, where z, x and y are signal variables. Then we would have:

$$[\![z = x + y]\!]_p = ([\![z]\!]_p = [\![x + y]\!]_p)$$

But the period of $(x + y)$ must be either the period of x or the period of y. Whatever it is, there must be a guarantee that the period of x is equal to the period of y. If the periods of every signal in Silage were declared, this could be checked statically by defining checking rules for each kind of Silage expression. Since this is not the case, whenever an expression is composed of many subexpressions, there are implicit constraints about the periods of the subexpressions. In [11] we have defined in ML the *check semantics of expressions* $[\![e]\!]_c$ to model these constraints in the logic for every kind of Silage expression. So, refining the definition of $[\![e_1 = e_2]\!]_p$ we have:

$$[\![e_1 = e_2]\!]_p = (\text{if } ([\![e_2]\!]_p = 0) \text{ then T else } (([\![e_1]\!]_p = [\![e_2]\!]_p) \wedge [\![e_2]\!]_c))$$

In particular, $[\![e]\!]_c$ and $[\![e]\!]_p$ are defined in such a way that:

$[\![x + y]\!]_c$ = ((PER x) = (PER y))
$[\![x + y]\!]_p$ = (PER x)

Then the period semantics of $z = x + y$ is defined as follows:

$[\![z = x + y]\!]_p$ = (((PER z) = (PER x)) ∧ ((PER x) = (PER y)))

The body of a Silage function is composed of a set of Silage definitions. No external or internal signal in a Silage function has period equal to zero. The arguments of a Silage function (inputs of the DSP algorithm) must have equal periods. These constraints and the set of period semantics of the definitions in the body of a Silage function define a linear system of equations with natural numbers. If a Silage function defines consistent periods for the signals, there must be a solution for this system. For instance, consider the following Silage function (where the initialisations are omitted):

```
func example (in1:word;in2:word) out:word =
a:word; b:word;
begin
    b = in1 + (a - b@2);
    out = interpolate(a,b);
    a = in2 + (a@2 + b@1);
end;
```

The period semantics of this function is shown below and it is defined in terms of the HOL functions **NOT_ZERO**, **SYNC**, and **MULTIPLE**. Given a list of natural numbers, the predicate **NOT_ZERO** holds if all elements of the list are different from zero. The predicate **SYNC** holds if all elements in a list are equal to a given element. Given two natural numbers **n** and **m**, the predicate **MULTIPLE** holds if **n** is a multiple of **m**.

```
∀ in1 in2 out. PER_example(in1,in2,out) =
  (NOT_ZERO[PER in1; PER in2; PER out]) ∧
  (SYNC [PER in1; PER in2] (PER in1)) ∧
  (∃ a b.
     (NOT_ZERO[PER a; PER b]) ∧
     (((PER b) = (PER in1)) ∧ ((PER in1) = (PER a)) ∧ ((PER a) = (PER b))) ∧
     (((PER out) = ((PER a) DIV (LENGTH [a;b]))) ∧
      (MULTIPLE (PER a) (LENGTH [a;b])) ∧
      (SYNC [PER a;PER b] (PER a))) ∧
     ((PER a) = (PER in2)) ∧ ((PER in2) = (PER a)) ∧ ((PER a) = (PER b)))
```

4 Discussing the Model

In this section we discuss the model outlined in the previous section and the idea of degrees of formality in the shallow embedding style.

The interpretation of Silage definitions in a relational style is less elaborate than it is in a functional style because it is less context dependent. In the relational style, Silage definitions are interpreted one for one into terms of the logic in one step of compilation. In a functional style, a group of Silage definitions is interpreted all together to define the samples of a signal as a function from time instants to values. The definitions take as parameters the input signals and return the output signals. For each signal, the definitions about its initialisation samples and its algorithmical samples are considered all together to define the signal as a function. Such an approach has to handle functions that are not defined for all discrete time instants and has also to cope with the complexity of mutually recursive definitions. The interpretation of Silage programs in the functional style cannot be performed in one step of compilation and it involves more informal manipulation in ML to derive the meaning of Silage programs in the logic than the relational style does. Therefore, the choice between a relational style and a functional style has an impact on the degree of formality of the shallow embedding of Silage in HOL.

The simpler the semantics of a language in the shallow embedding style, the lighter the weight of the informal part of the embedding (the ML code) and therefore the safer is the whole system. There is no formal way to measure the complexity of a semantic interpretation and the complexity of the verification to be performed on specific interpretations. However, it is intuitive that the greater the complexity of a semantic interpretation (in ML) with respect to the complexity of the verification to be performed on specific interpretations (in the HOL logic), the worse is the quality of the embedding with respect to security. Ultimately, both the semantic interpretation and the verification should be informal (to have a system that is simpler but not much less safe than the one built in an unbalanced shallow embedding style) or the deep embedding style should be used (to have a system that is much safer than the one built in the unbalanced shallow embedding style).

Not only the way meaning is attached to programs affects the degree of formality in shallow embedding languages in HOL. Consistency issues also have to be addressed informally or formally, affecting the degree of formality. Next we discuss how one could address in the logic the well-formedness issue of consistency of the periods in Silage. Let X_p be the relational model of the period semantics of a Silage function x. For instance, the relational model of the period semantics of the Silage function **example** of the previous section is **PER_example**. The Silage function x defines consistent periods for its signals only if there are signals for which X_p holds. This is not the case, for instance, of the Silage function **notOkToo** defined in a previous section. This consistency issue can be addressed informally by an algorithm implemented in ML but it can also be addressed more formally in the logic, improving the degree of formality of the embedding. Although we have not automated the proof that a Silage program

defines consistent periods for its signals, we have proved theorems about the constants defined to handle the periods of Silage expressions that allow us to prove automatically that the period semantics of a Silage function is equal to a linear system of equations about the periods. We will call this system **x_system**[4]. The proof about the consistency of the periods can be reduced to a first-order proof about natural numbers. If **x_system** can be solved informally using conventional techniques, the solutions of this system are the witnesses necessary to prove in the logic that the periods are consistent. *Addressing well-formedness issues in the logic, rather than informally, is a way to move closer to the logic in the scale of formality.* A system built in the shallow embedding style that addresses consistency issues both in its informal components (ML code) and in its formal component (HOL logic) is a hybrid verification system.

The main motivation for the definition of the multi-rate semantics of Silage was to prove the correctness of source-to-source transformations [20, 21, 22]. These transformations are used to optimize the results of the silicon compilation but they should not change the input-output behavior of the DSP algorithm specified in Silage. To prove the correctness of the transformations, one has to prove that the period semantics of a Silage function is equal to the period semantics of a new Silage function[5]. In particular, to prove the correctness of some transformations, it would be easier to have a more compact model of the periods of the signals. This simpler model could be obtained by trying to solve the **x_system** with conventional informal techniques. For instance, suppose that, for the Silage function **example** of the previous section we would have the following model:

```
∀ in1 in2 out. PER_example'(in1,in2,out) = ∃ a b.
  (NOT_ZERO[PER in1; PER in2; PER out;PER a; PER b]) ∧
  (SYNC [PER in1; PER in2; PER a; PER b] (2 * (PER out)))
```

One can prove in HOL the theorem below and then use it to prove the correctness of transformations on the Silage function **example**.

```
⊢ ∀ in1 in2 out. PER_example(in1,in2,out) = PER_example'(in1,in2,out)
```

If the semantic interpretation were more elaborate so that **PER_example'**, rather than **PER_example**, would be the period semantics of **example**, the effort to prove the equivalence between these two models would be saved, but we would be moving away from the logic in the scale of formality. This is because the equivalence between the two models would have to be done informally by implementing an algorithm in ML. A proof procedure that formalizes the steps of this algorithm could prove in the logic, rather than informally, the equivalence between the two models in general. *In the shallow embedding style, to move simplification algorithms from the semantic interpretation to proof procedures formalizing them means to move closer to the logic in the scale of formality.*

[4] This system should be possible and not determined.
[5] This is necessary but not sufficient.

This move implies more safety for the system but it also implies more reasoning in the logic.

If Silage had specific constructs to define explicitly the periods of every signal in Silage, the interpretation of these constructs would be a straightforward and clean period semantics of Silage programs in the logic. The interpretation of these constructs would be the only information necessary to prove the equivalence of the periods of the input/output signals of two well-formed Silage programs. Proving the equivalence of the period semantics of two Silage programs would then be simple for any kind of transformation. As far as the periods are concerned, there would be no need to elaborate the ML interpretation to get a simpler model such as **PER_example'** or to prove intermediate theorems about the period semantics of two Silage programs in order to prove the correctness of the transformations in a straightforward way. Requiring that Silage programs have explicit period information might be a drawback for the CAD environment. However, this is not a serious drawback and explicit period information in Silage programs could reduce the proof effort to verify the correctness of transformations in Silage without deteriorating the quality of the embedding.

Whether or not there is explicit period information in Silage programs, the consistency of the periods has to be addressed somewhere, either informally in ML or in the logic. The difference is that, with the explicit period information, the check is simpler than without the explicit period information. Therefore, with the explicit period information, addressing the consistency of the periods informally, rather than in the logic, means less loss in the degree of formality than it would mean without having the explicit period information. With the explicit period information, the checking rules of the *check semantics* could be used in a straightforward way to verify informally the consistency of the periods in Silage programs, rather than be used to define in the logic what the periods of the signals are. The explicitness of the period information in Silage would also make it simple to address the issue of consistency of the periods formally in the logic without much proof effort. There are choices in the design of a language that can make it easier to move closer to the logic in the scale of formality, without implying serious drawbacks for the CAD environment.

5 Conclusions

We have discussed that there are degrees of formality in shallow embedding a language in HOL. The choice of the degree of formality is a trade-off between the security of the embedding and the amount and complexity of the proof effort in the logic. The design of a language could consider this verifiability issue. There are choices in the design of a language that can make it easier to improve the degree of formality, without implying serious drawbacks for the CAD environment. We have illustrated these ideas with the multi-rate semantics of Silage.

References

1. M. Gordon. "HOL: A Proof Generating System for Higher-Order Logic". In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, 1988.
2. R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. Van Tassel. "Experience with Embedding Hardware Description Languages in HOL". In V. Stavridou, T.F. Melham, and R. Boute, editors, *Proceedings of the IFIP International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 129–156. Nijmegen, The Netherlands, North-Holland, Amsterdam, June 1992.
3. T.F. Melham. "Using Recursive Types to Reason about Hardware in Higher Order Logic". In G.J. Milne, editor, *The Fusion of Hardware Design and Verification: Proceedings of the IFIP WG 10.2 Working Conference*, pages 27–50. Glasgow, North-Holland, Amsterdam, July 1988.
4. T.F. Melham. "Automating Recursive Type Definitions in Higher Order Logic". In G. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386. Springer-Verlag, 1989.
5. G. Cousineau, M. Gordon, G. Huet, R. Milner, L. Paulson, and C. Wadsworth. *The ML Handbook*. INRIA, France, 1986.
6. E.L. Gunter. "Why We Can't Have SML Style Datatype Declarations in HOL". In L. Claesen and M. Gordon, editors, *Proceedings of the IFIP International Workshop on Higher Order Logic Theorem Proving and its Applications - HOL-92*, pages 561–568. IMEC, Leuven, Belgium, Elsevier Science Publishers B. V. (North-Holland), Amsterdam, September 1992.
7. R. Boulton, M. Gordon, J. Herbert, and J. Van Tassel. "The HOL Verification of ELLA Designs". In *Proceedings of the ACM/SIGDA International Workshop in Formal Methods in VLSI Design*. Miami, FL, January 1991.
8. R. Boulton. *A HOL Semantics for a Subset of ELLA*. Technical Report 254, University of Cambridge Computer Laboratory, April 1992.
9. A.D. Gordon. *A Mechanised Definition of Silage in HOL*. Technical Report 287, University of Cambridge Computer Laboratory, February 1993.
10. A.D. Gordon. "The Formal Definition of a Synchronous Hardware-description Language in Higher Order Logic". In *ICCD92: 1992 IEEE International Conference on Computer Design: VLSI in Computers & Processors*, pages 531–534. Cambridge, Massachusetts, IEEE Computer Society Press, October 1992.
11. C.M. Angelo, L. Claesen, and H. De Man. "The Formal Semantics Definition of a Multi-Rate DSP Specification Language in HOL". In L. Claesen and M. Gordon, editors, *Proceedings of the IFIP International Workshop on Higher Order Logic Theorem Proving and its Applications - HOL-92*, pages 375–394. IMEC, Leuven, Belgium, Elsevier Science Publishers B. V. (North-Holland), Amsterdam, September 1992.
12. J. Van Tassel. *A Formalisation of the VHDL Simulation Cycle*. Technical Report 249, University of Cambridge Computer Laboratory, March 1992.
13. J. Van Tassel. "A Formalisation of the VHDL Simulation Cycle". In L. Claesen and M. Gordon, editors, *Proceedings of the IFIP International Workshop on Higher Order Logic Theorem Proving and its Applications - HOL-92*, pages 359–374. IMEC, Leuven, Belgium, Elsevier Science Publishers B. V. (North-Holland), Amsterdam, September 1992.

14. P.N. Hilfinger. "Silage, a High-level Language and Silicon Compiler for Digital Signal Processing". In *Proceedings of the IEEE 1985 Custom Integrated Circuits Conference - CICC-85*, pages 213–216. Portland, OR, May 1985.
15. P.N. Hilfinger. *Silage Reference Manual*, December 1987.
16. D. Genin, P.N. Hilfinger, J. Rabaey, C. Scheers, and H. De Man. "DSP Specification Using the Silage Language". In *Proceedings of the IEEE International Conference on Accoustics, Speech and Signal Processing*, pages 1057–1060. Albuquerque, NM, April 1990.
17. L. Nachtergaele. *A Silage Tutorial.* IMEC, Leuven, Belgium, May 1990.
18. L. Nachtergaele. *User Manual for the S2C Silage to C Compiler.* IMEC, Leuven, Belgium, May 1990.
19. H. De Man, J. Rabaey, P. Six, and L. Claesen. "Cathedral-II: a Silicon Compiler for Digital Signal Processing". *IEEE Design & Test of Computers*, 3(6):73–85, December 1986.
20. P. Lippens. *Defining Control Flow from an Applicative Specification.* Technical report, Philips Research Laboratories, Eindhoven, December 1988.
21. I. Verbauwhede. *VLSI Design Methodologies for Application-specific Cryptographic and Algebraic Systems.* PhD thesis, Katholieke Universiteit Leuven - IMEC, Leuven, Belgium, 1991.
22. J. Vanhoof. *Multi-rate Expansion for CATHEDRAL-II/III. A tutorial.* Technical report, IMEC, Leuven, Belgium, October 1992.

The University of British Columbia

Department of Computer Science,
Centre for Integrated Computer Systems Research
and Continuing Studies

present

HUG '93

HOL User's Group Workshop

August 10-13, 1993
Vancouver, B.C.