Language Independent RTL Semantics

M. J. C. Gordon

Computer Laboratory
New Museums Site, Pembroke Street
Cambridge, CB2 3QG, UK

A. Ghosh

Synopsys, Inc.
700 East Middlefield Road
Mountain View, CA 94043, USA

Abstract

A language-independent semantic framework for specifying RTL components is described and discussed. This is intended to help ensure that synthesis, formal verification and simulation can be made to agree, an essential prerequisite for RTL signoff. It is suggested that specifications be based on simple guarded assignment constructs and represented as XML documents with HDL simulation, formal logic and other views defined by style sheets. The proposal is tied to current practice by showing how it can be used to provide a semantics for part of Verilog.

This paper is a preliminary report on work arising from an ongoing project to define the semantics of RTL synthesisable HDL (initially Verilog, later VHDL).

1 Introduction

Much design is conducted at the register transfer level (RTL) and this is where many aspects of the functional behaviour of systems-on-a-chip (SOC) components will be specified. Furthermore, it seems likely that RTL will become a standard specification format for manufacture (RTL signoff).

RTL designs are normally represented in hardware description languages like Verilog or VHDL. This representation is interpreted by a variety of tools (simulators, formal verifiers, compilers etc). Unfortunately industry-standard HDLs do not determine an unambiguous semantics at the RTL level (or, in fact, at any level). The IEEE standards documents are incomplete and hard to interpret, so in practice the semantics are determined by proprietary "golden" tools, like Cadence's Verilog-XL for simulation and Synopsys Design Compiler for synthesis.

In Section 2 we describe a language-independent way of looking at RTL intended to provide a vehicle for defining HDL semantics so that synthesis, formal verification and simulation can agree, which is absolutely essential for RTL signoff. In Section 6 we tie this to current practice by outlining how our approach can be used to give a semantics to the synthesisable subset of Verilog.

Since RTL specifications will be passed between tools and may also serve as part of signoff contracts, we suggest that they be expressed via an abstract document type using a standard data exchange technology like XML (see Section 7). This avoids the need to invent yet another HDL concrete syntax. It will also make it straightforward to define different presentations of the specification (e.g. in HDL or formal logic) using appropriate style sheets.

For the one-off purposes of this paper, we use a simple throwaway notation. However, it is the structure of the constructs we propose that is significant, not their presentation here.

Since the 1960s numerous HDLs have been developed (six are described in the June 1992 issue of IEEE Design & Test [1] and another four in the September issue [2]). Most of them contain facilities for specifying RTL function. There is no need for yet another HDL, and we are not proposing one. What we are proposing are language neutral constructs for representing and exchanging the semantics of RTL designs. There already exist formalisms for representing RTL function such as SMV [3] and the synchronous languages Esterel and Lustre [4]. The proposal here is simpler (and less general) than these and is, we feel, well tailored to the particular task of specifying the RTL function of SOC components. Our contribution is that this new RTL semantics will help unify the disparate semantics used in synthesis, simulation and formal verification and so help in RTL signoff.

2 RTL semantic specification

Our goal is to provide an RTL semantic representation that combines the event-oriented approach of industry-standard HDLs (Verilog, VHDL) with the logical approach needed for formal verification notations (e.g. SMV).

The proposal here is not fully formed. There are many details that need to be elaborated (see Section 8), but we hope enough is given to provide a preliminary sketch. Where we do give details they are somewhat biased towards compatibility with Verilog. This is because the authors are more experienced with that HDL (and also because this work emerged from a project to define the semantics of synthesisable Verilog).

The core of our proposal is that a semantic specification is of a set of items (threads), each an infinitely repeating choice between a set of guarded parallel assignments to state variables (registers). The guard for each parallel assignment consists of a predicate and an HDL-style event control.

A predicate $\mathcal{P}$ is a boolean valued expression. We use a Verilog-style expression syntax for the examples here, though our proposal is that expressions be defined as a structured document type, rather than by a concrete syntax (see sections 7 and 8). However, we differ from Verilog in using = to denote equality between expressions (rather than `==`).

An event control has the form `@(`$\mathcal{T}$`)`, where $\mathcal{T}$ can either be a variable $\mathcal{V}$, a positive edge `posedge` $\mathcal{V}$, a negative edge `negedge` $\mathcal{V}$ or a disjunction of event controls $\mathcal{T}_1$ `or` $\cdots$ `or` $\mathcal{T}_n$. When the state changes, an event control $\mathcal{V}$ fires if the value of $\mathcal{V}$ changes, `posedge` $\mathcal{V}$ fires if $\mathcal{V}$ changes to 1, `negedge` $\mathcal{V}$ fires if $\mathcal{V}$ changes to 0 and $\mathcal{T}_1$ `or` $\cdots$ `or` $\mathcal{T}_n$ fires if any of $\mathcal{T}_1, \ldots, \mathcal{T}_n$ fire.

If $\mathcal{E}$ ranges over expressions, then a guarded parallel assignment has the form:

$$\mathcal{P}\ \texttt{@(}\mathcal{T}\texttt{)}\ \mathcal{V}_1 := \mathcal{E}_1,\ \cdots\ ,\ \mathcal{V}_n := \mathcal{E}_n$$

The guard is enabled when $\mathcal{P}$ is true and $\mathcal{T}$ fires. The variables assigned to, namely $\mathcal{V}_1, \ldots, \mathcal{V}_n$, are called state variables. Other variables that occur in the expressions $\mathcal{E}_1, \ldots, \mathcal{E}_n$ are input variables. When an assignment is done the current value of the state variables becomes the value of the corresponding expressions to the right of the `:=`. Input variables are interpreted with respect to the current time, but state variables in expressions have their previous value. This is made explicit in the logical semantics in Section 4.

RTL behaviour is specified by an infinite repetition of a set of guarded assignments:

```
FOREVER
  P0 @(T0) V1 := E01, ··· , Vn := E0n
  ⋮
  Pm @(Tm) V1 := Em1, ··· , Vn := Emn
END
```

This specifies the repetition forever of first choosing a true predicate, $\mathcal{P}_i$ say, then waiting until $\mathcal{T}_i$ fires and finally performing the selected parallel assignment to modify the state.

It is simplest to assume that for any state exactly one of the predicates $\mathcal{P}_0, \ldots, \mathcal{P}_m$ is true (i.e. they are mutually exclusive and exhaustive). This ensures deterministic behaviour.

It may be convenient for an RTL specification to use some auxiliary variables that are not part of the observable state. For example, the semantics of Verilog in Section 6 uses an auxiliary program counter variable `pc`. To indicate that variables $\mathcal{V}_1, \ldots, \mathcal{V}_l$ are local we precede `FOREVER`$\cdots$`END` by `LOCAL` $\mathcal{V}_1$`,` $\ldots$ `,` $\mathcal{V}_l$.

Each guarded assignment can be viewed either as a simulation thread or as a logical formula; some details are in the following two sections.

3 Event simulation semantics

A specification like:

```
FOREVER
  P0 @(T0) V1 := E01, ··· , Vn := E0n
  P1 @(T1) V1 := E11, ··· , Vn := E1n
  ⋮
  Pm @(Tm) V1 := Em1, ··· , Vn := Emn
END
```

is straightforward to simulate using a conventional HDL. For example, it may be represented in Verilog as:

```
always #0
      if (P1) @(T1)
       begin V1 <= E01; ··· ; Vn <= E0n end
 else if (P2) @(T2)
       begin V1 <= E11; ··· ; Vn <= E1n end
   ⋮
 else if (Pm) @(Tm)
       begin V1 <= Em1; ··· ; Vmn <= Emn end
```

Non-blocking assignment is used to implement parallel assignment, and the `#0` after `always` is to force the assignments to complete before the conditional tests are reevaluated.

A set of `FOREVER`$\cdots$`END` specifications is simulated by running the corresponding `always`-blocks in parallel.

The simulation cycles of VHDL and Verilog are quite complex, and are consequently hard to reason about (though not impossible [5]). A much simpler simulation cycle is adequate for our RTL specifications, and work is in progress to show that the event semantics defined by such a cyle is consistent with the logical semantics given below. It is also possible that a simple language-neutral simulation cycle for specifications will be relevant to the open model interface (OMI) objectives.[1]

4 Logical semantics

Several logical semantics are possible. A common approach used in equivalence and model checking is to represent a transition system by a formula containing primed and unprimed variables [3]. For example, the formula $x' = x + 1$ means that the value of x increases by 1 in each successive state. With this logical semantics the specification:

$$
\begin{array}{l}
\texttt{FOREVER} \\
\quad \mathcal{P}_0 \ \texttt{@(}\mathcal{T}_0\texttt{)} \ \mathcal{V}_1 \ \texttt{:=} \ \mathcal{E}_{01}\texttt{,} \ \cdots \ \texttt{,} \ \mathcal{V}_n \ \texttt{:=} \ \mathcal{E}_{0n} \\
\quad \vdots \\
\quad \mathcal{P}_m \ \texttt{@(}\mathcal{T}_m\texttt{)} \ \mathcal{V}_1 \ \texttt{:=} \ \mathcal{E}_{m1}\texttt{,} \ \cdots \ \texttt{,} \ \mathcal{V}_n \ \texttt{:=} \ \mathcal{E}_{mn} \\
\texttt{END}
\end{array}
$$

is represented (using $\cdots \ \texttt{?} \ \cdots \ \texttt{:} \ \cdots$ for the conditional) by:

$$
\begin{array}{l}
([\mathcal{P}_0]' \wedge [\![\mathcal{T}_0]\!] \ \texttt{?} \ (\mathcal{V}_1' = [\mathcal{E}_{01}]' \wedge \cdots \wedge \mathcal{V}_n' = [\mathcal{E}_{0n}]') \\
\qquad\qquad\qquad \texttt{:} \ (\mathcal{V}_1' = \mathcal{V}_1 \wedge \cdots \wedge \mathcal{V}_n' = \mathcal{V}_n)) \\
\quad \wedge \\
\quad \vdots \\
\quad \wedge \\
([\mathcal{P}_m]' \wedge [\![\mathcal{T}_m]\!] \ \texttt{?} \ (\mathcal{V}_1' = [\mathcal{E}_{m1}]' \wedge \cdots \wedge \mathcal{V}_n' = [\mathcal{E}_{mn}]') \\
\qquad\qquad\qquad \texttt{:} \ (\mathcal{V}_1' = \mathcal{V}_1 \wedge \cdots \wedge \mathcal{V}_n' = \mathcal{V}_n))
\end{array}
$$

where $[\mathcal{P}]'$, $[\mathcal{E}]'$ denote $\mathcal{P}$, $\mathcal{E}$ with all input variables (but not state variables) occurring within them primed, and

$$
\begin{array}{lcl}
[\![\mathcal{V}]\!] & \equiv & \mathcal{V}' \neq \mathcal{V} \\
[\![\texttt{posedge}\ \mathcal{V}]\!] & \equiv & \mathcal{V} \neq 1 \ \wedge \ \mathcal{V}' = 1 \\
[\![\texttt{negedge}\ \mathcal{V}]\!] & \equiv & \mathcal{V} \neq 0 \ \wedge \ \mathcal{V}' = 0 \\
[\![\mathcal{T}_1 \ \texttt{or} \ \cdots \ \texttt{or} \ \mathcal{T}_n]\!] & \equiv & [\![\mathcal{T}_1]\!] \ \vee \ \cdots \ \vee \ [\![\mathcal{T}_n]\!]
\end{array}
$$

The logical semantics of a collection of `FOREVER`$\cdots$`END` items is obtained by conjoining the formulae from each item.

Modelling a transition system with formulae containing primed and unprimed variables is one common approach. However, there are other logical models, e.g. associating each variable with a function of time representing the sequence of successive values it takes on [6]. This is often used in formal verification with theorem provers.

We expect that several logical views of a specification will be needed. By defining them explicitly and openly using style sheets (see Section 7) it is hoped to ensure that all the views are consistent and hence that verifications based on them are compatible.[2]

5 Examples

In this section we give three standard examples: some combinational logic, a level sensitive latch and an edge-triggered flip-flop. For each we give an RTL specification and show the logical view described in Section 4 (the event simulation views are obvious, so are omitted).

In the three examples the predicate part of the guard is `true`. The semantics of Verilog in the next section illustrates the need for non-trivial predicates in guards.

Combinational logic is specified by making the outputs track any changes to the inputs. For example, a combinational adder (inputs `b`, `c` and output `a`) is specified by:

```
FOREVER
  true @(b or c) a := b+c
END
```

where `true` denotes the universally true predicate (`1` in Verilog). The logical view is:

$$([\texttt{true}]' \wedge [\![\texttt{@(b or c)}]\!]) \ \texttt{?} \ \texttt{a}' = [\texttt{b} + \texttt{c}]' \ \texttt{:} \ \texttt{a}' = \texttt{a}$$

which, since $[\texttt{true}]' = \texttt{true}$ and `b` and `c` are input variables, is logically equivalent to:

$$(\texttt{b}' \neq \texttt{b} \vee \texttt{c}' \neq \texttt{c}) \ \texttt{?} \ \texttt{a}' = \texttt{b}' + \texttt{c}' \ \texttt{:} \ \texttt{a}' = \texttt{a}$$

This clearly maintains invariant $\texttt{a} = \texttt{b} + \texttt{c}$, as required for combinational logic.

Consider now a level-sensitive latch:

```
FOREVER
  true @(phi1 or d) q := phi1 ? d : q
END
```

The logical view of this is:

$$
\begin{array}{l}
(\texttt{true}' \wedge [\![\texttt{@(phi1 or d)}]\!]) \ \texttt{?} \ \texttt{q}' = [\texttt{phi1?d:q}]' \\
\qquad\qquad\qquad\qquad\qquad\quad \texttt{:} \ \texttt{q}' = \texttt{q}
\end{array}
$$

In this example `phi1` and `d` are input variables, but `q` is a state variable. Thus $[\texttt{phi1?d:q}]' = \texttt{phi1}'\texttt{?d}'\texttt{:q}$. Hence the formula above is equivalent to:

[1] See `www.vhdl.org/omf/`.

[2] The language XSL for defining style sheets is not yet finalised, but it is believed that it will be powerful enough to compute the various different views. If not, some other mechanism will be needed.

$$(\texttt{phi1}' \neq \texttt{phi1} \ \lor \ \texttt{d}' \neq \texttt{d}) \ \texttt{?} \ \texttt{q}' = \texttt{phi1}'?\texttt{d}'\texttt{:q}$$
$$\texttt{:} \ \texttt{q}' = \texttt{q}$$

This maintains the invariant $\texttt{phi1} = 1 \Rightarrow \texttt{q} = \texttt{d}$ and when $\texttt{phi1} \neq 1$, `q` holds the value of `d` when `phi1` was last 1.

Finally, consider the specification of an edge-triggered flip-flop:

```
FOREVER
  true @(posedge clk) q := d
END
```

The logical view of this is:

$$([\mathsf{true}]' \land [\![\texttt{@(posedge clk)}]\!]) \ \texttt{?} \ \texttt{q}' = [\texttt{d}]' \ \texttt{:} \ \texttt{q}' = \texttt{q}$$

which is equivalent to:

$$(\texttt{clk} \neq 1 \land \texttt{clk}' = 1) \ \texttt{?} \ \texttt{q}' = \texttt{d}' \ \texttt{:} \ \texttt{q}' = \texttt{q}$$

Thus `q` holds the value of `d` at the end of the last positive edge.

6 Semantics of synthesizable Verilog

We outline here a semantics of synthesizable Verilog. The approach is to provide a translation from a Verilog specification (i.e. a collection of modules) to a set of `FOREVER`$\cdots$`END` items. This translation could be viewed as a naive synthesis algorithm (indeed, it was inspired by the algorithm used in the CSYN Verilog compiler [7]). However, the role of the translation is the explication of semantics not the generation of hardware, it is thus optimised for clarity and formal tractability, not efficiency.

A Verilog specification consists of a collection of modules. The first step is to elaborate the top-level module by replacing all instances by the module bodies, renaming variables to avoid clashes. Similarly tasks can be eliminated by inlining. The result of this is a single module consisting of continuous assignments, `always`-blocks and function declarations.

A continuous assignment `assign` $\mathcal{V}$ `=` $\mathcal{E}$ is translated to

$$\texttt{always @(}\mathcal{V}_1 \texttt{ or } \cdots \texttt{ or } \mathcal{V}_r\texttt{)} \ \mathcal{V} \ \texttt{:=} \ \mathcal{E}\texttt{;}$$

where $\mathcal{V}_1$, ..., $\mathcal{V}_r$ are the variables (wires or registers) occuring in $\mathcal{E}$.

Each module item of the form `always @(`$\mathcal{T}$`)` $\mathcal{S}$ can be translated to one or more guarded assignments as described below.

Let $\mathcal{V}_1$, ..., $\mathcal{V}_m$ be the state variables, i.e. the variables occurring on the left of assignments in $\mathcal{S}$.

Introduce a new variable called `pc` (say) for "program counter". This will range over $\{0, 1, \ldots, n\}$, where there are $n+1$ occurrences of event controls in $\mathcal{S}$. The event control $\mathcal{T}$ immediately following `always` is numbered 0 and the initial value of `pc` is taken to be 0. A Verilog module item `always` $\mathcal{S}$ is then translated to a specification of the form:

$$
\begin{array}{l}
\texttt{LOCAL pc} \\
\texttt{FOREVER} \\
\quad \texttt{pc=0 @(}\mathcal{T}_0\texttt{)} \\
\quad\quad \texttt{pc := } \mathcal{E}_0,\ \mathcal{V}_1 \texttt{ := } \mathcal{E}_{01},\ \cdots\ ,\ \mathcal{V}_m \texttt{ := } \mathcal{E}_{0m} \\
\quad\quad\quad\quad\quad\quad\quad\quad \vdots \\
\quad \texttt{pc=}n\texttt{ @(}\mathcal{T}_n\texttt{)} \\
\quad\quad \texttt{pc := } \mathcal{E}_n,\ \mathcal{V}_1 \texttt{ := } \mathcal{E}_{n1},\ \cdots\ ,\ \mathcal{V}_m \texttt{ := } \mathcal{E}_{nm} \\
\texttt{END}
\end{array}
$$

The expressions $\mathcal{E}_i$, $\mathcal{E}_{i1}$, ..., $\mathcal{E}_{im}$ are computed by symbolically executing from the event control numbered i until another event control is reached. Rather than give the details of this symbolic execution here, we will illustrate the idea with an example:

```
always
 @(posedge clk)                  \*0*\
 begin
  case (state)
  0:  begin total = data;
            @(negedge clk)       \*1*\
            state = 1;
      end
  1:  begin total = total + 1;
            @(negedge clk)       \*2*\
            state = 2;
      end
  default:
      begin total = total + 2;
            @(negedge clk)       \*3*\
            state = 0;
      end
  endcase
 end
```

There are four occurrences of event controls here: one `posedge clk` and three occurrences of `negedge clk`. Thus `pc` can take on four values: 0, 1, 2, 3 as indicated in the comments in the Verilog code above. The state variables are `state` and `total` (`data` and `clk` are inputs).

Initially `pc` $= 0$ and a positive edge on `clk` is awaited. When this occurs what happens depends on the value of `state`. If `state` $= 0$, then the value of `data` is stored in `total` and the first `negedge clk` is reached, so `pc` becomes 1. If `state` $= 1$ then `total` is incremented and the second `negedge clk` is reached, so `pc` becomes 2. If `state` is neither 0 or 1 then `total` is incremented by 2 and the third `negedge clk` is reached, so `pc` becomes 3. The variable `state` is not updated during the initial cycle.

By considering the (simpler) flow of control from the other event controls, we arrive at the following semantic representation (note that = denotes equality, not assignment):

```
FOREVER
  pc=0 @(posedge clk)
    pc := state=0 ? 1 : state=1 ? 2 : 3,
    total := state=0
              ? data
              : state=1 ? total+1 : total+2,
    state := state
  pc=1 @(negedge clk)
    pc := 0,
    total := total,
    state := 1
  pc=2 @(negedge clk)
    pc := 0,
    total := total,
    state := 2
  pc=3 @(negedge clk)
    pc := 0,
    total := total,
    state := 0
END
```

This example should give the idea of our semantics. A detailed algorithm based on this idea for transforming synthesizable Verilog to an RTL semantic specification is straightforward but tedious to describe.[3] Care is needed for the correct handling of block exit (`disable`) and non-blocking assignment. The semantics assumes that all loops are broken by an event control (if not, symbolic execution could loop); this is a condition for the Verilog to be well-formed synthesisable RTL.

Functions in Verilog cannot contain event controls, as they are intended to denote combinational logic. Their bodies are statements containing an assignment to the function's name. The result returned by the function is the value of the function's name when the execution of the body completes. The body can be symbolically executed to calculate an expression containing the function's formal parameters as free variables and which evaluates to the result of calling the function. A function call can then be replaced by an instance of this expression, with the actual parameters of the call substituted for the formal parameters.

The semantics of Verilog outlined above makes a number of arbitrary choices. For example, it encodes the program counter as a number. An alternative (taken by CSYN) is to use one-hot control. Such choices have an impact on implementations (one-hot is faster, but uses more registers) but should not be an observable part of the semantics – i.e. one wants the logical view of a semantics based on one-hot control to be equivalent to the logical view of the semantics just outlined. This is the reason why the program counter is hidden using the `LOCAL` contructs.

Although sketchy, we hope this section shows how straightforward it is to convert synthesisable HDL into the proposed RTL format.

7 Brief review of XML

XML[4] is a standard set of conventions for representing structured data as text files. It provides a way of separating the "content" of data from particular presentations. The structure is represented by an XML document type and presentations are defined by style sheets. The data is viewed with respect to a style sheet using a browser (preliminary support for XML in Internet Explorer is already available; Netscape will follow soon). A familiar example of a document type is HTML, which is used for web pages. A well formed HTML document has a title and a body; the body can contain various constructs such as lists of items, pictures, programs (Java) etc. XML allows custom document types to be defined that are tailored to particular applications. We would define a document type to represent RTL specifications as having a set of items, with each item having a guard and assignments etc. This is just like a datatype definition in a high level language, but the data consists of text files and so can easily be passed between programs. We would define style sheets that enabled specifications to be viewed in application specific ways, e.g. as HDL (e.g. VHDL or Verilog) or as logical formulae. By not having concrete syntax (e.g. BNF) as primary, one both reduces "language-wars" issues and simplifies the passing of specifications between applications. For example, an application does not need to have a specification parser, it can just use an off-the-shelf API from one of several already-available XML parsers.

8 Future directions

In the short term, the RTL semantics here needs to be augmented with (i) some structuring mechanism (e.g. a module hierarchy) and (ii) a detailed definition of expressions. We have been using Verilog's constructs for (i) and (ii), but a simpler and language-independent approach is needed. The complex bit-width coercion rules in Verilog make the statement

[3] See `www.cl.cam.ac.uk/users/mjcg/HDLsemantics`.

[4] See `www.w3.org/TR/PR-xml`.

of logical laws complicated, and we expect to use a much simpler type system. A simple treatment of undefined and don't-care situations is needed, and also a treatment of tri-state components. These details are orthogonal to the main idea of guarded assignments presented here. Once a set of specification contructs is defined, document types in XML can be defined to represent them and style sheets written to provide HDL and logical presentations.

In the medium term we plan to write out the semantics of synthesisable Verilog in detail, and to implement a tool that will convert Verilog texts to specification documents. At Cambridge we already have programs written in Standard ML that compute the explicit bit-widths of all expressions (by Myra VanInwegen) and extract guarded assignments by symbolic execution (by Mike Gordon and Daryl Stewart).

We hope to validate our semantics so that it is accepted as conforming to the implicit industry-standard RTL interpretation of synthesisable Verilog. We plan to make the semantics, supporting XML documents and semantics-extraction tool available in the public domain.

9 Summary and Conclusions

The main general ideas here are first that the RTL functional semantics be expressed in a clean and simple form that meshes well with simulation and formal specification and second that this form be expressed not via a concrete syntax, but as a document type supported by style sheets that allow it to be presented as HDL or as logical equations. We have shown that, although simple, our semantics connects with reality in that synthesisable Verilog can be mapped into it.

Our overall goal is to define the semantics of HDLs so that synthesis, formal verification and simulation can agree. By having a semantics with simple simulation and logical views, we have a hope of ensuring by theoretical analysis that simulation and formal verification agree. Work is already in progress on this. To ensure synthesised hardware matches the models used by verification, it is either necessary to verify the synthesis tools, which is impractical for commercial code, or to check the outputs of each synthesis run are logically equivalent to the inputs. This latter approach is what is normally done. Our contribution is to propose a way of making explicit the specification against which the implementation must be checked.

Acknowledgments

M. Gordon acknowledges the influence of David Greaves on the work here, especially his CSYN compiler, and also conversations with Ken Larsen, Peter Sewell, Konrad Slind, Daryl Stewart and Myra VanInwegen. Richard Boulton of the University of Edinburgh suggested some improvements and pointed out numerous errata. The work at Cambridge was supported by EPSRC grant GR/K57343 (Principal Investigator D. Greaves). Luca Cardelli of Microsoft Research suggested using XML and Henry Thompson of the Language Technology Group at the University of Edinburgh provided advice on its use.

References

[1] Y. Chu, D. L. Dietmeyer, J. R. Duley, F. J. Hill, M. R. Barbacci, C. W. Rose, B Johnson, M. Roberts, "Three Decades of HDLs, Part 1: CDL Through TI-HDL", *IEEE Design & Test of Computers*, June, 1992.

[2] D. Borrione, R. Piloty, D. Hill, K. J. Lieberherr, P. Moorby, "Three Decades of HDLs, Part 2: Conlan Through Verilog", *IEEE Design & Test of Computers*, September, 1992.

[3] K. L. McMillan, *Symbolic Model Checking*, Kluwer, 1993.

[4] N. Halbwachs, *Synchronous Programming of Reactive Systems*, Kluwer, 1993.

[5] D. Borrione, *Special Issue on VHDL Semantics*, *Formal Methods in System Design*, Vol. 7, Number 1/2, August, 1995.

[6] M. J. C. Gordon, "Why higher-order logic is a good formalism for specifying and verifying hardware", in *Formal Aspects of VLSI Design* edited by G. Milne and P. A. Subrahmanyam, North-Holland, 1986.

[7] D. J. Greaves, "CSYN Verilog Compiler: Professional Reference Manual", contact D. Greaves for details: `www.cl.cam.ac.uk/users/djg/`.