Using the HOL Prove Assistant for proving the Correctness of Term Rewriting Rules reducing Terms of Sequential Behavior

Matthias Mutz
Universität Passau, Fakultät für Mathematik und Informatik,
Lehrstuhl für Rechnerstrukturen, Innstraße 33, D-8390 Passau, Germany

Abstract

There are several approaches of using automated theorem provers and assistants in hardware verification. It has been shown, that hardware behavior can be modelled and verified using theorem proving tools. But the task of generating a proof remains difficult and often needs a big amount of interaction. Therefore, the methods of our hardware verification system VERENA are based on term rewriting. It is shown how we use the expressive power of type theory to model circuit behavior. The crucial point in implementing a term rewriting system is to guarantee that the term rewriting rules used have specific properties like correctness, confluence, completeness, etc. It is demonstrated how we use the HOL prove assistant to prove the correctness of term rewriting rules.

1. Introduction

There are several approaches of using computer assisted theorem proving in hardware verification. For example, the Boyer-Moore prover [BoMo 79] establishes a partially automated theorem proving system. VERITAS [HaDa 85] provides a specification languange and a computational implementation for theorem proving. The HOL prove assistant [HOL 88] supports theorem proving with user-defined prove strategies.

Hardware verification by using theorem provers has been demonstrated successfully. For example, in [Pie 89, BrCa 89] the Boyer-Moore prover is used to verify the Min-Max sequential benchmark circuit and a pipeline, respectively. HOL is used in [RaC 89] to verify cascading properties of a parallel sorting circuit. But fully automated theorem proving is very difficult to achieve. To automate the verification of hardware as much as possible, algebraic simplification methods like rewriting techniques have been suggested, e.g. [Lar 88].

We developed some proof strategies for hardware verification and implemented them in our verification system VERENA [Gra 88]. We decided to use term rewriting methods to transform behavior descriptions and implementation descriptions into canonical representations, which can easily be checked for equivalence. The approach becomes practicable since the verification goals are partitioned into manageable parts. The close relationship between term representations and specifications based on type theory let us decide to use the HOL prove assistant to prove the correctness of the term rewriting rules. By this means we have a strict seperation of two tasks: a manager of VERENA establishes term rewriting rules verified by assistance of HOL while a user of VERENA is only concerned with defining algebraic specifications based on the introduced functions.

Our tools for the verification of descriptions concerning sequential designs have to deal with specific representations of temporal relationships. In similar approaches based on Temporal Logic, e.g. [BrCDM 85, FuKTM 86], also in approaches aiming at the verification of synchronous hardware, e.g. [Pai 87], the advantages by abstracting from time are shown. These approaches use modal operators for hiding the time

parameter. In these approaches the user of the verification system is much concerned with theoretical aspects of specification and verification. Tools with close relationships to mathematics may be hard to handle by designers. We think, the VERENA user interface is much more oriented in hardware design.

In Paragraph 2, we motivate the use of term rewriting rules for the verification of sequential behavior descriptions. Paragraph 3 shows how we model circuit behavior expressing time relations explicitly and how we use HOL specifications to define the semantics of temporal expressions. We introduce time dependant operators and give some examples for operators used. In Paragraph 4, we introduce operators used in our hardware description language (VIOLA) based on HOL definitions hiding the time parameter. We show how to prove the correctness of term rewriting rules given in terms of VIOLA using the HOL prove assistant. As an example, a term rewriting rule is applied to a latch specification. Finally, we give some results and conclusions.

2. Hardware verification with VERENA using term rewriting techniques

The structure of a logical system is given as a hierarchical netlist: a structural refinement of a logical system has structural components (each defined as a logical system) with input, output, and bidirectional terminals and nets connecting the terminals of the components and the terminals of the refined logical system. The behavioral description of the logical system is derived from the composition of the behavioral descriptions of the components.

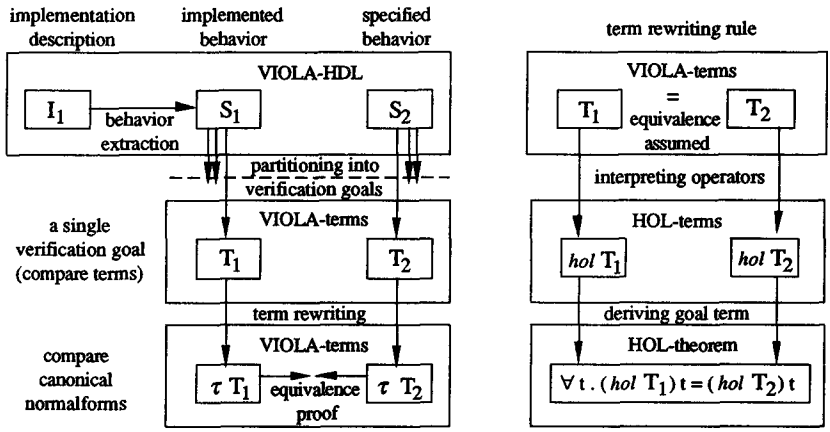
The verification task is to show, that an implemented behavior satisfies the restrictions established by the specification of the expected behavior. The implementation terms are derived from the implementation description defined by a netlist using our hardware description language VIOLA. The expected sequential behavior is given in terms of an *interval dependency graph* (IDG). The term representations derived from the implementation description are valid for all time instants. The IDG defines time intervals, in which input functions and expected output functions are valid in the specific interval. By this way, the verification task is partitioned into several verification goals, where for each verification goal two term representations are to be compared. A detailed description of the derivation of subgoals is behind the scope of the paper and it will be given elsewhere [Mut 91].

To make the strategy more clear we consider two extreme IDGs. An IDG consisting of two nodes t_1 and t_2 has assigned the complete behavior of the considered system. Additional assertions may define relations between the set of input and output signals of the implemented and specified behavior. Another extreme IDG has time intervals corresponding to clock cycles and establishes different behavior and relations for each clock cycle. The first approach establishes difficult specification and verification tasks while the second approach needs to much details about the implementation for specifying the behavior. Our approach intends that the user selects a description between the two extremes.

As a subgoal, we have to prove that two terms, one derived from the implemented behavior description and one derived from the specified behavior description, are equivalent. Fig. 2.1 a) shows the transformation steps used in hardware verification with VERENA.

The crucial point in implementing a term rewriting system is the guarantee that the term rewriting rules used have specific properties like correctness, confluence, completeness. Fig. 2.1 b) shows how we prove the *correctness* of term rewriting rules using the HOL prove assistant. At first, the left side of an equation defining the rewriting rule and the right side are transformed into two HOL terms depending on the

interpretation given for the used operators. Then time is introduced to build a HOL theorem. If we can show that the theorem holds we have proved, that the term rewriting rule is correct with respect to the HOL-interpretation.



a) verification of hardware descriptions Fig 2.1 b) verification of term rewriting rules

3. Formal behavior description: explicit time

For the definition of sequential behavior, one has to use formalisms in which to represent temporal precedence and other temporal relationships. The ad hoc way is to use an explicit time parameter *t* denoting an instant of time taken from a set that is isomorphic to the reals, the integers, or the natural numbers depending on the systems to be described and the manipulations of descriptions required. In our approach time is isomorphic to the integers.

In the next section we introduce operators that model sequential behavior with explicit time. All axioms, definitions, theorems, and proofs are specified or generated using the HOL prove assistant. Together they form a new theory. The reader should be familiar with standard type theory and predicate calculus notation. We will use formalisms according to [Gor 85].

3.1 Time base

The discrete time base is represented by the type *time*. The set of instants of type *time* is isomorphic to the integers. We employ the properties of integers according to [Zei 76]. The constant "-∞:time" denotes the fictitious lower bound. The operators "π:time→time" and "σ:time→time" are introduced to denote the instant (π t) directly preceeding t and the instant (σ t) directly following t. The constant -∞ and the two operators are set into relation by the following axioms:

$\Vdash \forall t\ t'.\ (t' = \pi\ t) = (\sigma\ t' = t)$ [1] [πσ_ax1]

$\Vdash \pi\ -\infty = -\infty$ [π_-∞]

[1] We use ⊩ to denote an axiom or a definition. ⊢ is used to denote a theorem, derived from axioms, definitions, and previously derived theorems.

The *time functions* "**tpred**:num→time→time" and "**tsucc**:num→time→time" determining the order of instants are given as definitions satisfying the Primitive Recursion theorem [Gor 85]:

$\Vdash (\forall t.\ \mathbf{tpred}\ 0\ t = t) \wedge (\forall n\ t.\ \mathbf{tpred}\ (\mathbf{suc}\ n)\ t = \pi\ (\mathbf{tpred}\ n\ t))$ [2] [tpred_def]

$\Vdash (\forall t.\ \mathbf{tsucc}\ 0\ t = t) \wedge (\forall n\ t.\ \mathbf{tsucc}\ (\mathbf{suc}\ n)\ t = \sigma\ (\mathbf{tsucc}\ n\ t))$ [tsucc_def]

For example, the following theorems are derived:

$\vdash (\forall n.\ \mathbf{tsucc}\ -\infty = -\infty)$ [tsucc_-∞]

$\vdash (\forall n\ t.\ \mathbf{tsucc}\ n\ (\mathbf{tpred}\ n\ t) = t) \wedge (\forall n\ t.\ \mathbf{tpred}\ n\ (\mathbf{tsucc}\ n\ t) = t)$ [tsucc_tpred]

Due to the discrete time-base, the **tpred** operator may be compared with the **P** operator in [BoPP 88] describing a past occurence of a synchronizing event. We do not use the **tpred** operator to distinguish between two synchronization points. Instead, we interpret the "distance" between two instants as the smallest time between two distinguishable instants. We use explicit event functions to figure out synchronization points. Therefore, the **P** operator is better compared with the **llast** operator introduced later.

3.2 Signals and event functions

Signals are modelled by functions of type *signal* = time→num list. A signal therefore represents a time-dependant logical *vector* function.

Functions assigning each instant a boolean value are called *event functions*. Event functions are used to determine the instants of synchronizing events. The type of event functions is abbreviated as *event* = time→bool. An event function *e* defines the occurrences of an event *E*: an event *E* occurs at instant *t*, iff *e*(t)=T holds.

3.3 Sequential behavior

Besides **tpred** and **tsucc**, the basic operators for describing sequential behavior are **llast**, **llastn**, and **stable**:

llast	:	event → time → time
llastn	:	num → event → time → time
stable	:	num → signal → time → bool

The introduction of the **llast** operator is inspired by [AmCH 86] where a time base isomorphic to the reals is used. It is also closely related to the use of the **delta** function introduced by [Eve 86] where a time base isomorphic to the natural numbers is used.

The **llast** operator is defined as [llast_def]:

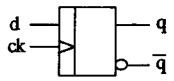
$\Vdash \mathbf{llast}\ e\ t = (\neg\exists n.\ e\ (\mathbf{tpred}\ n\ t)) \Rightarrow -\infty \mid \mathbf{tpred}\ (\varepsilon n.e\ (\mathbf{tpred}\ n\ t) \wedge \forall m.\ m<n \supset \neg(e\ (\mathbf{tpred}\ m\ t)))\ t$ [3] [4]

---

[2] **suc** is the successor function defined for natural numbers (**suc** n = n+1)

[3] ε denotes the choice operator. εx.P(x) denotes some value of type σ, *a* say, such that P(a) is true. If there is no *a* of type σ such that P(a) is true, εx.P(x) denotes a fixed but unspecified value of type σ [Gor 85]. It holds $(\exists x.P(x)) \equiv P(\varepsilon x.P(x))$.

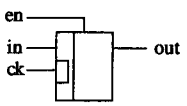
If *e* is the event function of an event *E*, e.g. a synchronization event like the edge of a trigger signal, the term "**llast** e t" associates with each instant t the instant of the last occurence of *E* loosely preceeding *t*. If no such event exists, we define the term to represent the fictitious lower bound of *time*. Fig 3.1 gives typical examples for the use of the **llast** function in the description of memory elements. The logical values are denoted by H (high), L (low) and Z (tristate). The range of the signals is {L,H} except *out* which has range {L,H,Z}.



$q(t) = d(\textbf{llast}\ up(ck)\ t)$
$\overline{q}(t) = not(q(t))$

$up(ck,t) = (ck(t)=H) \wedge (ck(\textbf{tpred}\ 1\ t)=L)$

a) positive edge triggered D-flipflop



$out(t) = (en(t)=L) \Rrightarrow Z \mid in(\textbf{llast}\ and(en,ck)\ t)$

Fig 3.1

b) latch with enable

While **llast** determines the last occurence of an event *E*, we may be interested in the instant of the last occurrence strictly before "**llast** e t" or the n^{th} occurence of *E* before *t*. For this sake, we introduce the **llastn** operator. The **llastn** operator is defined by Primitive Recursion :

$$\vdash (\forall e\ t.\textbf{llastn}\ 0\ e\ t = \textbf{llast}\ e\ t) \wedge (\forall n\ e\ t.\textbf{llastn}\ (n+1)\ e\ t = \textbf{llastn}\ n\ e\ (\textbf{tpred}\ 1\ (\textbf{llast}\ e\ t))) \quad [llastn\_def]$$

The boolean term "**stable** n s t" is true at an instant *t*, iff the value of the signal *s* at *t* is equal to the value of s at t-1, t-2, ..., t-n. The definition of the **stable** operator is:

$$\vdash (\forall s\ t.\textbf{stable}\ 0\ s\ t = T) \wedge \quad [stable\_def]$$
$$(\forall n\ s\ t.\textbf{stable}\ (\textbf{suc}\ n)\ s\ t = (s\ t = s\ (\textbf{tpred}\ 1\ t)) \wedge (\textbf{stable}\ n\ s\ (\textbf{tpred}\ 1\ t)))$$

The stable operator is used in combinations defining event functions.

4. Hardware description language: abstracting from time

We aim at behavioral descriptions abstracting from time, i.e. there is no explicit use of the time parameter *t*. Time-dependency is introduced by the usage of *modal* operators, i.e. operators with implicite consideration of the time parameter. The time parameter *t* is explicitly used in the definitions of modal operators.

In our approach of hardware verification, we have to reduce terms only containing signal variables, combinational or modal operators. For the reduction process, we use sets of term rewriting rules depending on the nature of the verification goal. The correctness of these rules is imperative for the correctness of the verification results.

We introduce a HOL definition for each operator defined in VIOLA. It is not the actual syntax of VIOLA terms, but it is the corresponding HOL term notation.

---

[4] The conditional operator $\Rrightarrow$ is defined by $(b \Rrightarrow t_1 | t_2) = \varepsilon t.((b=T) \supset (t=t_1)) \wedge ((b=F) \supset (t=t_2))$.

4.1 Operators

We introduce operators combining natural numbers and signals to obtain new signals:

TPRED	:	num → signal → signal
TSUCC	:	num → signal → signal
LLASTN	:	num → signal → signal → signal
STABLE	:	num → signal → signal

The HOL-definitions are given as λ-terms:

(TPRED n s)	=	λt. s (**tpred** n t)	[TPRED_def]
(TSUCC n s)	=	λt. s (**tsucc** n t)	[TSUCC_def]
(LLASTN n es s)	=	λt. s (**llastn** n (λt'. es t' = [H]) t)	[LLASTN_def]
(STABLE n s)	=	λt. (**stable** n s t) ↠ [H] | [L]	[STABLE_def]

[H] denotes a single digit logical vector. Single digit signals may represent event functions. We use H=2 (high) to represent T (true) and L=1 (low) to represent F (false).

4.2 Verification by theorem proving

The HOL interpretation of a term rewriting rule $T_1 = T_2$ is obtained by transforming the rule according to the given λ-term definitions of the used operators:

$$T_1 = T_2\ [v_1, v_2, ...] \quad \xrightarrow{HOL} \quad \forall\, v_1\, v_2\, ...\, t.\ (T_1)\, t = (T_2)\, t$$

$[v_1, v_2, ...]$ denotes the set of free variables occuring in T_1 and T_2.

We say, that the rewriting rule is *correct with respect to the HOL-interpretation* (or simply *correct*), iff the HOL-term obtained is a theorem of our theory that is based on the predefined theories (like theories on natural numbers, lists, booleans). To prove the correctness of a term rewriting rule, we enter the HOL representation of the rule containing the combinational and modal operators as a goal for theorem proving. Then we use the definitions of the modal operators to get an equation with an explicit time parameter *t*.

Unlike other theorem provers like Boyer-Moore, there are no automated proof generation strategies in the HOL system. To generate the theorems that follow the user has to tell the system how to do the proofs. There is much support to do this job. The next examples illustrate some tools typically used in the verification of our term rewriting rules supporting the construction of a proof.

4.3 Verification examples

One way to prove a term to be a theorem of a given theory is to derive T (true) using the inference rules, definition, axioms, and pre-proved theorems given by the HOL proof assistant. This kind of theorem proving is called *goal directed*. In our examples, we illustrate the use of some *tactics* defining how to use previously derived theorems and inference rules.

As an example, we give the first steps of a goal-oriented proof of the following HOL-representation of a term rewriting rule. The goal is defined by

```
g " ∀n m s t. ( TPRED n ( STABLE m s ) ) t = ( STABLE m ( TPRED n s ) ) t "
```

The first step in verifying a rule is to get the term representation containing time functions. The actual command in HOL-88 is:

```
e ( REWRITE_TAC[TPRED_def;STABLE_def] THEN BETA_TAC )
```

The new goal term is obtained by rewriting the goal term using the equations given as definitions of TPRED and STABLE as left to right rewriting rules and by β-conversion:

➥ ∀n m s t. (**stable** m s (**tpred** n t)) ⇏ [H] ! [L] = (**stable** m (λt'. s (**tpred** n t')) t) ⇏ [H] ! [L]

The proofs of theorems containing operators defined by Primitive Recursion are mostly done by induction over natural numbers. The HOL-88 command

```
expand ( INDUCT_TAC )
```

splits the goal into two subgoals

➥ ∀m s t.(**stable** m s (**tpred** 0 t)) ⇏ [H] ! [L] = (**stable** m (λt'.s (**tpred** 0 t')) t) ⇏ [H] ! [L]

and

➥ ∀m s t.(**stable** m s (**tpred** (**suc** n) t)) ⇏ [H] ! [L] = (**stable** m (λt'.s (**tpred** (**suc** n) t')) t) ⇏ [H] ! [L]

with the assumption

∀m s t. (**stable** m s (**tpred** n t)) ⇏ [H] ! [L] = (**stable** m (λt'. s (**tpred** n t')) t) ⇏ [H] ! [L]

The first goal is solved by rewriting with the **tpred** definition and by η-conversion. The second goal is solved by induction over *m*: for P(0) we rewrite with the **stable** definition. The proof for P(m)⊃P(**suc** m) is a little bit more complicated to notate (but not very complicated to prove) and therefore behind the scope of this paper.

As a second example we give a sketch of the proof for the following rule:

```
g " ∀n m es s t.  ( TSUCC m ( LLASTN n ( TPRED m es ) s ) ) t =
                  ( LLASTN n es ( TSUCC m s ) ) t "
```

The derived goal is obtained by the basic rewritings:

```
e ( REWRITE_TAC[TPRED_def;TSUCC_def;LLASTN_def] THEN BETA_TAC )
```

that results in

➥ ∀n m es s t. s (**llastn** n (λt'. es (**tpred** m t') = [H]) (**tsucc** m t))
= s (**tsucc** m (**llastn** n (λt'. es t' = [H]) t))

We want to prove the following lemma:

$\forall n\ k\ e\ t.$ **llastn** n ($\lambda t'$.e(**tpred** k t')) (**tsucc** k t) = **tsucc** k (**llastn** n e t)

The lemma is formulated as a subgoal, and the subgoal term is used as a rewriting rule applied to the term of the original goal. We first have to solve the subgoal to continue with the proof of the transformed original goal:

```
e ( SUBGOAL_THEN
      "∀n k e t.llastn n (λt'.e(tpred k t')) (tsucc k t) = tsucc k (llastn n e t)"
      (λth. REWRITE_TAC[th]) )
```

The goal derived from the original goal with the assumption of the subgoal term is

➥ $\forall n\ m\ es\ s\ t.$ s (**tsucc** m (**llastn** n (es t = [H])) = s (**tsucc** m (**llastn** n ($\lambda t'$. es t' = [H]) t))

We start with induction over *m*. For P(0) we rewrite with the definitions of **llastn** and **llast** to obtain

➥ **tsucc** k ($\neg\exists$n. e(**tpred** n t)) ⇛ -∞ | **tpred** (εn. e(**tpred** n t) ∧ ∀m. m<n ⊃ ¬e(**tpred** m t)) t)
= ($\neg\exists$n. e(**tpred** n)) ⇛ -∞ | **tpred** (εn. e(**tpred** n t) ∧ ∀m. m<n ⊃ ¬e(**tpred** m t)) (**tsucc** k t)

This goal is solved by a boolean case decision

```
e ( ASM_CASES_TAC "∃n. e(tpred n t)" )
```

obtaining the two subgoals

➥ **tsucc** k (**tpred** (εn. e(**tpred** n t) ∧ ∀m. m<n ⊃ ¬e(**tpred** m t)) t)
= **tpred** (εn. e(**tpred** n t) ∧ ∀m. m<n ⊃ ¬e(**tpred** m t)) (**tsucc** k t)

and

➥ **tsucc** k -∞ = -∞

The two goals are easily proved using rewriting tactics.

4.4 Application example

To show the use of term rewriting rules, we give the term representing the function of a latch (memory element) with inputs *in* and *ck* and output *out* regarding setup, hold, and delay times:

```
out   = ( TPRED Δd ( LLASTN 1 ( OR ck↓ ck ) ( IF error X ( LLASTN 1 ck in ) ) ) )
error = ( AND ( ck↓ ( NOT ( TSUCC Δh ( STABLE ( Δs+Δh ) in ) ) ) )
ck↓   = ( AND ( NOT ck ) ( TPRED 1 ck ) )
```

We first consider the times of possible changes of the output *out*. If *ck* is L (low) no change can occur. If *ck* is H (high) the value of input *in* is observed at *out* delayed by Δd time units. At the falling edge (*ck* changes from H to L) the setup and hold condition must be considered: the output becomes X (undefined) if the input is not stable since Δs time units or if it is not stable for the next Δh time units. Because the input value

must not change at the falling edge, we formulate the error condition as follows: the input value was not stable since Δs+Δh time units Δh time units after the falling edge of *ck*.

To illustrate the effect of term rewriting we give a structural representation of terms. Each subterm with an operator *op* is represented by an one-output / multi-input component marked with *op*. Fig 4.1 is the structural representation of the latch term.

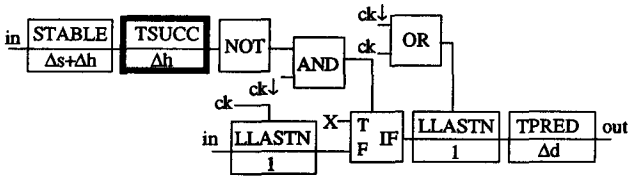


Fig. 4.1: structural representation of the latch term regarding setup, hold, and delay times

We use a confluent and noetherian set of term rewriting rules to "move" TSUCC occurences from the inputs to the outputs of components. We use this set to remove the TSUCC component.

We have proved, that an occurence of a [TSUCC n] component at any input of a *combinational component* can be moved to the output if we insert a [TPRED n] component at the other inputs. This property is used to obtain the structure of Fig 4.2.

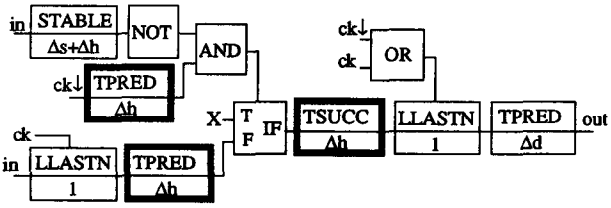


Fig. 4.2: structure obtained by moving the TSUCC component toward the outputs

The rule given as our second example in section 4.2 is applied to obtain the structure given in Fig. 4.3.

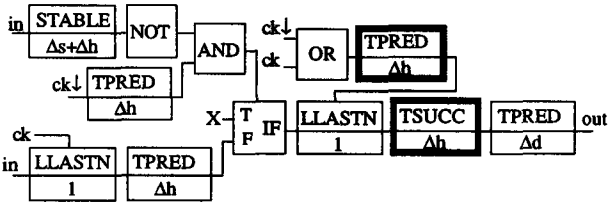


Fig. 4.3: structure after applying example term rewriting rule

In a final step we use the following rule (assuming Δd≤Δh) to remove the TSUCC component:

$$n \leq m \Rightarrow (\,\mathrm{TPRED}\ m\ (\,\mathrm{TSUCC}\ n\ *\,)\,) = (\,\mathrm{TPRED}\ (\,m\text{-}n\,)\ *\,)\,.$$

5. Results and conclusions

We developed and implemented (and are currently developing) tools for hardware verification based on term rewriting techniques. Most of the methods concerning the verification of complex combinational circuits are implemented in the VERENA system. Descriptions of synchronous sequential circuits with corresponding state variables can be verified automatically too. We are currently implementing the procedure to obtain verfication goals that can be handled by the term comparison methods already implemented.

To prove the correctness of term rewriting rules we had to choose a formalism both covering the low level description of hardware behavior based on time functions and the descriptions based on VIOLA providing the term rewriting rules.

The HOL proof assistant was used to specify hardware behavior based on type theory. Term rewriting rules are transformed into HOL-terms that are proved to be theorems of the theory defining the framework for specifications of hardware behavior. The use of the HOL prove assistant was very helpful in the detection of inaccuracies in hand-made proofs and it much fastens the development of new term rewriting rules concerning sequential behavior.

Currently, we are thinking about additional applications of the HOL88 system related to the development of our verification tools. For example, we want to use HOL to prove that the verification of the subgoals obtained from a sequential description is sufficient to solve the whole verification goal. The proofs are up to now done by hand.

Acknowledgements

I would like to thank Phil Windley and Sara Kalvala who sent me addresses of FTP sites to get a version of the HOL88 proof assistant for Sun 3.

References

[AmCH 86] P.Amblard, P.Caspi, N.Halbwachs: "Use of time functions to describe and explain circuit behavior," *IEE Proceedings*, Vol.133, Pt.E, No.5, 1986, pp. 271-275

[BoMo 79] R.Boyer, J.Moore: "A Computational Logic," Academic Press, New York, 1979

[BoPP 88] D.Borrione, J.-L.Paillet, L.Pierre: "Formal Verification of CASCADE descriptions," *The Fusion of Hardware Design and Verification*, G.J.Milne (ed.), Elsevier Science Publishers B.V. (North-Holland), 1988, pp. 185-210

[BrCa 89] A.Bronstein, C.L.Talcott: "Formal Verification of Pipelines based on String-Functional Semantics," *Proc. of the IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, L.J.M.Claesen (ed.), Houlthalen Belgium, Nov., 1989, pp. 347-364

[BrCDM 85] M.Browne, E.Clarke, D.Dill, B.Mishra: "Automatic Verification of Sequential Circuits using Temporal Logic," *CHDLs and their Applications*, C.J.Koomen and T.Moto-oka (eds.), North-Holland, 1985, S.98-113

[Eve 86] A.Eveking: "Formal Verification of Synchronous Systems," *Formal Aspects of VLSI Design*, G.J.Milne and P.A.Subrahmanyam (eds.), Elsevier Science Publishers B.V. (North-Holland), 1986, pp. 137-151

[FuKTM 86] M.Fujita, S.Kono, H.Tanaka, T.Moto-oka: "Aid to hierarchical and structural logic design using temporal logic and Prolog," *IEE PROCEEDINGS*, Vol.133, Pt.E, No.5, Sept. 1986, S283-294

[Gor 85] M.Gordon: "HOL - A Machine Oriented Formulation of Higher Order Logic," University of Cambridge, Computer Laboratory, Technical Report no. 68, 1985

[Gra 88] W.Grass: "VERENA - A CAD tool for designing guaranteed correct logic circuits," *Proceedings of the 2nd ABAKUS workshop*, Innsbruck-Igls, Austria, Sept.1988, pp. 41-56

[HaDa 85] F.K.Hanna, N.Daeche: "Specification and Verification using Higher-Order Logic," *Proc. CHDL*, Kommen and Moto-oka (eds.), North Holland, 1985

[HOL 88] M.Gordon: "The HOL Reference Manual," "The HOL Description," and "The HOL Tutorial," *Documentation of the HOL88 System*, Cambridge, 1988

[Lar 88] T.Larsson: "Hardware Verification based on Algebraic Manipulation and Partial Evolution," *The Fusion of Hardware Design and Verification*, G.J.Milne (ed.), Elsevier Science Publishers B.V. (North-Holland), 1988, pp. 231-252

[Mut 91] M.Mutz: "Formal verification of sequential circuits with VERENA: a case study," *Proc. of the Advanced Research Workshop on Correct Hardware Design Methodologies*, Turin, Italy, June 1991, P.Prinetto and P.Camurati (eds.), Elsevier Science Publishers B.V. (North-Holland)

[Pai 87] J.-L.Paillet: "Descriptions and Specifications of Digital Devices," *From HDL to Guaranteed Correct Designs*, IFIP, pp. 21-42,

[Pie 89] L.Pierre: "The Formal Proof of the Min-Max sequential benchmark described in CASCADE using the Boyer-Moore Theorem Prover," *Proc. of the IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, L.J.M.Claesen (ed.), Houlthalen Belgium, Nov., 1989, pp. 129-148

[RaC 89] S.R.Ramirez Chavez: "Formal proof of the cascading properties of a parallel sorting circuit," *Proc. of the IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, L.J.M.Claesen (ed.), Houlthalen Belgium, Nov, 1989, 338-346

[Zei 76] B.Zeigler: "Theory of Modelling and Simulation", John Wiley & Sons, New York, Chichester, Brisbane, Toronto, 1976