Automating Most Parts of Hardware Proofs in HOL

Klaus Schneider, Ramayya Kumar and Thomas Kropf
University of Karlsruhe, Institute of Computer Design and Fault Tolerance (Prof. D. Schmid)
P.O. Box 6980, 7500 Karlsruhe, Germany

1 INTRODUCTION

In safety critical applications it is mandatory to fabricate chips which are design error free. With the increasing complexity of designs this goal is hard to satisfy without methods specially dedicated to this task. Hence formal verification methods are gaining more and more importance.

To verify a today's ASIC, containing some 100,000 transistors, methods are needed which are capable of managing hierarchical and modular designs, as well as large and complex proof tasks. Moreover, it turns out that the underlying formalism must be powerful enough to allow natural descriptions which closely reflect the informal specification [1], [2].

Successful approaches in this regard are mostly based on higher-order logic [3], [4]. This formalism is ideally suited to compactly describe circuits, where input and output signals are represented as functions of time. In addition, it is easily possible to use parameterized modules, which are recursively definable, e.g. n-bit regular structures like adders and registers [5], [1]. However, higher-order logic is undecidable and automated theorem proving tools are not available. Hence, most of these approaches are based on interactive proof assistants like HOL, which grounds on natural deduction [1]. It provides a set of inference rules and theorems which may be combined by user-definable tactics to automate small portions of the proving process. Based on these approaches, parts of the processor VIPER and the complete TAMARACK have been successfully verified [6], [7], [8].

Although extensive research has been performed on hardware verification, it is still far away from being available to and accepted by normal designers as a standard tool like simulation. This is due to the fact, that up to now full automation is only achieved in the context of finite state verification and propositional temporal logic, both suited to verify only small and medium sized circuits [9], [10] [11]. On the other hand, the interactive approach as described above requires a fundamental knowledge of logic and theorem proving, so that any initial enthusiasm of a typical circuit designer, on hearing about the capabilities of verification, is instantly throttled. Hence the requirements for verification as an adequate design tool are automation (at least as much as possible) and guidance for the remaining interactive verification, so that the designer's creativity may be exercised without a sophisticated knowledge of formal logic.

This paper focuses on possibilities for automation which can be achieved in a twofold way. Our experiences in interactive hardware verification with HOL and LAMBDA [12] have shown, that most proofs follow a specific sequence of steps. This observation can be used to structure the hardware verification process and to find automatic decomposition methods (similar to a manually performed proof) to transform the original goal into smaller pieces. This kind of automation is implemented in MEPHISTO (Managing Exhaustive Proofs of Hardware for Integrated circuit designers by Structuring Theorem proving Operations), elaborated in more detail in the next section.

A large number of subgoals emerge from the decomposition process. The manual proof is cumbersome and takes a lot of time, although most of these subgoals are quite simple to prove since they are first-order-like with only few higher-order constructs. Automating the proof of these subgoals is also possible by integrating an automated theorem proving tool in HOL. It is based on $\mathcal{RSEQ}$, a modified form of the known sequent calculus $\mathcal{SEQ}$. $\mathcal{RSEQ}$ overcomes the inefficiencies of the standard sequent calculus approach. In section 3 $\mathcal{RSEQ}$ is described. The implementation of the prover FAUST (First-order Automation using Unification in a Sequent calculus Technique), which is based on $\mathcal{RSEQ}$ is explained in chapter 4. Experimental results are reported in section 5 and section 6 concludes the paper.

2 STRUCTURE OF HARDWARE PROOFS

In this section a brief overview of the structure of hardware proofs is given. A more elaborate version of MEPHISTO – the hardware oriented proof tool – is found in [13].

A thorough study of various reports on hardware verification [6], [7], [8] as well as our own investigations have shown, that it is possible to structure and classify the steps in interactive hardware verification as follows:

Step 1: Describe the specification and implementation of the circuit to be verified, and set the goal to be proved.

Step 2: Expand the definitions of the specification and the implementation, to obtain formulae at the desired level of abstraction.

Step 3: Simplify the goal into subgoals by applying induction rules and/or domain specific rules, e.g. theorems about n-bit values, natural numbers etc.

After this step several subgoals may be obtained, which are all proved using steps 4 and 5.

Step 4: Simplify each subgoal.

Step 5: Prove all the subgoals.

We illustrate the above-mentioned five steps by means of the parity example used in [14]. An informal specification of the synchronous even parity circuit is as follows:

Initially the output (out) is set to "T" (true). At every n+1th clock, the output is T iff there have been an even number of T's on the input line (in).

A sample formal specification is stated below and figure 1 shows a possible implementation:

$$\forall \text{ in, out. PARITY\_SPEC(in, out)} := \forall \text{ t . ((out } 0 \leftrightarrow \text{T)} \wedge \text{(out (suc t)} \leftrightarrow \text{EVEN (in,out))}$$

where the predicate EVEN is defined as:

$$\forall \text{ in, out. EVEN(in, out)} := \forall \text{ t . (in (suc t)} \leftrightarrow \neg \text{ out t)}$$

The predicate EVEN, encodes the informal specification — *at all time instants, EVEN is true, iff "in t+1" is equivalent to the complement of "out t".*

Step 1:

The specification and implementation of the circuit are defined as predicates at the desired level of abstraction. They correspond to the behavior and the structure of the circuit, respectively and are described in the usual manner using higher-order logic [5]. The implementation can be automatically derived as a conjunction of predicates, each of which corresponds to the specification of some previously verified components. The formal implementation of the parity circuit is given in figure 2.

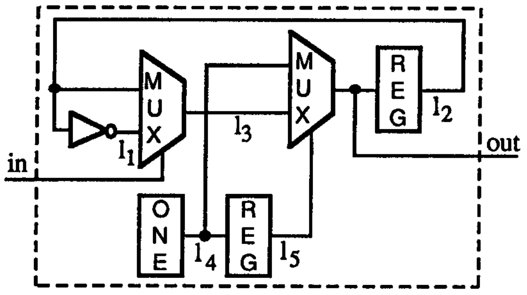

Figure 1: Parity Implementation

$\forall$ in, out . PARITY_IMP(in,out) :=
$\exists\, l_1, l_2, l_3, l_4, l_5.\ \forall\, t\,.$
NOT_SPEC(l_2 t, l_1 t) $\wedge$
MUX_SPEC(in t, l_1 t, l_2 t, l_3 t) $\wedge$
REG_SPEC(out, l_2) $\wedge$
ONE_SPEC(l_4 t) $\wedge$
REG_SPEC(l_4, l_5) $\wedge$
MUX_SPEC(l_5 t, l_3 t, l_4 t, out t)

Figure 2: Formal Description

The goal to be verified can now be specified as:

$\forall$ in, out. PARITY_IMP(in, out) $\leftrightarrow$ PARITY_SPEC(in, out)

It is evident from the description of the goal, specification and implementation that a hierarchical verification is being performed. In the implementation specified above, a library containing the behavioral and structural descriptions of the used gates and their corresponding correctness theorems has been used (Table 1).

Table 1: Formal specifications of the library components

Component	Definition
NOT_SPEC (in,out)	$\forall$ in,out. (out $\leftrightarrow \neg$ in)
ONE_SPEC (out)	$\forall$ out. (out $\leftrightarrow$ T)
MUX_SPEC (sel,in_1,in_2,out)	$\forall$ sel, in_1, in_2, out. (out $\leftrightarrow$ ((sel $\rightarrow$ in_1) $\wedge$ ($\neg$sel $\rightarrow$ in_2)))
REG_SPEC (in,out)	$\forall$ in, out. ($\forall$ t. ((out 0 $\leftrightarrow$ F) $\wedge$ (out (suc t) $\leftrightarrow$ in t)))

Step 2:

The specification and the implementation are now expanded using the definitions in Table 1. The datatypes used, are also refined into their functional relationships at the next level of abstraction, e.g. natural numbers to bit-vectors. Applying this step on the parity example generates the formula:

$$
\begin{array}{l}
\forall \text{ in, out} . \\
\quad \exists\, l_1, l_2, l_3, l_4, l_5 .\ \forall\, t . \ (l_1\, t \leftrightarrow \neg\, l_2\, t) \wedge \\
\qquad\qquad (l_3\, t \leftrightarrow ((\text{in } t \rightarrow l_1\, t) \wedge (\neg \text{ in } t \rightarrow l_2\, t))) \wedge \\
\qquad\qquad (\forall\, t_1 .\ (l_2\, 0 \leftrightarrow F) \wedge \\
\qquad\qquad\qquad (l_2\, (\text{suc } t_1) \leftrightarrow \text{out } t_1)) \wedge \\
\qquad\qquad (l_4\, t \leftrightarrow T) \wedge \\
\qquad\qquad (\forall\, t_2 .\ (l_5\, 0 \leftrightarrow F) \wedge \\
\qquad\qquad\qquad (l_5\, (\text{suc } t_2) \leftrightarrow l_4\, t_2)) \wedge \\
\qquad\qquad (\text{out } t \leftrightarrow ((l_5\, t \rightarrow l_3\, t) \wedge (\neg\, l_5\, t \rightarrow l_4\, t))) \\
\leftrightarrow \\
\forall\, t . \ ((\text{out } 0 \leftrightarrow T) \wedge \\
\qquad (\text{out (suc } t) \leftrightarrow (\text{in (suc } t) \leftrightarrow \neg \text{ out } t))
\end{array}
$$

Step 3:

This is the creative step, where the user has to use his knowledge in breaking up the goal into subgoals, apply induction and the lemmas needed. Many design specific heuristics can be built in to aid the user here. However, due to the very nature of the problem, automating this step is often impossible. In the simple parity example, this step can be skipped.

Step 4:

Having broken up the original goal into subgoals, the subgoals can then be automatically simplified. This is performed by eliminating internal lines. This step, also called the unwind step, is performed by first converting the formula into a prenex form [15]. Afterwards, different rewrite rules are applied to eliminate internal lines, e.g. by replacing an output of a combinational element in terms of its inputs. Then further logical simplifications are performed.

Applying the unwind step to the parity example results in the following description -

$$
\begin{array}{l}
\forall \text{ in, out} . \\
\quad \forall\, t . \ ((\text{out } 0 \leftrightarrow T) \wedge \\
\qquad (\text{out (suc } t) \leftrightarrow ((\text{in (suc } t) \rightarrow \neg \text{ out } t) \wedge (\neg \text{ in (suc } t) \rightarrow \text{out } t)))) \\
\leftrightarrow \\
\quad \forall\, t . \ ((\text{out } 0 \leftrightarrow T) \wedge \\
\qquad (\text{out (suc } t) \leftrightarrow (\text{in (suc } t) \leftrightarrow \neg \text{ out } t)))
\end{array}
$$

Step 5:

Our automated prover FAUST can now be used to prove automatically each of the subgoals generated in step 4.

3 THE THEORY UNDERLYING FAUST

FAUST is based on a modified form of sequent calculus ($\mathcal{SEQ}$) called "Restricted Sequent Calculus" or $\mathcal{RSEQ}$, which lends itself to efficient implementation. We shall first give a brief description of $\mathcal{SEQ}$ and a few basic definitions before the reasons for its inefficiency are outlined.

3.1 Sequent Calculus

Definition 3.1: A *sequent* is a pair (Γ, Δ) of finite (possibly empty) sets of formulae $\Gamma := \{\phi_1, \ldots, \phi_m\}$, $\Delta := \{\psi_1, \ldots, \psi_n\}$. The pair (Γ,Δ) will be henceforth written as "$\Gamma \vdash \Delta$". Γ is called the *antecedent* and Δ is called the *succedent*.

Detailed semantics of sequents can be found in various textbooks on logic [16, 15] and are omitted here. Intuitively, a sequent is valid, if the formula $(\phi_1 \wedge \ldots \wedge \phi_m) \rightarrow (\psi_1 \vee \ldots \vee \psi_n)$ is valid.

The calculus based on such sequents contains several rules which reflect the semantics of the various operators (including quantifiers), and a single axiom or rather an axiom scheme which is a sequent "$\Gamma \vdash \Delta$", such that, Γ and Δ contain some common proposition ($\Gamma \cap \Delta \neq \{\}$). An informal semantic for the axiom scheme corresponds to the sequent "$\Phi \vdash \Phi$". Proving the correctness of any statement within $\mathcal{SEQ}$ then corresponds to iterative rule applications which decompose the original sequent into simpler sequents, so that finally axioms are obtained. This process can be visualized as a proof tree $\mathcal{P}$ (Fig. 3) and a closed proof tree is one which has an axiom at each leaf node.

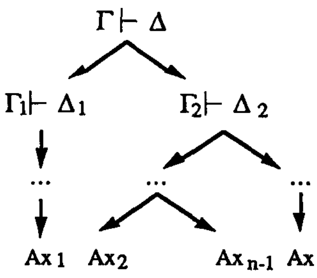

Figure 3: A closed proof tree

The rules can be classified into four types — α, β, δ and γ (cf. section 3.3). The former three rules are uncritical as they can be applied deterministically i.e. each application simplifies the sequent by eliminating the operator or quantifier. This implies that these rules can be applied only once on each operator (quantifier). The γ-rule on the other hand can be applied indefinitely (does not eliminate the quantifier) and the choice of the best term used for the quantifier substitution is unknown at the time of rule application. This choice greatly influences the depth of the proof tree. It is this rule which poses a major hurdle in the efficient implementation of $\mathcal{SEQ}$. The problems with critical rule application also appears in the implementation of tableaux-based, first-order provers like HARP [17] which overcome these problems by using good heuristics in guessing the right term for substitution. We on the other hand, use an exact approach which plugs in a place-holder called a *metavariable* (not a part of the universe of terms) during the γ-rule application, and thereby postpone the choice of the exact term to a later appropriate time. When the proof tree construction process is ripened, we then use first-order unification for computing the terms that instantiate the introduced metavariable. This concept can be thought of as being similar to lazy evaluation within functional language implementations. The introduction of the metavariable and its consequences are the subject of the next sub-section.

3.2 Modifications to $\mathcal{SEQ}$

The introduction of metavariables during the γ-rule application introduces problems as far as the δ-rules are concerned. An application of the δ-rule requires that the variable that is substituted for the quantified variable is new [15], i.e. it does not appear in the quantified formula. Since the choice of terms for the metavariables appearing in the formula is unknown at this point of time, we have to introduce *restrictions* on the terms that the metavariables can take, so that the terms to be computed in future do not contain the currently introduced constant. The use of such restrictions led us to christen our calculus as $\mathcal{RSEQ}$ or "restricted sequent calculus".

In the definitions to follow, the following notations are used:

<u>Notations</u>:

$\mathcal{F}$	set of all first-order formulae	$\mathcal{T}$	the set of all first-order terms
$\mathcal{V}$	the set of all variables	$\mathcal{V}_M$	the set of all metavariables
$[\,]_x^t$	substitution of a variable x by the term $t \in \mathcal{T}$		

<u>Definition 3.2</u>: A *forbidden set* $fs_m \subseteq \mathcal{T}$ is defined for each metavariable $m \in \mathcal{V}_M$, such that fs_m contains all the variables introduced by δ-rule applications after the introduction of the metavariable m.

<u>Definition 3.3</u>: A *restricted sequent* is a modified sequent which has the form $\Gamma \vdash \Delta \parallel \mathcal{R}$, where $\Gamma, \Delta \subseteq \mathcal{F}$; $\mathcal{R} \subseteq \mathcal{V}_M \times 2^{\mathcal{V}}$, i.e. $(m, fs_m) \in \mathcal{R}$, and "$\parallel$" binds the restriction to the sequent.

<u>Definition 3.4</u>: A substitution σ applied on a restricted sequent is defined as

$\sigma(\Gamma \vdash \Delta \parallel \mathcal{R}) := \sigma(\Gamma) \vdash \sigma(\Delta) \parallel \mathcal{R}$

<u>Definition 3.5</u>: An *allowed substitution* σ of a restricted sequent is a substitution such that, for each $(m, fs_m) \in \mathcal{R}$, the terms occurring in σ do not contain the forbidden variables or in other words: $\forall \tau \in fs_m$. τ does not occur in $\sigma(m)$ for each (m, fs_m)

<u>Definition 3.6</u>: An allowed substitution is said to *close* a restricted sequent if

$\sigma(\Gamma) \cap \sigma(\Delta) \neq \{\}$.

The substitution σ (metaunifier) can be found by modifying the normal Robinson's first-order unification algorithm [18] in such a manner that only metavariables are considered as substitutable sub-terms. This leads to the concept of metaunification where metaunifiers are found. Given $\Gamma = \{\phi_1, ..., \phi_n\}$ and $\Delta = \{\psi_1, ..., \psi_m\}$, metaunifiers σ_{ij} can then be computed for each pair (ϕ_i, ψ_j), if ϕ_i and ψ_j are unifiable. The most general unifiers that are useful are allowed substitutions which do not violate the restrictions. The remaining unifiers are removed from the set of computed unifiers. Each of these substitutions are candidates for closing the restricted sequent. It is additionally possible to refine these substitutions by composing them with additional substitutions η. The compound substitution $\sigma\eta$ continues to unify the pair (ϕ_i, ψ_j) since σ is more general than η. It is also possible that choosing an appropriate refinement results in the closure of further sequents in the overall proof tree. Such closed sequents are all valid in $\mathcal{SEQ}$ as they correspond to axioms by definition.

3.3 Rules of $\mathcal{RSEQ}$

In the rules given below, both the variable y and the metavariable m are new, i.e. they do not appear in the sequent until this point of time. The function $\rho_y(\mathcal{R})$ used for updating the restrictions of the existing metavariables is defined recursively as follows:

$$\rho_y(\mathcal{R}) := \begin{cases} \{\}, \text{ if } \mathcal{R} = \{\} \\ \{(m, \{y\} \cup fs_m)\} \cup \rho_y(\mathcal{R}_1), \text{ if } \mathcal{R} = \{(m, fs_m)\} \cup \mathcal{R}_1 \end{cases}$$

Given that Γ, Δ are all sets of formulae, ϕ and ψ are formulae, m is a metavariable and x and y are variables, the following are the rules of $\mathcal{RSEQ}$. We use the notation ϕ,Γ instead of $\{\phi\}\cup\Gamma$.

NOT_LEFT
$$\frac{\neg\phi,\Gamma\vdash\Delta\|\mathcal{R}}{\Gamma\vdash\phi,\Delta\|\mathcal{R}}$$

NOT_RIGHT
$$\frac{\Gamma\vdash\neg\phi,\Delta\|\mathcal{R}}{\phi,\Gamma\vdash\Delta\|\mathcal{R}}$$

AND_LEFT
$$\frac{\phi\wedge\psi,\Gamma\vdash\Delta\|\mathcal{R}}{\phi,\psi,\Gamma\vdash\Delta\|\mathcal{R}}$$

AND_RIGHT
$$\frac{\Gamma\vdash\phi\wedge\psi,\Delta\|\mathcal{R}}{\Gamma\vdash\phi,\Delta\|\mathcal{R}\quad\Gamma\vdash\psi,\Delta\|\mathcal{R}}$$

OR_LEFT
$$\frac{\phi\vee\psi,\Gamma\vdash\Delta\|\mathcal{R}}{\phi,\Gamma\vdash\Delta\|\mathcal{R}\quad\psi,\Gamma\vdash\Delta\|\mathcal{R}}$$

OR_RIGHT
$$\frac{\Gamma\vdash\phi\vee\psi,\Delta\|\mathcal{R}}{\Gamma\vdash\phi,\psi,\Delta\|\mathcal{R}}$$

IMP_LEFT
$$\frac{\phi\rightarrow\psi,\Gamma\vdash\Delta\|\mathcal{R}}{\Gamma\vdash\phi,\Delta\|\mathcal{R}\quad\psi,\Gamma\vdash\Delta\|\mathcal{R}}$$

IMP_RIGHT
$$\frac{\Gamma\vdash\phi\rightarrow\psi,\Delta\|\mathcal{R}}{\phi,\Gamma\vdash\psi,\Delta\|\mathcal{R}}$$

EQUIV_LEFT
$$\frac{\phi\leftrightarrow\psi,\Gamma\vdash\Delta\|\mathcal{R}}{\Gamma\vdash\phi,\Delta\|\mathcal{R}\quad\psi,\Gamma\vdash\Delta\|\mathcal{R}}$$

EQUIV_RIGHT
$$\frac{\Gamma\vdash\phi\leftrightarrow\psi,\Delta\|\mathcal{R}}{\phi,\Gamma\vdash\psi,\Delta\|\mathcal{R}\quad\psi,\Gamma\vdash\phi,\Delta\|\mathcal{R}}$$

ALL_LEFT
$$\frac{\forall x.\phi,\Gamma\vdash\Delta\|\mathcal{R}}{[\phi]^m_x,\forall x.\phi,\Gamma\vdash\Delta\|\mathcal{R}\cup\{(m,\{\})\}}$$

ALL_RIGHT
$$\frac{\Gamma\vdash\forall x.\phi,\Delta\|\mathcal{R}}{\Gamma\vdash[\phi]^y_x,\Delta\|p_y(\mathcal{R})}$$

EXISTS_LEFT
$$\frac{\exists x.\phi,\Gamma\vdash\Delta\|\mathcal{R}}{[\phi]^y_x,\Gamma\vdash\Delta\|p_y(\mathcal{R})}$$

EXISTS_RIGHT
$$\frac{\Gamma\vdash\exists x.\phi,\Delta\|\mathcal{R}}{\Gamma\vdash\exists x.\phi,[\phi]^m_x,\Delta\|\mathcal{R}\cup\{(m,\{\})\}}$$

These rules can be classified into four types as stated earlier.

α-rule	NOT_LEFT, NOT_RIGHT, AND_LEFT, OR_RIGHT, IMP_RIGHT
β-rule	AND_RIGHT, OR_LEFT, IMP_LEFT, EQUIV_LEFT, EQUIV_RIGHT
δ-rule	ALL_RIGHT, EXISTS_LEFT
γ-rule	ALL_LEFT, EXISTS_RIGHT

In constructing proof trees the rules themselves are not as important as the types. Applying the different types of rules yields the following sequents:

$$\frac{\alpha}{\alpha_1}\qquad\frac{\beta}{\beta_1\quad\beta_2}\qquad\frac{\delta}{\delta(y)}\qquad\frac{\gamma}{\gamma(m)}$$

Starting from the original sequent and recursively applying the rules the proof tree can be derived. An example illustrating the application of the rules is given below. It is to be noted that the formula appearing in the sequent obtained after the first δ-rule application "$\exists y\,\forall z\,.\,P\,c_1\,z\rightarrow P\,c_1\,y$", is abbreviated as Φ :

$\vdash\ \forall x\,\exists y\,\forall z.P\,x z\rightarrow P\,x y\ \|\{\}$

$\downarrow\delta$ (ALL_RIGHT)

$\vdash\ \exists y\,\forall z.P\,c_1 z\rightarrow P\,c_1 y\ \|\{\}$

$\downarrow\gamma$ (EXISTS_RIGHT)

$\vdash\ \Phi,\forall z.P\,c_1 z\rightarrow P\,c_1 m_1\ \|\{(m_1,\{\})\}$

$\downarrow\delta$ (ALL_RIGHT)

$\vdash\ \Phi,\ P\,c_1 c_2\rightarrow P\,c_1 m_1\|\{(m_1,\{c_2\})\}$

$\downarrow\alpha$ (IMP_RIGHT)

$P\,c_1 c_2\vdash\ \Phi,P\,c_1 m_1\|\{(m_1,\{c_2\})\}$

$\downarrow$ UNIFY ($m_1\leftarrow c_2$) possible, but forbidden

$P\,c_1 c_2\vdash\ \Phi,P\,c_1 m_1\|\{(m_1,\{c_2\})\}$

$\downarrow\gamma$ (EXISTS_RIGHT)

$P\,c_1 c_2\vdash\Phi,P\,c_1 m_1,\forall z,P c_1 z\rightarrow P\,c_1 m_2$
$\|\{(m_1,\{c_2\}),(m_2,\{\})\}$

$\downarrow\delta$ (ALL_RIGHT)

$P\,c_1 c_2\vdash\Phi,P\,c_1 m_1,P\,c_1 c_3\rightarrow P\,c_1 m_2$
$\|\{(m_1,\{c_2,c_3\}),(m_2,\{c_3\})\}$

$\downarrow\alpha$ (IMP_RIGHT)

$P\,c_1 c_2,P\,c_1 c_3\vdash\Phi,P\,c_1 m_1,P\,c_1 m_2$
$\|\{(m_1,\{c_2,c_3\}),(m_2,\{c_3\})\}$

$\downarrow$ UNIFY ($m_2\leftarrow c_2$) possible

$P\,c_1 c_2,P\,c_1 c_3\vdash\Phi,P\,c_1 m_1,P\,c_1 c_2$
$\|\{(m_1,\{c_2,c_3\}),(m_2,\{c_3\})\}$

closed

The soundness and completeness proofs of $\mathcal{RSEQ}$ are given in [19] and [20].

4. IMPLEMENTATION OF $\mathcal{RSEQ}$ IN FAUST

An efficient implementation of $\mathcal{RSEQ}$ requires the clarification of certain concepts which are briefly given in this section.

4.1 Fairness of the rule application

In the course of the proof tree construction, it is possible that many different types of rules can be applied on the sequent, at any given time. A random application of the rules is dangerous as it could lead to an infinite growth of the proof tree. A trivial example of this would be to apply the γ-rule over and over again. Avoiding such pitfalls without the use of heuristics is achieved by giving an order of precedence for the rules – $\alpha >> \delta >> \beta >> \gamma$.

Definition 4.1: An application of the rule is defined to be *fair* if no rule gets a continuing precedence over the others.

The uncritical rules (α, δ, β), can be applied only a finite number of times and hence they are fair among themselves. The γ-rules on the other hand, can be applied infinitely. Due to definition of the rule precedence, a γ-rule can be applied only when the uncritical rules are not applicable. Now it only remains to ascertain that the γ-rules are fair among themselves. This is achieved by introducing a queue local to each sequent containing the formulae belonging to the sequent, on which γ-rules have been applied. When a γ-rule is applied, the formula on which this rule has been applied is deleted from it and added to the end of the queue. This ensures the fairness among the γ-rules, as further γ-rule applications are done on quantified variables which have not been instantiated so far. If no further γ-rules can be applied and the sequent cannot be closed, then further γ-rule applications are done on the formulae stored in the queue, local to the sequent. A fair application of the rules on a valid first-order statement will always terminate and the proof of this statement is given in [19].

4.2 Depth-first construction of the Proof Tree

The unification algorithm produces the most general metaunifier σ of two formulae, i.e. σ satisfies the sufficiency conditions for being a unifier. Given that η is any substitution, the composition $\sigma\eta$ (also written as $\eta \bullet \sigma$) is still a unifier for the two original formulae, however no more the most general. This observation indicates that the substitutions needed for closing the proof-tree can be computed along with the construction of the proof-tree itself. A depth-first construction of the proof-tree incorporating the above-mentioned strategy is as follows:

1. The proof-tree $\mathcal{P}_0$ is initialized to $\Gamma \vdash \Delta \parallel \{\}$ and the substitution set Σ_0 to $\{id\}$, which is the identity substitution.
2. Given the proof-tree $\mathcal{P}_n$ after n rule applications and the substitution set Σ_n, we proceed with the left most node $\mathcal{S}$ which is not yet closed, in the following manner:
   (a) If an α rule is applicable, the path leading to $\mathcal{S}$ is extended by α_1 to generate $\mathcal{P}_{n+1}$ and $\Sigma_{n+1} := \Sigma_n$.
   (b) If a δ rule is applicable and no α rule is applicable, the path leading to $\mathcal{S}$ is extended by $\delta(y)$ to generate $\mathcal{P}_{n+1}$ and $\Sigma_{n+1} := \Sigma_n$. The variable y used is any new variable.
   (c) If a β rule is applicable and neither an α rule or a δ rule is applicable, the path leading to $\mathcal{S}$ is extended by two child nodes - β_1 and β_2 to generate $\mathcal{P}_{n+1}$ and $\Sigma_{n+1} := \Sigma_n$.

(d) Given that none of the uncritical rules are applicable but a γ rule is, the path leading to $\mathcal{S}$ is extended by $\gamma(m)$, to generate $\mathcal{P}_{n+1}$ and $\Sigma_{n+1} := \Sigma_n$, where m is a new metavariable. The queue local to the sequent $\mathcal{S}$ is updated as stated in 4.1.

(e) The steps a to d are repeated until they are not applicable any more directly on the sequent.

(f) $\mathcal{S} := \Gamma \vdash \Delta \parallel \mathcal{R}$ now contains only atomic formulae and no more rules can be applied. Given $\Sigma_n = \{\sigma_1,...,\sigma_k\}$, we then try to unify the sequent $\sigma_i(\Gamma) \vdash \sigma_i(\Delta)$ for all i, where $1 \leq i \leq k$. This is achieved by unifying each formula in $\sigma_i(\Gamma)$ with each formula in $\sigma_i(\Delta)$ to obtain the set of unifiers for σ_i, represented as $\Pi_i = \{\pi_1^{(i)},..., \pi_{l_i}^{(i)}\}$. Now there are two possibilities, the first of which being that all Π_is are empty. In this case, the sequent $\mathcal{S}$ cannot be closed at this step and we proceed to step 2(g). On the other hand, even if one of the Π_is are not empty the substitution set Σ_{n+1} is calculated as follows:

$$\Sigma_{n+1} := \{\pi_j^{(i)} \circ \sigma_i : \pi_j^{(i)} \in \Pi_i; \Pi_i \neq \{\}; i = 1,..., k; j = 1,..., l_i\}$$

It is to be noted that each unifier belonging to Σ_{n+1} continues to unify the sequent $\mathcal{S}$. $\mathcal{P}_{n+1}$ is now obtained by declaring the sequent $\mathcal{S}$ as closed and step 2 of the proof construction is continued with the next left most node $\mathcal{S}'$ which is not closed. If all the sequents in $\mathcal{P}_{n+1}$ are closed, a proof of validity has been obtained.

(g) When no substitutions which close the leaf $\mathcal{S}$ are found in step 2(f), then there are two possibilities -

  (i) The queue local to the sequent is empty. In this case the sequent is invalid and construction of the proof-tree is stopped with the message "Invalid Sequent ".

  (ii) If the queue is not empty, a γ rule is applied to the head of the queue local to $\mathcal{S}$. and the proof-tree construction proceeds from step 2(d).

We have also implemented a breadth-first algorithm and algorithms which perform skolemization within FAUST. Although the breadth-first algorithm is much slower than the depth-first algorithm, certain problems which are not solvable using a depth-first approach are provable using the breadth-first prover.

It can be observed that the above-mentioned depth-first algorithm generates a closed proof-tree in a fair manner. Furthermore due to the definition of the precedence rules and the proof of the completeness theorem, all valid sequents can be *theoretically* proved by the breadth-first prover after a finite number of rule applications, although this number may be very large. On the other hand, if the sequent to be proved is invalid then the proof construction process may diverge. Hence a definition of an upper bound on the number of rule applications is desirable, after which the proof construction is terminated with a message - "Goal too complex or invalid sequent".

Interaction with HOL has been achieved by introducing the proofs completed by FAUST as theorems using the "mk_thm" (make theorem) function in HOL. Since this can be dangerous, FAUST also generates a single HOL tactic which can then be used to *automatically* validate the automatic proofs within a normal HOL session [21].

5 EXPERIMENTAL RESULTS

The prover embedded in HOL was first tested for its correctness by using the propositional and first-order formulae in [22] and [23]. The runtimes of the more difficult Pelletier examples are found in Table 2. The ML-code has been incorporated in the public domain version of HOL, which runs on top of Common-Lisp on a SUN 4/65. The problem called Andrew's challenge was solved by generating 86 subgoals as compared to 1600 subgoals generated by resolution provers. Additionally, we have observed that specialized HOL tactics can be developed for difficult problems such as Uruquart's problems, which was then solved in linear time.

Having gained confidence about the correctness of our prover we have looked at some combinational circuits which also required a matter of seconds. At present we have proved the correctness of only small sequential circuits such as parity, serial adder, flipflops, and minmax. Thy did not require any interaction and were proved in a few seconds.

Table 2:Runtimes of Benchmark-Formulae (* indicates times taken by the breadth-first version)

Formula	Time	Formula	Time	Formula	Time	Formula	Time
P_{24}	1.4	P_{36}	1.8	P_{40}	9.9	P_{44}	1.2
P_{26}	1.0	P_{37}	1.8	P_{41}	3.0	P_{45}	6.2
P_{34}	19.1	P_{38}	14.4	P_{43}	- / 147.5*	P_{46}	182.1 / 9.9*

6 CONCLUSIONS AND FUTURE WORK

In this paper it has been shown that most hardware proofs can be broken into easily solvable subgoals by following the sequence of steps given in section 2. The creative steps involved in proving the correctness are few in number and most of the other steps can be automated. This part of the proof process has been implemented in MEPHISTO [13]. Furthermore we have elucidated that, although one needs higher order for specifying hardware, it is a restricted form which can be handled by first-order proving techniques. For this purpose, a modified form of sequent calculus has been proposed. An efficient implementation of the prover FAUST has been presented.We are also working on embedding our approach within the CADENCE framework, so that verification proceeds hand in hand with design.

Even if full automation in the context of complex hardware proofs is not reached with our approach, at least HOL-based verification is freed from a significant part of tedious interactive proof drudgery.

REFERENCES

1 M. J. C. Gordon: Why High-Order Logic is a good Formalism for Specifying and Verifying Hardware; Milne/Subrahmanyam (Eds.), Formal Aspects of VLSI Design, Proc. Edinburgh Workshop on VLSI 1985, North-Holland 1986, pp. 153-178.

2 J. Joyce: More Reasons Why Higher-Order Logic is a Good Formalism for Specifiying and Verifying Hardware; Proc. International Workshop on Formal Methods in VLSI Design, Miami, January 1991.

3 A. Camilleri, M. J. C. Gordon, T. Melham: Hardware Verification using Higher-Order Logic; Borrione (Ed.), Proc. IFIP Workshop on "From H.D.L. Descriptions to Guaranteed Correct Circuit Design", Grenoble 1986, North-Holland, pp.43-67.

4 S. Finn, M. Fourman, M. Francis, B. Harris: Formal System Design - Interactive Synthesis based on Computer Assisted Formal Reasoning; Proc. Intl. Workshop on Applied Formal Methods for Correct VLSI Design, Leuven, Nov. 1989.

5 F.K. Hanna, N. Daeche: Specification and Verification of Digital Systems Using Higher-Order Predicate Logic; IEE Proc. Pt. E, Vol. 133, No. 3, September 1986, pp. 242-254.

6 A. Cohn: Correctness Properties of the Viper Block Model: The Second Level; Current Trends in Hardware Verification and Automated Theorem Proving, Springer Verlag, 1988.

7 W.J. Cullyer: Implementing Safety Critical Systems: The VIPER Microprocessor; VLSI Specification, Verification and Synthesis, Eds. Birwistle G. and Subrahmanyam P.A., Kluwer, 1988.

8 J. Joyce: Formal Verification and Implementation of a Microprocessor; VLSI Specification, Verification and Synthesis, Eds. Birwistle G. and Subrahmanyam P.A., Kluwer, 1988.

9 J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill: Sequential Circuit Verification Using Symbolic Model Checking; Proc. 27th Design Automation Conference (DAC 90), 1990, pp. 46-51.

10 O. Coudert, C. Berthet, J.C. Madre: Verification of Synchronous Sequential Machines Based on Symbolic Execution; Proc. Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, June 1989.

11 T. Kropf, H.-J. Wunderlich: A Common Approach to Hardware Verification and Test Generation Based on Temporal Logic; Proc. International Test Conference (ITC 91), Nashville, 1991.

12 Abstract Hardware Limited: LAMBDA - Logic and Mathematics behind Design Automation; User and Reference Manuals, Version 3.1, 1990.

13 K. Schneider, R. Kumar, T. Kropf: Structuring Hardware Proofs: First Steps towards Automation in a Higher-Order Environment; Proc. VLSI '91, Edinburgh, P.B. Denyer, A. Halaas (Eds.), North-Holland, 1991.

14 M. Gordon: A Proof Generating System for Higher-Order Logic; VLSI Specification, Verification and Synthesis, Eds. Birwistle G. and Subrahmanyam P.A., Kluwer, 1988.

15 M. Fitting: First-Order Logic and Automated Theorem Proving; Springer Verlag, 1990.

16 J.H. Gallier: Logic for Computer Science: Foundations of Automatic Theorem Proving; Harper & Row Computer Science and Technology Series No. 5, Harper & Row Publishers,New York, 1986.

17 Oppacher E., Suen: HARP: A Tableaux-based Theorem Prover, Journal of Automated Reasoning; Vol. 4, 1988, pp.69-100.

18 J.A. Robinson: A Machine-oriented logic based on the resolution principle; Journal of the ACM, Vol.12, pp.23-41, 1965.

19 K. Schneider: Ein Sequenzenkalkül für die Hardware-Verifikation in HOL; Diploma Thesis, Institute of Computer Design and Fault-Tolerance, University of Karlsruhe, 1991.

20 Schneider K., Kumar R., Kropf T.: Technical Report, Dept. of Comp. Sc. Univ. of Karlsruhe, 1991, (to appear).

21 R. Kumar, T. Kropf, K. Schneider: Integrating a First-Order Automatic Prover in the HOL Environment; Proc. 1991 International Tutorial and Workshop on the HOL Theorem Proving System and its Applications, Davis, California, Aug. 1991.

22 D. Kalish, R. Montague: Logic: Techniques of Formal Reasoning; World, Harcourt & Brace, 1964.

23 F.J. Pelletier: Seventy-Five Problems for Testing Automatic Theorem Provers; Journal of Automated Reasoning, Vol.2, pp.191-216, 1986.

24 Proceedings of the Third HOL Users Meeting; Aarhus University, Oct. 1990.