INTEGRATING A FIRST-ORDER AUTOMATIC PROVER IN THE HOL ENVIRONMENT

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

The HOL system is a powerful tool for proving higher-order formulae. However, proofs have to be performed interactively and only little automation using tactics is possible. Even though interaction is desirable to guide major and creative backward proof steps of complex proofs, a deluge of simple sub-goals may evolve which all have to be proven manually in order to accomplish the proof. Although these sub-goals are often simple formulae, their proof has not yet been automated in HOL.

In this paper it is shown how it is possible to automate these tasks by integrating a first-order automated theorem proving tool, called FAUST, into HOL. It is based on an efficient variant of the well-known sequent calculus. In order to maintain the high confidence in HOL-generated proofs, FAUST is able to generate HOL tactics which may be used to post-justify the theorems derived by FAUST in HOL. The underlying calculus of FAUST, the tactic generation, as well as experimental results are presented.

1. INTRODUCTION

Although HOL is a very powerful tool for proving various formulae in higher-order logic, it is primarily interactive. In using HOL for hardware verification, we have found that one very often comes across first-order or simple higher-order formulae whose correctness is easy but tedious to prove. Since our main motivation is to build a hardware verification tool based on HOL, which can be used by normal circuit designers with little knowledge in theorem proving, we were keen on automating as much as possible.

The existing HOL library "taut" and the tactics (TAUT_TAC) within are useful only for proving tautologies or substitutions thereof, and furthermore, they can handle only outer universal quantifiers. For example, a simple statement like, \( \forall x. P \ x \rightarrow \exists x. P \ x \), cannot be solved. This motivated us to implement a full first-order automated prover within HOL, called FAUST (First-order Automation using Unification in a Sequent calculus Technique), which is capable of automatically solving not only first order formulae but also certain restricted second-order formulae, which are essential in the context of hardware verification.

Of the three commonly known techniques namely, resolution, tableaux and sequent calculus for first order automation, we have chosen to use sequent calculus as it is closest to natural deduction [Fitt 90] on which HOL is based. Since the normal sequent calculus does not lend itself to an efficient implementation, we have modified the sequent calculus as shown in section 2. In section 3, we discuss some implementational aspects of FAUST. Section 4, discusses the issues of interfacing FAUST with HOL, and it is followed by some results and conclusions.

2. THE THEORY BENEATH FAUST

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

2.1 Sequent Calculus

\begin{definition}
A sequent is a pair \((\Gamma, \Delta)\) of finite (possibly empty) sets of formulae \( \Gamma := \{ \phi_1, ..., \phi_m \} \), \( \Delta := \{ \psi_1, ..., \psi_n \} \). The pair \((\Gamma, \Delta)\) will be henceforth written as \( \Gamma \vdash \Delta \). \( \Gamma \) is called the antecedent and \( \Delta \) is called the succedent.
\end{definition}

Detailed semantics of sequents can be found in various textbooks on logic [Gall 86, Fitt 90] and are omitted here. Intuitively, a sequent is valid, if the formula \( (\phi_1 \land ... \land \phi_m) \rightarrow (\psi_1 \lor ... \lor \psi_n) \) 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, \( \Gamma \) and \( \Delta \) 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 \( SSEQ \) 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 tree (Fig. 1) and a closed proof tree is one which has axioms at all its leaf nodes.
The rules can be classified into four types – $\alpha$, $\beta$, $\delta$ and $\gamma$ (cf. section 2.3). 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 $\gamma$-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 efficient implementation of $\mathcal{SEQ}$. The problems with critical rule application also appears in the implementation of tableaux-based, first-order provers like HARP [OpSu 88] 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 $\gamma$-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.

2.2 Modifications to $\mathcal{SEQ}$

The introduction of metavariables during the $\gamma$-rule application introduces problems as far as the $\delta$-rules are concerned. An application of the $\delta$-rule requires that the variable that is substituted for the quantified variable is new [Fitt 90], 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:

Notations:

- $\mathcal{F}$ set of all first-order formulae
- $\mathcal{T}$ the set of all first-order terms
- $\forall$ the set of all variables
- $\forall_M$ the set of all metavariables
- $\forall_x$ substitution of a variable $x$ by the term $t \in \mathcal{T}$

Definition 2.2: A forbidden set $\forall_M \subseteq \mathcal{T}$ is defined for each metavariable $m \in \forall_M$, such that $\forall_M$ contains all the variables introduced by $\delta$-rule applications after the introduction of the metavariable $m$.

Definition 2.3: A restricted sequent is a modified sequent which has the form $\Gamma \vdash \Delta \parallel \mathcal{R}$, where $\Gamma, \Delta \in \mathcal{F}$; $\mathcal{R} \subseteq \forall_M \times 2^\forall$, i.e. $(m, \forall_m) \in \mathcal{R}$, and "\parallel" binds the restriction to the sequent.

Definition 2.4: A substitution $\sigma$ applied on a restricted sequent is defined as

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

Definition 2.5: An allowed substitution $\sigma$ of a restricted sequent is a substitution such that, for each $(m, \forall_m) \in \mathcal{R}$, the terms occurring in $\sigma$ do not contain the forbidden variables or in other words:

$$\forall \tau \in \forall_m. \tau \text{ does not occur in } \sigma(m) \text{ for each } (m, \forall_m)$$

Definition 2.6: An allowed substitution is said to close a restricted sequent if $\sigma(\Gamma) \cap \sigma(\Delta) \neq \emptyset$.

The substitution $\sigma$ (metaunifier) can be found by modifying the normal Robinson’s first-order unification algorithm [Robi 65] 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 $\Gamma = \{\phi_1, \ldots, \phi_n\}$ and $\Delta = \{\psi_1, \ldots, \psi_m\}$, metaunifiers $\sigma_{ij}$ can then be computed for each pair $(\phi_i, \psi_j)$ if $\phi_i$ and $\psi_j$ 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 $\eta$. The compound substitution $\sigma \eta$ continues to unify the pair $(\phi_i, \psi_j)$ since $\sigma$ is more general than $\eta$. 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.

2.3 Rules of \( \mathcal{SEQ} \)

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:

\[
\rho_y(\mathcal{R}) := \begin{cases} 
\{ \} & \text{if } \mathcal{R} = \{ \} \\
(((m, (y) \cup \mathcal{F}_m)) \cup \rho_y(\mathcal{R}_1)) & \text{if } \mathcal{R} = ((m, \mathcal{F}_m)) \cup \mathcal{R}_1
\end{cases}
\]

Given that \( \Gamma, \Delta \), are all sets of formulae, \( \phi \) and \( \psi \) are formulae, \( m \) is a metavariable and \( x \) and \( y \) are variables, the following are the rules of \( \mathcal{SEQ} \). We use the notation \( \phi, \Gamma \) instead of \( \{\phi\} \cup \Gamma \).

\begin{align*}
\text{NOT_LEFT} & \quad \text{NOT_RIGHT} \\
\phi, \Gamma \vdash \text{All} & \quad \Gamma \vdash -\phi, \text{All} \\
\end{align*}

\begin{align*}
\text{AND_LEFT} & \quad \text{AND_RIGHT} \\
\phi \land \psi, \Gamma \vdash \text{All} & \quad \Gamma \vdash \phi \land \psi, \text{All} \\
\phi, \psi, \Gamma \vdash \text{All} & \quad \Gamma \vdash \phi, \psi, \text{All} \\
\end{align*}

\begin{align*}
\text{OR_LEFT} & \quad \text{OR_RIGHT} \\
\phi \lor \psi, \Gamma \vdash \text{All} & \quad \Gamma \vdash \phi \lor \psi, \text{All} \\
\phi, \Gamma \vdash \text{All} & \quad \Gamma \vdash \phi, \psi, \text{All} \\
\psi, \Gamma \vdash \text{All} & \quad \Gamma \vdash \psi, \text{All} \\
\end{align*}

\begin{align*}
\text{IMP_LEFT} & \quad \text{IMP_RIGHT} \\
\phi \rightarrow \psi, \Gamma \vdash \text{All} & \quad \Gamma \vdash \phi \rightarrow \psi, \text{All} \\
\Gamma \vdash \phi, \text{All} \land \psi, \Gamma \vdash \text{All} & \quad \phi, \Gamma \vdash \psi, \text{All} \\
\end{align*}

\begin{align*}
\text{EQUIV_LEFT} & \quad \text{EQUIV_RIGHT} \\
\phi \leftrightarrow \psi, \Gamma \vdash \text{All} & \quad \Gamma \vdash \phi \leftrightarrow \psi, \text{All} \\
\phi, \Gamma \vdash \phi, \text{All} \land \psi, \Gamma \vdash \text{All} & \quad \phi, \Gamma \vdash \psi, \text{All} \\
\end{align*}

\begin{align*}
\text{ALL_LEFT} & \quad \text{ALL_RIGHT} \\
\forall x, \phi, \Gamma \vdash \text{All} & \quad \Gamma \vdash \forall x, \phi, \text{All} \\
\mathcal{M} \vdash \forall x, \phi, \Gamma \vdash \text{All} & \quad \Gamma \vdash \mathcal{M} \vdash \forall x, \phi, \text{All} \\
\end{align*}

\begin{align*}
\text{EXISTS_LEFT} & \quad \text{EXISTS_RIGHT} \\
\exists x, \phi, \Gamma \vdash \text{All} & \quad \Gamma \vdash \exists x, \phi, \text{All} \\
\mathcal{M} \vdash \exists x, \phi, \Gamma \vdash \text{All} & \quad \Gamma \vdash \mathcal{M} \vdash \exists x, \phi, \text{All} \\
\end{align*}

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

\begin{align*}
\alpha & : \text{NOT_LEFT, NOT_RIGHT, AND_LEFT, OR_RIGHT, IMP_RIGHT} \\
\beta & : \text{AND_RIGHT, OR_LEFT, IMP_LEFT, IFF_LEFT, IFF_RIGHT} \\
\delta & : \text{ALL_RIGHT, EXS_LEFT} \\
\gamma & : \text{ALL_LEFT, EXS_RIGHT}
\end{align*}

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

\[
\begin{array}{cccccc}
\alpha_1 & \beta_1 & \beta_2 & \delta(y) & \gamma(m)
\end{array}
\]

Starting from the original sequent and recursively applying the rules the proof tree can be derived. For more details about the proof tree construction the reader is referred to [ScKK 91a]. 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 \( \delta \)-rule application \( \exists y \forall z. P c_1 z \rightarrow P c_1 y \) , is abbreviated as \( \Phi \) :

\[
\begin{array}{c}
\vdash \forall x \exists y \forall z. P x z \rightarrow P x y \parallel () \downarrow \delta \text{(ALL_RIGHT)} \\
\vdash \exists y \forall z. P c_1 z \rightarrow P c_1 y \parallel () \downarrow \gamma \text{(EXISTS_RIGHT)} \\
\vdash \Phi, \forall z. P c_1 z \rightarrow P c_1 m_1 \parallel ((m_1, \{\}]) \downarrow \delta \text{(ALL_RIGHT)} \\
\vdash \Phi, P c_1 c_2 \rightarrow P c_1 m_1 \parallel ((m_1, \{c_2\})) \downarrow \alpha \text{(IMP_RIGHT)} \\
\vdash P c_1 c_2 \rightarrow \Phi, P c_1 m_1 \parallel ((m_1, \{c_2\})) \downarrow \gamma \text{(EXISTS_RIGHT)} \\
\vdash P c_1 c_2 \rightarrow \Phi, P c_1 m_1, \forall z. P c_1 z \rightarrow P c_1 m_2 \parallel ((m_1, \{c_2\}), \{m_2, \{\}]) \downarrow \delta \text{(ALL_RIGHT)} \\
\end{array}
\]

The soundness and completeness proofs of \( \mathcal{SEQ} \) are given in [Schn 91] and in the technical report [ScKK 91b].
3. IMPLEMENTATION OF RSEQ IN FAUST

An efficient implementation of RSEQ requires the clarification of certain concepts which are briefly given in this section.

3.1 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 \( \gamma \)-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 3.1: An application of the rule is defined to be \textit{fair} if no rule gets a continuing precedence over the others.

The uncritical rules (\( \alpha, \delta, \beta \)), can be applied only a finite number of times and hence they are fair among themselves. The \( \gamma \)-rules on the other hand, can be applied infinitely. Due to definition of the rule precedence, a \( \gamma \)-rule can be applied only when the uncritical rules are not applicable. Now it only remains to ascertain that the \( \gamma \)-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 \( \gamma \)-rules have been applied. When a \( \gamma \)-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 \( \gamma \)-rules, as further \( \gamma \)-rule applications are done on quantified variables which have not been instantiated so far. If no further \( \gamma \)-rules can be applied and the sequent cannot be closed, then further \( \gamma \)-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 [Schn 91].

3.2 Construction of the Proof Tree

Since the proof construction process generates a tree it is possible to either have a depth-first or a breadth-first construction procedure.

The depth-first procedure continues to apply rules in a fair manner on the left most branch until it can be closed by finding a set of unifiers. These unifiers can then be refined by the unifiers found in the next left-most branch so that the next node is also closed. This process is repeated until all the nodes are closed. The details of the depth-first algorithm are given in [ScKK 91a]. The depth-first procedure is inherently an incomplete procedure and not all valid first-order statements can be proved. The fact that the termination of a fairly constructed proof tree has been proven does not violate the incompleteness of the depth-first approach since the substitutions that close a particular leaf may not eventually lead to the closure of all the other branches. It is possible that if that leaf was not closed at that point of time but extended by further \( \gamma \)-rule applications on it, then the new set of unifiers thus found will ultimately close the entire proof tree. Unfortunately, the depth to which the proof process has to be deepened cannot be exactly determined in advance.

The breadth-first procedure on the other hand applies the rules fairly on all the branches until the leaf sequents contain only atomic formulæ. An unifier, which closes the entire proof tree is then calculated. If such an unifier can be found the proof tree is declared to be closed or else more \( \gamma \)-rules are applied and the process is continued further. This approach, although a complete one, is rather slow and requires enormous space requirements. We shall investigate a technique called “depth-first iterative deepening approach” [Korf 85], which combines the best features of the depth-first and the breadth-first techniques, in our future implementations of the prover.

3.3 Speeding-up the Algorithm and Run-time considerations

If the above-mentioned techniques for depth-first and breadth-first construction are naively implemented, then the number of unifiers to be manipulated grows in an uncontrolled manner. In order to keep the number of unifiers under control and to speed up the process of the proof tree construction, the following enhancements have been made (refer to [ScKK 91b] for details):

- Metavariabes are managed locally within the branches, thus the unifiers corresponding to those metavariables which do not appear in the current branch need not be refined.
- A partial order on the set of unifiers of a leaf are found reflecting the generality of the unifiers. This ordering reduces the number of unifiers that are to be carried over to the next branch.
- The sequent is split-up right at the start, into different sets, each corresponding to the kind of rule type that can be applied on it. This eliminates the search to be performed before the rule application.

Although these improvements speed up the proof process it is possible that the process diverges if the goal to be proven is not valid, since first-order predicate calculus is only semi-decidable. Hence some upper bounds on the time for the proof process can be set which reflects the complexity of the goal. This measure of complexity that
we have chosen is an expression involving some constant factor and the number of operators (quantifiers) in the goal.

4. INTERFACING FAUST WITH HOL

FAUST has been implemented using the ML within HOL. In the beginning, we experimented with the idea of implementing using tactics and the subgoal package of HOL. This however proved to be extremely slow due to the rewrite mechanisms currently implemented in HOL. We therefore resorted to the following two-pass method:
1. Prove the validity of statements outside the formal framework of HOL.
2. Generate a single tactic from the generated closed proof-tree which can then be used from within the subgoal package.

The validity of the statement proved during the first pass can be used within HOL by smuggling it in via the function “mk_thm”. This result could be used by HOL users during the development phase as it is orders of magnitude faster than solving the goal by using the generated tactic, as shown in Table 1.

4.1 Relationship between SEQ and RSEQ

The rules of SEQ and RSEQ differ only as far the δ and γ-rules are concerned. If these rules in SEQ are additionally supplied with parameters corresponding to the exact terms that are to be used for substitution, then the effect of the rule application in SEQ is exactly the same as that in RSEQ. Each rule of RSEQ is then coded as a rule of SEQ (with modifications) within HOL, as tactics for use in conjunction with the subgoal package. These tactics can then be combined into 4 tactics namely, ALPHA_TAC, BETA_TAC, DELTA_TAC and GAMMA_TAC, each reflecting the type of rule (α, β, δ, γ) in RSEQ/SEQ.

4.2 Tactic generation

For tactic generation, the proof construction process is slightly modified. Each node of the proof tree is annotated with the type of rule which has been applied on the node in order to extend the branch. Furthermore, the δ-rule application in SEQ is exactly the same as that in RSEQ. Each rule of RSEQ is then coded as a rule of SEQ (with modifications) within HOL, as tactics for use in conjunction with the subgoal package. These tactics can then be combined into 4 tactics namely, ALPHA_TAC, BETA_TAC, DELTA_TAC and GAMMA_TAC, each reflecting the type of rule (α, β, δ, γ) in RSEQ/SEQ.

Another possibility as suggested by [Arth 91] is to use the closed proof tree in FAUST to perform forward proofs in HOL. The leaves which correspond to axioms can be taken as assumptions and a new set of rules, performing the reverse action, are then used to combine the simpler assumptions into a more complex formula until the root of the proof tree is reached. The newer versions of FAUST will also incorporate this strategy.

5. EXPERIMENTAL RESULTS

The prover embedded in HOL was first tested for its correctness by using the propositional and first-order formulae in [KaMo 64] and [Pel 86]. The runtimes of all the Pelletier examples, except those involving equality, are as shown in Table 1. In Table 1, all the runtimes are the execution time in seconds on a SUN 4/330. The column corresponding to FAUST and HOL signify the runtimes as achieved by FAUST with and without the tactic generation, respectively. The column titled subgoals, corresponds to the number of intermediate theorems generated in HOL. The entries which are marked by an asterix have been proven by the breadth-first prover. It can furthermore be observed that the formal proof within the HOL philosophy using tactic generation, can take as much as 74 times the time taken by the prover which uses “mk_thm”, as seen in the example — P9.
The problem called Andrew's challenge (ref. Table 1, p34) was solved by generating 86 subgoals in FAUST without tactic generation, as compared to 1600 subgoals generated by resolution provers. A sample session in HOL for proving it is given in figure 4.

```ml
#let g = ([|"(!t.
   ~14 t \land
   ~17 t \land
   (i4 t, t) = e t \land (i4 t, v 17 t)) \land
   (i7 t, t) = e t \land ~(i4 t, v 17 t)) \land
   (a t = ~e t \land (i4 t)) |
   ~e(SUC t) \land e(SUC t) \land
   (e t \land (i4 t, v 17 t) \lor e t \land ~e(SUC t)) \land
   (i4 t, v 17 t))
= ~e(SUC t) \land e(SUC t) \land e t
|];

#pred_prove g;;
.l- (\x. P x = (\y. P y))
Run time: 0.4s
Intermediate theorems generated: 1
#let t = mk_tactic g;;
tac = - : tactic
Run time: 0.3s
#set_goal g;;
#e(tac);;
OK..
goal proved
l- (\x. P x = (\y. P y))
Run time: 259.8s
Intermediate theorems generated: 123369
```

Figure 4. Proving Andrew's Challenge with FAUST

As our primary interest in building FAUST arose from hardware verification, we have built a tool called MEPHISTO [ScKK 91c, KuKS 91x], which massages specifications so that they can be fed into FAUST. Although these statements are in second-order, FAUST was able to solve all of them. We have been able to prove many combinational circuits and also some sequential circuits such as parity, serial adder, flipflops, shift registers, twisted ring counter and a sequence detector. A flavor of the ease in which complex hardware statements can be proved by FAUST is shown in figure 5. The example used in the figure is a detector which flags true if it detects a sequence of "1", "1", "0" on its input line "e".

```ml
#let g = ([|"(!t.
   ~14 0 \land
   ~17 0 \land
   (i4 t) = e t \land (i4 t, v 17 t)) \land
   (i7 t) = e t \land ~(i4 t, v 17 t)) \land
   (a t = ~e t \land (i4 t)) |
   ~e(SUC t) \land e(SUC t) \land
   (e t \land (i4 t, v 17 t) \lor e t \land ~(i4 t, v 17 t)) \land
   (i4 t, v 17 t))
= ~e(SUC t) \land e(SUC t) \land e t
|];

#pred_prove g;;
.l- (\x. P x = (\y. P y))
Run time: 0.4s
Intermediate theorems generated: 1
#let t = mk_tactic g;;
tac = - : tactic
Run time: 0.3s
#set_goal g;;
#e(tac);;
OK..
goal proved
l- (\x. P x = (\y. P y))
Run time: 8.6s
Intermediate theorems generated: 5005
```

Figure 5. A sample hardware proof in FAUST

6. CONCLUSIONS

An automatic first-order prover, FAUST, has been implemented which greatly enhances the ease of interaction in HOL. FAUST is not only capable of solving first-order formulae but also certain second-order formulae which often occur in the domain of hardware verification. Presently there are two modes of using FAUST within
HOL. The first being the fast mode which makes use of "mk_thm", and the second which proves the goal by using automatically generated tactics.

Further improvements to FAUST will be undertaken so as to incorporate the "depth-first iterative deepening" strategy and also perform forward proofs which would reduce the time taken for formally proving goals in HOL.

REFERENCES


Table 1

<table>
<thead>
<tr>
<th>Reference</th>
<th>Formula</th>
<th>FAUST</th>
<th>HOL</th>
<th>Subgoals</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0,1</td>
<td>1,1</td>
<td>353</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>0,1</td>
<td>0,3</td>
<td>98</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>0,2</td>
<td>0,7</td>
<td>257</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0,1</td>
<td>0,5</td>
<td>193</td>
<td></td>
</tr>
<tr>
<td>P5</td>
<td>0,2</td>
<td>0,6</td>
<td>340</td>
<td></td>
</tr>
<tr>
<td>P6</td>
<td>0,1</td>
<td>0,5</td>
<td>200</td>
<td></td>
</tr>
<tr>
<td>P7</td>
<td>0,1</td>
<td>0,5</td>
<td>242</td>
<td></td>
</tr>
<tr>
<td>P8</td>
<td>0,1</td>
<td>0,2</td>
<td>78</td>
<td></td>
</tr>
<tr>
<td>P9</td>
<td>0,1</td>
<td>2,5</td>
<td>1069</td>
<td></td>
</tr>
<tr>
<td>P10</td>
<td>0,1</td>
<td>7,4</td>
<td>3344</td>
<td></td>
</tr>
<tr>
<td>P11</td>
<td>0,1</td>
<td>0,2</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>P12</td>
<td>0,2</td>
<td>8,1</td>
<td>3930</td>
<td></td>
</tr>
<tr>
<td>P13</td>
<td>0,2</td>
<td>1,8</td>
<td>674</td>
<td></td>
</tr>
<tr>
<td>P14</td>
<td>0,2</td>
<td>3,3</td>
<td>1541</td>
<td></td>
</tr>
<tr>
<td>P15</td>
<td>0,1</td>
<td>0,4</td>
<td>156</td>
<td></td>
</tr>
<tr>
<td>P16</td>
<td>0,1</td>
<td>0,5</td>
<td>240</td>
<td></td>
</tr>
<tr>
<td>P17</td>
<td>0,1</td>
<td>3</td>
<td>1260</td>
<td></td>
</tr>
<tr>
<td>P18</td>
<td>0,2</td>
<td>1,5</td>
<td>647</td>
<td></td>
</tr>
<tr>
<td>P19</td>
<td>0,4</td>
<td>5,4</td>
<td>2518</td>
<td></td>
</tr>
<tr>
<td>P20</td>
<td>0,5</td>
<td>2,3</td>
<td>824</td>
<td></td>
</tr>
<tr>
<td>P21</td>
<td>0,3</td>
<td>5,4</td>
<td>2382</td>
<td></td>
</tr>
<tr>
<td>P22</td>
<td>0,3</td>
<td>1,1</td>
<td>334</td>
<td></td>
</tr>
<tr>
<td>P23</td>
<td>0,3</td>
<td>1,3</td>
<td>551</td>
<td></td>
</tr>
<tr>
<td>P24</td>
<td>1,3</td>
<td>19,2</td>
<td>9310</td>
<td></td>
</tr>
<tr>
<td>P25</td>
<td>0,5</td>
<td>1,8</td>
<td>461</td>
<td></td>
</tr>
<tr>
<td>P26</td>
<td>0,9</td>
<td>6,2</td>
<td>2410</td>
<td></td>
</tr>
<tr>
<td>P27</td>
<td>0,6</td>
<td>2,5</td>
<td>630</td>
<td></td>
</tr>
<tr>
<td>P28</td>
<td>0,7</td>
<td>3,2</td>
<td>1080</td>
<td></td>
</tr>
<tr>
<td>P29</td>
<td>0,8</td>
<td>3,5</td>
<td>1013</td>
<td></td>
</tr>
<tr>
<td>P30</td>
<td>0,5</td>
<td>2,1</td>
<td>573</td>
<td></td>
</tr>
<tr>
<td>P31</td>
<td>0,5</td>
<td>4,5</td>
<td>2143</td>
<td></td>
</tr>
<tr>
<td>P32</td>
<td>0,6</td>
<td>2,3</td>
<td>672</td>
<td></td>
</tr>
<tr>
<td>P33</td>
<td>0,7</td>
<td>2,7</td>
<td>832</td>
<td></td>
</tr>
<tr>
<td>P34</td>
<td>17,2</td>
<td>259,8</td>
<td>123369</td>
<td></td>
</tr>
<tr>
<td>P35</td>
<td>0,5</td>
<td>4,9</td>
<td>2442</td>
<td></td>
</tr>
<tr>
<td>P36</td>
<td>1,6</td>
<td>7,5</td>
<td>2870</td>
<td></td>
</tr>
<tr>
<td>P37</td>
<td>1,6</td>
<td>28,1</td>
<td>13628</td>
<td></td>
</tr>
<tr>
<td>P38</td>
<td>13,0</td>
<td>85,5</td>
<td>34378</td>
<td></td>
</tr>
<tr>
<td>P39</td>
<td>0,5</td>
<td>1,4</td>
<td>451</td>
<td></td>
</tr>
<tr>
<td>P40</td>
<td>8,9</td>
<td>30,9</td>
<td>11296</td>
<td></td>
</tr>
<tr>
<td>P41</td>
<td>2,7</td>
<td>10,5</td>
<td>3109</td>
<td></td>
</tr>
<tr>
<td>P42</td>
<td>0,8</td>
<td>4,8</td>
<td>2033</td>
<td></td>
</tr>
<tr>
<td>P43</td>
<td>132,8*</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>P44</td>
<td>1,1</td>
<td>6,8</td>
<td>2844</td>
<td></td>
</tr>
<tr>
<td>P45</td>
<td>5,6</td>
<td>36,6</td>
<td>14421</td>
<td></td>
</tr>
<tr>
<td>P46</td>
<td>163,9/9,9*</td>
<td>191,5</td>
<td>1496</td>
<td></td>
</tr>
</tbody>
</table>