Structuring and Automating Hardware Proofs in a Higher-Order Theorem-Proving Environment

Ramayya Kumar, Klaus Schneider and Thomas Kropf

Universität Karlsruhe, Institut für Rechnerentwurf und Fehlertoleranz, P.O. Box 6980, W-7500 Karlsruhe 1, Germany
e-mail: {schneide,kumar,kropf}@ira.uka.de

Abstract. In this article we present a structured approach to formal hardware verification by modelling circuits at the register-transfer level using a restricted form of higher-order logic. This restricted form of higher-order logic is sufficient for obtaining succinct descriptions of hierarchically designed register-transfer circuits. By exploiting the structure of the underlying hardware proofs and limiting the form of descriptions used, we have attained nearly complete automation in proving the equivalences of the specifications and implementations. A hardware-specific tool called MEPHISTO converts the original goal into a set of simpler subgoals, which are then automatically solved by a general-purpose, first-order prover called FAUST. Furthermore, the complete verification framework is being integrated within a commercial VLSI CAD framework.

Keywords: hardware verification, higher-order logic

1 Introduction

The past decade has witnessed the spiralling of interest within the academic community in developing methods for formal hardware verification, i.e., to formally prove that a given implementation satisfies its specification [1, 2, 3]. It is desirable, that the implementation description and the specifications are at different abstraction levels or within different domains (behavioural, structural, or physical) in order to enforce an incremental design philosophy used in designing complex circuits [4, 5]. The problems that accompany formal verification within such a context are [6]:

- the inherent complexity of modeling large systems,
- the complexity of related proofs,
- the difficulty of modeling an object in the physical world, and
- the difficulty of formally representing the intentions of a circuit designer.

Verifying the correctness of circuits within these abstract domains in a routine manner is also infeasible due to the paucity of formal verification tools. The use of conventional simulation-oriented tools is therefore the currently taken approach for validating hardware designs within the industry. However, these approaches approaches however suffer from the following drawbacks:

- the infeasibility of exhaustive simulation,
- the imprecise semantics of the underlying simulation models,
- the lack of an universally acceptable formal basis,
- the need for generating appropriate input stimuli, and
- the burden of going through pages of output data.

The methods used for hardware verification can be loosely classified into *the automatic, propositional logic-based approaches* and *the interactive, predicate logic-based ones.* Both approaches have not, however, established themselves within the hardware designer community, since the former can handle only a limited class of circuits and the latter requires a good understanding of mathematical logic.

The *automatic approaches* are based on variations of propositional logic and model checking of finite state machines [7, 8] and additionally also allow automatic test pattern generation [9]. Since the underlying calculi are decidable, complete automation can be achieved. Nevertheless, the complexity of the runtimes within such formalisms grows exponentially with the size of the problem. This limits the applicability of such approaches to only certain classes of problems, although considerable progress has been achieved in managing problems with large sizes [10]. Another major drawback of this approach is the difficulty in incorporating hierarchy, so as to have specifications at different abstraction levels and to perform verification between them. Such techniques are therefore not sufficient for verification while designing complex chips. The additional drawback of this approach corresponds to the difficulty in specifying using complex data types such as natural numbers, lists, enumerated data types, etc. Furthermore, the expressibility of propositional logic precludes the use of generalized n-bit parametrizable components. We therefore postulate that such approaches would not form the core, but would eventually be embedded within a hardware verification environment for solving specialized tasks, e.g., controller verification.

The *interactive approaches,* on the other hand, conform to the following paradigm: Given the formal descriptions in *predicate logic* of the specification S and the implementation I at different abstraction levels, the goal to be proved is to show the equivalence of the two descriptions, i.e., $I \leftrightarrow S$. This model is more suited to verifying complex chips, since it complies with the incremental refinement process (from higher levels of abstractions to lower levels) used in real-life designs. Furthermore, the hierarchy inherent in complex designs can be exploited to reduce the complexity of verification by limiting the size of the verification goals and identifying the possible erroneous components (see appendix F). This hierarchical verification process is illustrated in figure 1. At any juncture, the specification of a subcircuit is verified against one of its possible implementations, since many implementations can exist for a single specification. Once this has been done, the correctness theorem can be used to replace the implementation of the verified

subcircuit by its compact specification. Thus the implementation is itself composed of an interconnection of specifications at a lower level of hierarchy. This process is continued until base modules are reached, whose specifications are given in a library.

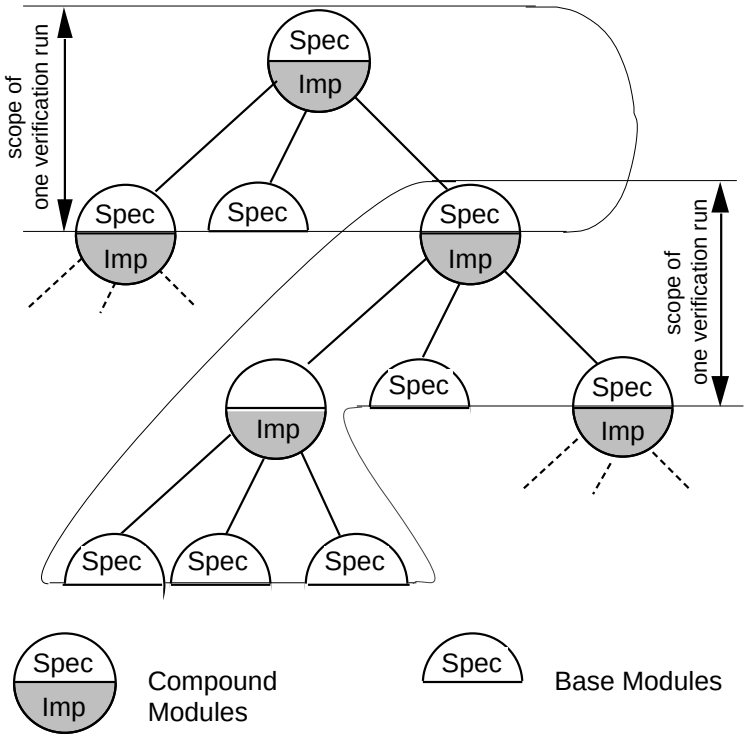


Figure 1: Hierarchical Verification.

If the specification is *partial*, e.g., if only certain safety or liveness properties are to be verified, such an equivalence cannot be shown, and therefore the implication of the specification from the implementation will be proved, i.e., $I \rightarrow S$.

The interactive approaches also embody the following advantages:

- they allow natural specifications that are closer to hardware description languages (HDLs),
- the specifications could use complex data types, such as natural numbers, lists, etc., and
- parametrizable n-bit components can also be specified.

The main weak point of such approaches relate to the lack of *automation* and the unequivocal need for a *good understanding of the logic* and its *associated proof techniques*.

The interactive approaches can be further classified into the *first-order- and higher-order*-based approaches. This classification corresponds to the language used for specification and the intrinsic proof mechanism. The approaches based on the former use either a restricted language (e.g., Prolog [11] or the Boyer-Moore logic [12]) or the complete first-order language as using automated theorem provers like Otter [13]. Although

those methods that use the restricted first-order language have a greater expressibility as compared to propositional logic, they continue to exhibit difficulties in satisfying the above-mentioned advantages, as shown in [14]. Additionally, the use of Otter in verifying hardware has also been very limited due to the underlying proof mechanism.[1]

All these outlined reasons have motivated us to use higher-order logic as the basis for building our hardware verification tools, in spite of its undecidability. We show how this weakness of the logic can be circumvented by restricting ourselves to the hardware domain and to certain specific levels of abstraction, although higher-order logic can be comfortably used to specify complex systems at all levels of abstraction (i.e., system-level down to the switch level). As a first step towards automating proofs of hardware described in higher-order logic, we shall restrict ourselves in this article to the *register-transfer* and *gate-level* abstractions and to *synchronous unit delay* time models.

This article is organized as follows. In the next section, we give a summary of the methods used in a higher-order context. Furthermore, a brief description of the two tools called MEPHISTO and FAUST, which perform the automation of proofs, is also given. In section 3, we elaborate on parts of MEPHISTO, which correspond to the structuring of hardware proofs in the higher-order logic environment. Section 4 deals with the theory behind FAUST. The implementational details of FAUST are given in section 5, which is then followed by some experimental results, conclusions, and an appendix.

2 Hardware verification with higher-order logic

Hanna and Daeche [15] realized that higher order logic is an appropriate language for modeling the behavior and structure of hardware. A surge in the use of this model for hardware verification took place with the coming of the theorem-proving assistant HOL [16], which was an extension of the Edinburgh ML/LCF project [17]. HOL is based on natural deduction and uses a set of five axioms and eight inference rules from which the entire system is built up. The correctness of all the proofs derived with further rules in HOL is guaranteed, since they are translated into a proof that uses exclusively these basic axioms and inference rules. HOL also contains a rich set of theories that comprises definitions, functions, and preproved theorems for facilitating the proof process. Additionally, derived rules can also be specified using the underlying metalanguage ML. The fact that HOL is a public-domain, general-purpose, interactive theorem-proving platform, strengthened by a set of dedicated researchers who constantly strive to bring it to the fore, and furthered by the naturalness of higher-order hardware descriptions [18, 19] led to a profusion of activity in this area [20, 21]. Large processors at different levels of abstraction [22, 23, 24, 25] and a number of medium-sized circuits were formally verified using this approach. Nevertheless, these investigations and other similar attempts using provers like VERITAS [26] and Nuprl [27] are all interactive in nature, and generalized methodologies usable by circuit designers have not yet emerged. Windley's work in providing a structured methodology for verifying microprocessors within HOL is the only work known to us in this regard [28]. A successful attempt at integrating higher-order techniques within a circuit design framework has been reported in [29, 30]. They however use a system called LAMBDA for performing interactive formal synthesis.

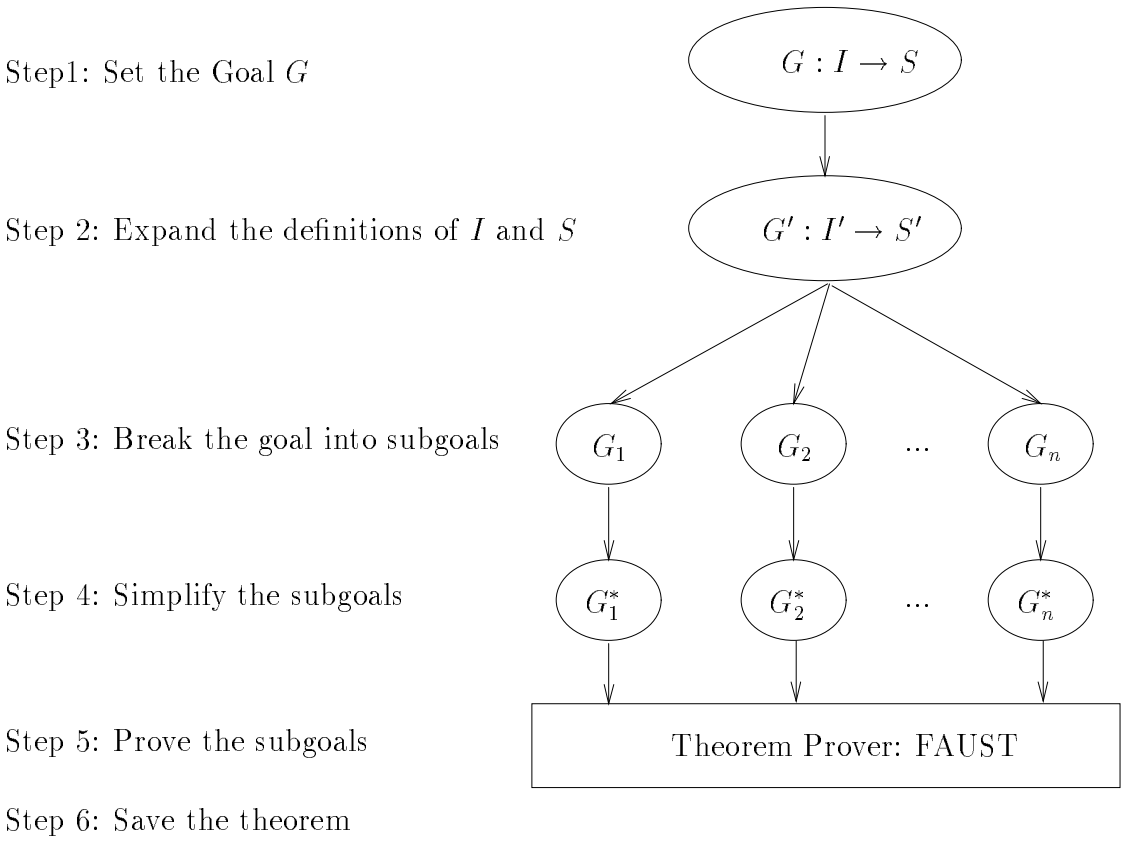


Figure 2: Structure of a hardware proof.

The previous work in hardware verification using HOL relies strongly on the backward proof technique.[2] The goal to be proved, e.g., equivalence of the specification and the implementation, is put on the goal stack. Then the abbreviations (library components) used within the implementation are expanded manually, the the goal is manually split up into smaller pieces, and finally each subgoal is proved manually.[3]

Our studies of various hardware proofs in HOL, hands-on experience with the tool, and additionally restricting the use of higher-order logic to specify hardware at register-transfer level lead us to the following two crucial observations:

- There exists a definite pattern in hardware proofs.
- A restricted higher-order logic (cf. 3.1.1) is sufficient for the description of hardware behaviors and structures.

The first observation steers us towards a methodology for hardware proofs using higher-order logic. By unravelling the structure of hardware proofs in such environments, we have discovered that many steps can be automated and a focussed domain-specific, heuristic-driven succour can be given for the remaining steps. This has been achieved by embedding a hardware specific environment on top of HOL, called MEPHISTO[4] [31, 32, 33]. MEPHISTO systematically converts the original goal into a set of subgoals, whose proofs correspond to the proof of the original one. The sequence of steps in MEPHISTO are summarized in figure 2.

The next section gives the details of all these six steps and also briefly sketches the implementational aspects of MEPHISTO.

The second observation insinuates that since hardware specifications and implementations at the register-transfer level do not require the full expressibility of higher-order logic, first-order techniques can be used to automatically solve most of the subgoals generated by MEPHISTO. We have therefore integrated a first-order prover called FAUST[5] in HOL [31, 34, 35]. FAUST can also be used as a stand-alone prover for solving simpler subgoals from within the goal package of HOL while performing interactive proofs. On the surface, it seems as if MEPHISTO and FAUST have nothing to do with each other. However, MEPHISTO and FAUST operate on the goals in a similar manner, i.e., MEPHISTO splits a goal into subgoals on the "hardware domain", while FAUST does the same on the "logical domain". Some of the subgoals can be solved by MEPHISTO alone, and most of the remaining subgoals are solved by FAUST automatically. Thus the combination of MEPHISTO and FAUST yields a qualitative improvement in the use of such powerful higher-order-based approaches for hardware verification to the extent that complete automation for simpler circuits and significant automation for more complex ones is achieved.

3 The structure of hardware proofs

With the motivation given in the earlier section, the existence of a definite pattern in hardware proofs and the use of restricted higher order logic can be exploited to derive a step-by-step method for hardware proofs in higher-order theorem-proving environments. Many of these steps are also performed during the interactive proof sessions in HOL. The main contribution of this article is to structure these to yield a specific sequence of steps that are always applicable for descriptions at the register-transfer level. In this section, this sequence of steps (figure 2) to be taken in each verification run between one level of hierarchy, as shown in figure 1, is explained. In the following, we assume formal descriptions of a circuit specification S at an abstraction level i and an implementation I at the next lower level $i+1$, both expressed in higher-order logic as described in [18].

3.1 Step 1: Set the goal

This step is a partly interactive step, where the specification, the implementation description, and the goal to be proved are defined. Before examining the specification of the even-parity circuit, we shall formally define the restricted subset of higher-order logic that we use.

3.1.1 Restricted higher order logic

Since all circuits at this level correspond to some form of finite-state machines, the following subset of higher-order logic is sufficient for formalizing specifications and implementations at register-transfer and gate-level:

Definition 3.1 (Hardware-Formulae) *Let $\vec{i}, \vec{o}, \vec{q}, \vec{\ell}$ be tuples of variables of type $\mathbb{N} \to \mathbb{B}$,*[6] *having lengths m, n, k, and l, respectively. Furthermore, let $\vec{\Delta}(p_1, \ldots, p_{m+k+l})$, $\vec{\Omega}(p_1, \ldots, p_{m+k+l})$ and $\vec{\Phi}(p_1, \ldots, p_{m+k+l})$ be propositional formulae with the variables $p_1, \ldots, p_{m+k+l}$ all of type $\mathbb{B}$. Then all formulae $\mathcal{H}(\vec{i}, \vec{o})$ as defined below, are called hardware-formulae:*[7]

$$\begin{aligned}\mathcal{H}(\vec{i}, \vec{o}) := \exists \vec{\ell}.\exists \vec{q}.\forall t.\\ &[\vec{\ell}t \leftrightarrow \vec{\Delta}(\vec{i}t, \vec{\ell}t, \vec{q}t)]\wedge\\ &[(\vec{q}0 \leftrightarrow \vec{\omega}) \wedge (\vec{q}(t+1) \leftrightarrow \vec{\Omega}(\vec{i}t, \vec{\ell}t, \vec{q}t))]\wedge\\ &[\vec{o}t \leftrightarrow \vec{\Phi}(\vec{i}t, \vec{\ell}t, \vec{q}t)]\end{aligned}$$

The variables $IN(\mathcal{H}(\vec{i}, \vec{o})) := \{i_1, \ldots, i_m\}$ are called the input line variables, $OUT(\mathcal{H}(\vec{i}, \vec{o})) := \{o_1, \ldots, o_n\}$ are called the output line variables, $COMB(\mathcal{H}(\vec{i}, \vec{o})) := \{\ell_1, \ldots, \ell_k\}$ are called combinational internal line variables, and $SEQ(\mathcal{H}(\vec{i}, \vec{o})) := \{q_1, \ldots, q_l\}$ are called the sequential internal line variables.

Hardware-formulae have, moreover, to fullfill the following two restrictions:

1. *The* **dependency relation** *of the variables $\ell_i \in COMB(\mathcal{H}(\vec{i}, \vec{o}))$ must be acyclic. The dependency relation is the transitive closure of the relation R, which is defined as*
$$\ell_j \; R \; \ell_i \quad :\Leftrightarrow \ell_j \text{ occurs in } \Delta_i(\vec{i}t, \vec{\ell}t, \vec{q}t)$$
2. *For each variable belonging to $COMB(\mathcal{H}(\vec{i}, \vec{o})) \cup SEQ(\mathcal{H}(\vec{i}, \vec{o})) \cup OUT(\mathcal{H}(\vec{i}, \vec{o}))$, there exists exactly one equation in $\mathcal{H}(\vec{i}, \vec{o})$ with the variable occurring on the left-hand side.*

The sequential internal line variables correspond to internal state transition variables of the finite-state machine, represented by $\mathcal{H}(\vec{i}, \vec{o})$. $\vec{q}0 \leftrightarrow \vec{\omega}$ corresponds to the initialization of these internal state transition variables with the constants $\omega_i \in \mathbb{B}$; $\vec{q}(t+1) \leftrightarrow \vec{\Omega}(\vec{i}t, \vec{\ell}t, \vec{q}t)$ corresponds to the state transition functions; and $\vec{o}t \leftrightarrow \vec{\Phi}(\vec{i}t, \vec{\ell}t, \vec{q}t)$ defines the output functions.

The restrictions given above forbid inconsistencies in hardware. The first restriction prohibits the existence of zero-delay loops and the latter disallows the shortcircuiting of output lines.

This restriction of the higher-order logic, which corresponds to a quantification over unary predicate variables (representing the input/output and internal signals), allows us to achieve almost complete automation of register-transfer level proofs. The use of unrestricted higher-order logic for specifications can be handled by using MEPHISTO as the core and augmenting it with domain-specific approaches such as the generic model for microprocessors [28]. In this article, however, we restrict ourselves to hardware proofs at register-transfer and gate-level abstractions and its details are given in the rest of this section.

3.1.2 Specifications

The informal specification is converted to a formal one by choosing predicates and abstract data types that closely reflect the notions in the informal specification, so that the

validation of the formal specification is easy to perform. It is with respect to this step that higher-order logic has outstanding advantages over the propositional and first-order based approaches [18]. In spite of this fact, the generation of formal specifications is a considerably hard task that should not be underestimated. However, progress is being made in this direction, with the initial work in automatically converting hardware description languages into formal specifications in higher-order logic [36]. The correctness of the specification itself is another issue and will not be considered here.

In the rest of this section, these six steps are elaborated using the parity circuit as an example [16]. 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 (inp).

A possible formal specification of the informally specified 'even parity circuit' is as follows:

$$\text{SPARITY\_SPEC} : \forall\, inp\ out.SPARITY\_SPEC(inp, out) := \\ \forall t.(out\ 0 \leftrightarrow T) \wedge (out(SUC\ t) \leftrightarrow EVEN(inp(SUC\ t), (out\ t)))$$

It can be seen from the above that this specification is easy to visualize once the syntax of the language is understood. This specification is, however, insufficient, since the predicate EVEN has not yet been defined. Pondering over the functionality of EVEN[8] for a while, it is possible to realize that starting with the initial value of *out* as T, and toggling the output at the previous time instant whenever the current input is true, accomplishes the result. Hence the predicate EVEN can be replaced by a predicate TOGGLE, which can be informally specified as follows:

At all time instants, TOGGLE is true iff inp is equivalent to the complement of out.

A formal specification of the predicate TOGGLE can be now defined as follows:

$$\text{TOGGLE\_DEF} : \forall\, inp\ out.TOGGLE(inp, out) \ := \ (inp \leftrightarrow \neg out)$$

3.1.3 Implementations

The implementations, on the other hand, are easier to handle. In MEPHISTO, they can be automatically derived from netlists[9] as a conjunction of predicates, with free predicate variables for input/output lines and existentially quantified predicate variables for internal lines (neither primary inputs nor primary outputs). The predicates used in the conjunction correspond to the specification of previously verified or library components (cf. figure 1). Converting a netlist into such a formula is done by a simple translator that uses the hierarchy in the netlist, recognizing the primary input/output lines and the internal lines, in order to generate a logical description in HOL. This conversion is not formally embedded within HOL. However, certain checks can be made within HOL to ensure the consistency of such descriptions.

The automatically derived formal description of the parity circuit implementation given in figure 3, is as follows:

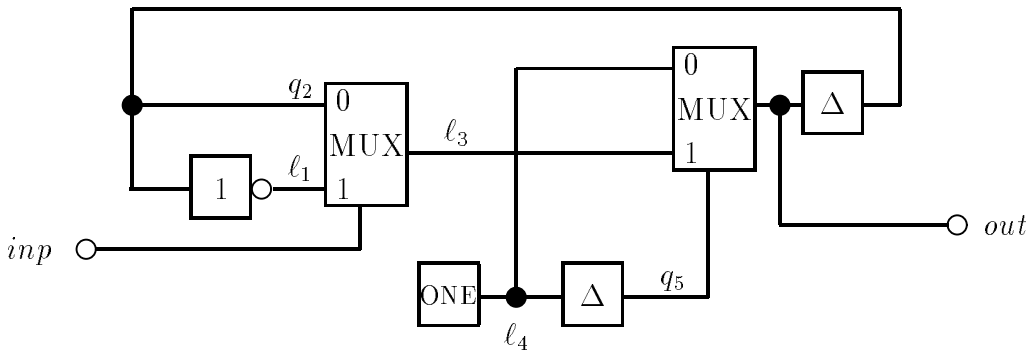


Figure 3: A possible implementation of the even parity circuit.

SPARITY_IMP :

$$\begin{aligned}
&\vdash \forall inp\ out.SPARITY_IMP(inp, out) := \\
&\quad (\exists \ell_1 q_2 \ell_3 \ell_4 q_5.\forall t. \\
&\qquad NOT(q_2\ t, \ell_1\ t) \wedge \\
&\qquad MUX(inp\ t, \ell_1\ t, q_2\ t, \ell_3\ t) \wedge \\
&\qquad DELAY(out, q_2) \wedge \\
&\qquad ONE(\ell_4\ t) \wedge \\
&\qquad DELAY(\ell_4, q_5) \wedge \\
&\qquad MUX(q_5\ t, \ell_3\ t, \ell_4\ t, out\ t))
\end{aligned}$$

The value of a line depends upon the time modeled by natural numbers using the HOL-constants '0' and 'SUC'. Each line variable therefore has the type ': $num \rightarrow bool$'. The implementation description defines a new predicate SPARITY_IMP whose arguments are the input and output signals of the circuit. The value of the internal lines depends on the input variables and are existentially quantified. It has to be noted that the arguments of predicates for combinational circuits are of type ':bool', while the arguments of sequential components have the type ': $num \rightarrow bool$'. For type consistency reasons, the line variables occurring in the predicates are function applications on the time variable to obtain the required type *bool*.

3.1.4 Goal to be proved

The goal to be proved is normally either the equivalence of or the implication between the implementation and the specification. Since a complete specification is available for our parity example, the goal to be proved is as follows:

$$\forall inp\ out.SPARITY\_IMP(inp, out) \ \leftrightarrow \ SPARITY\_SPEC(inp, out)$$

We illustrate the following steps of the verification of the parity example by listing a complete HOL-session,[10] which is characterized by a typewriter font in numbered boxes. We assume that a file 'parity_imp.ml' already exists, which contains the HOL code for the

description of the implementation as given above. We assume further the existence of a theory named 'basics', which contains the specifications for the library components listed in table 1. The description of other components is given in appendix A.

A typical HOL-session for the verification will start as follows:

```
i81s14|HW_EXAMPLES: hol                                                    1


          _  _    __    _      __    __
   |___   |__|   |  |   |     |__|  |__|
   |      |  |   |__|   |__   |__|  |__|

          Version 2.0, built on 14/10/91

#timer true;;
false : bool
Run time: 0.0s

#loadf `~/FAUST/hardprover`;;
.................................() : void
Run time: 3.4s

#new_theory `parity`;;
() : void
Run time: 0.1s
Intermediate theorems generated: 1

#new_parent `basics`;;
Theory basics loaded
() : void
Run time: 0.2s

#let TOGGLE_DEF=new_definition(`TOGGLE_DEF`,
#       "!inp out.TOGGLE(inp,out) = (inp = ~ out)");;
TOGGLE_DEF = |- !inp out. TOGGLE(inp,out) = (inp = ~out)
Run time: 0.3s
Intermediate theorems generated: 2

#let SPARITY_SPEC=new_definition(`SPARITY_SPEC`,
        "!inp out.SPARITY_SPEC(inp,out) =
                !t.(out 0=T) /\ (out(SUC t) =  TOGGLE(inp(SUC t),(out t)))");;
SPARITY_SPEC =
|- !inp out.
    SPARITY_SPEC(inp,out) =
    (!t. (out 0 = T) /\ (out(SUC t) = TOGGLE(inp(SUC t),out t)))
Run time: 0.2s
Intermediate theorems generated: 2

#loadt `~/EXAMPLES/HW_EXAMPLES/parity_imp.ml`;;
```

```
SPARITY_IMP =                                                          2
|- !inp out.
    SPARITY_IMP(inp,out) =
    (?l1 q2 l3 l4 q5.
      !t.
       NOT(q2 t,l1 t) /\
       MUX(inp t,l1 t,q2 t,l3 t) /\
       DELAY(out,q2) /\
       ONE(l4 t) /\
       DELAY(l4,q5) /\
       MUX(q5 t,l3 t,l4 t,out t))
Run time: 0.5s
Intermediate theorems generated: 2


File ~/EXAMPLES/HW_EXAMPLES/parity_imp.ml loaded
() : void
Run time: 1.1s
Intermediate theorems generated: 2

#let goal=[],"!inp out.SPARITY_IMP(inp,out) = SPARITY_SPEC(inp,out)";;
goal =
([], "!inp out. SPARITY_IMP(inp,out) = SPARITY_SPEC(inp,out)")
: (* list # term)
Run time: 0.1s
```

3.2 Step 2: Expand the definitions of I and S

Both specification and implementation usually rely on the usage of predefined predicates and modules to keep the descriptions hierarchical, modular, and understandable. To perform the proof task, these definitions have to be expanded. This step can be completely automated.

Component	Definition
ONE(out)	out
NOT(in,out)	$out \leftrightarrow \neg in$
MUX(s,e_1,e_2,a)	$a \leftrightarrow (s \Rightarrow e_1 \mid e_2)$
DELAY(e,a)	$\forall t.(\neg(a\ 0) \wedge (a(suc\ t) \leftrightarrow e\ t))$

Table 1: Formal specifications of the used library components of the theory 'basics'.

Applying this step on the parity example using the definition of the predicate TOGGLE and the components as specified in table 1, generates the following formula:

```
#let goal1=expand_all goal;;                                                    3
goal1 =
([],
 "!inp out.
   (?l1 q2 l3 l4 q5.
     !t.
      (l1 t = ~q2 t) /\
      (l3 t = (inp t => l1 t | q2 t)) /\
      (!t. (q2 0 = F) /\ (q2(SUC t) = out t)) /\
      (l4 t = T) /\
      (!t. (q5 0 = F) /\ (q5(SUC t) = l4 t)) /\
      (out t = (q5 t => l3 t | l4 t))) =
   (!t. (out 0 = T) /\ (out(SUC t) = (inp(SUC t) = ~out t)))")
: (* list # term)
Run time: 0.1s
```

3.3 Step 3: Break the goal into subgoals

This is the creative step, where the user has to use his knowledge in breaking up the goal into subgoals, apply proof strategies like induction, and use the lemmas needed. This step can be supported by developing appropriate user interfaces. Furthermore, multifarious domain-specific heuristics (e.g., for processors, controllers, signal processors, etc.) can be built in as specialized tactics for guiding the designer. Additionally, the design-specific decisions can also be incorporated via justifications as given in [37]. However, due to the very nature of the problem, a complete automation of this step is impossible in many cases. In the simple parity example, this step is superfluous and can be skipped.

3.4 Step 4: Simplify the subgoals

To curtail the complexity of the subgoals to be solved by an automatic prover, certain simplifications are in order. One of the main simplifications corresponds to the elimination of existentially quantified variables (internal lines) in the subgoals. The 'UNWIND' library [38] in HOL contains a rich set of rules and conversions for performing such simplifications. A closer study of this library, within our restricted higher-order context, has motivated us to examine the possibility of completely automating this step. This has been achieved by rewriting the subgoals into a kind of prenex normal form and then eliminating variables using the so called *comb-elim* and *seq-elim* rules, as described in the rest of this subsection.

3.4.1 Conversion to prenex normal form

Prenex normal form is an equivalent form of the original formula, containing quantifiers only at the beginning of the formula [39]. In our approach, the specification and the implementation in the subgoal are both transformed separately into hardware formulae $\mathcal{H}_s$ and $\mathcal{H}_i$. These formulae are prenex normal forms of the original formulae, where all existentially quantified variables appear at the beginning, followed by the universally quantified time variable. This transformation is performed by a set of rules as shown below (P and Q are formulae and t, t_1, t_2, x, and z are variables):[11]

Rule-1:	$\forall t.(P \wedge Q)$	$\rightsquigarrow$	$(\forall t.P) \wedge (\forall t.Q)$
Rule-2:	$\forall t.P$	$\rightsquigarrow$	P, if t is not free in P
Rule-3:	$\exists t.P$	$\rightsquigarrow$	$\exists z.[P]_t^z$, if z is not free in P
Rule-4:	$(\exists x.P) \wedge Q$	$\rightsquigarrow$	$\exists x.(P \wedge Q)$, if x is not free in Q
Rule-5:	$(\forall t_1.P) \wedge (\forall t_2.Q)$	$\rightsquigarrow$	$\forall t.([P]_{t_1}^t \wedge [Q]_{t_2}^t)$, if t is not free in P and Q

Each rule is applied repeatedly in the order given above, until it is not applicable anymore. Note that the termination of these rule applications is guaranteed. For the parity example the following occurs:

1. Using Rule-1, the formula in session box (3) is converted to an equivalent one where the quantified variable t occurs in front of each conjunct. The result is given in session box (4).

4

```
([],
 "!inp out.
   (?l1 q2 l3 l4 q5.
     (!t. l1 t = ~q2 t) /\
     (!t. l3 t = (inp t => l1 t | q2 t)) /\
     (!t t1. (q2 0 = F) /\ (q2(SUC t1) = out t1)) /\
     (!t. l4 t = T) /\
     (!t t2. (q5 0 = F) /\ (q5(SUC t2) = l4 t2)) /\
     (!t. out t = (q5 t => l3 t | l4 t))) =
   (!t. (out 0 = T) /\ (out(SUC t) = (inp(SUC t) = ~out t)))")
```

2. Using Rule-2, the extra quantifications over t are eliminated.
3. Using Rules 3 and 4, existential quantifications are moved outwards. These quantifications arise from the specifications of the used components. As all subcircuits in the pairity example are basic components, this step is not required in this example.
4. Using Rule-5, the quantifications over t are coerced and moved outwards.

This sequence of application results in the following formula:

5

```
([],
 "!inp out.
   (?l1 q2 l3 l4 q5.
     !t.
      (l1 t = ~q2 t) /\
      (l3 t = (inp t => l1 t | q2 t)) /\
      ((q2 0 = F) /\ (q2(SUC t) = out t)) /\
      (l4 t = T) /\
      ((q5 0 = F) /\ (q5(SUC t) = l4 t)) /\
      (out t = (q5 t => l3 t | l4 t))) =
   (!t. (out 0 = T) /\ (out(SUC t) = (inp(SUC t) = ~out t)))")
```

3.4.2 Elimination of internal lines

The elimination of internal lines comprises the application of two rules called the *comb-elim rule* and the *seq-elim rule*. The *comb-elim rule* can be implemented easily using the conversions and rules present in the UNWIND library. The *seq-elim rule*, however, requires additional conversions and rules.

The comb-elim rule corresponds to the elimination of those internal lines that are outputs of combinational components (figure 4). The rule can be written as follows:

Comb-elim:	$\exists \ell . \forall t . (\ell\ t \leftrightarrow Q\ t) \wedge P(\ell\ t)$	$\rightsquigarrow$	$\forall t . P(Q\ t)$

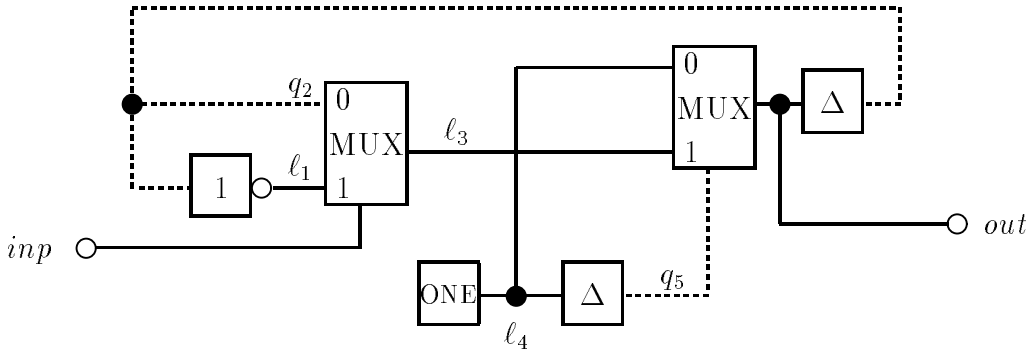


Figure 4: Parity implementation with combinational and sequential internal lines (dashed).

In applying the rule, one can imagine that ℓ represents an internal line, Q represents the definition of the combinational component whose output is ℓ, and P represents the rest of the circuit. Intuitively, this rule replaces the internal combinational lines by the definition of the components in terms of its inputs. This rule is applied repeatedly on the subgoal until no more combinational internal lines exist. It is obvious that this process always terminates[12] and that all such lines can be ultimately removed.

In the example, after minor syntactical changes, i.e., by reducing the scope of existentially quantified variables to their actual appearance, three applications of the comb-elim rule on the formula in session box (5) to remove the combinational internal lines ℓ_1, ℓ_3, and ℓ_4, and results in the formula shown in session box (6):

6

```
([],
 "!inp out.
   (?q2 q5.
     !t.
      ((q2 0 = F) /\ (q2(SUC t) = out t)) /\
      ((q5 0 = F) /\ (q5(SUC t) = T)) /\
      (out t = (q5 t => (inp t => ~q2 t | q2 t) | T))) =
   (!t. (out 0 = T) /\ (out(SUC t) = (inp(SUC t) = ~out t)))")
```

Analogously, the seq-elim rule eliminates sequential internal lines. It however uses a rule called *seq-elim-help*, which creates multiple copies of the formula whose need is explained below. The rules are as follows:

seq-elim:

$$\exists q.\forall t.(q\,0 \leftrightarrow \Phi_0)\wedge$$
$$\vdots$$
$$(q(suc^{n-1}\ 0) \leftrightarrow \Phi_{n-1})\wedge$$
$$(q(suc^{n}\ t) \leftrightarrow \Phi_{n})\wedge$$
$$P(q\,0,\ldots,q(suc^{n-1}\ 0),q(suc^{n}\ t),q(suc^{n+1}\ t),\ldots,q(suc^{n+m}\ t))$$
$$\rightsquigarrow$$
$$\forall t.P(\Phi_0,\ldots,\Phi_{n-1},\Phi_n,[\Phi_n]_t^{suc\ t},\ldots,[\Phi_n]_t^{suc^m\ t})$$

with $\Phi_i := \Psi_i(0, suc\ 0, \ldots, suc^{i-1}\ 0), 1 \leq i \leq (n-1)$, and
$\Phi_n := \Psi_n(t, suc\ t, \ldots, suc^{n-1}\ t)$
where Φ_i and Φ_n do not contain the variable q.

seq-elim-help: $\forall t.P\ t \quad \rightsquigarrow \quad \forall t.P\ 0 \wedge P(suc\ t)$

In the rule seq-elim, the sequential internal line is defined recursively using n initializations of the variable q, i.e., formulae Φ_0 to Φ_{n-1} do not depend on t, and the recursive step corresponding to formula Φ_n. Additionally, P corresponds to the formula, for that part of the circuit, which depends on q. If P already contains the variable q in the required form, i.e., $P(q\,0,\ldots,q(suc^{n-1}\ 0),q(suc^{n}\ t),q(suc^{n+1}\ t),\ldots,q(suc^{n+m}\ t))$,[13] then a simple substitution can take place as shown in the seq-elim rule. If P contains the variable q at time $suc^k t$ $(0 \leq k < n)$, then $n-k$ copies of it are made using the seq-elim-help rule to generate the necessary syntactical structure for seq-elim rule application.

In order to apply the seq-elim rule on formula (6) to eliminate q_2, we see that a copy has to be generated using the seq-elim-help-rule. A single application of the rule results in the formula in session box (7):

7

```
([],
 "!inp out.
   (?q2 q5.
     !t.
      ((q2 0 = F) /\ (q2(SUC t) = out t)) /\
      ((q5 0 = F) /\ (q5(SUC t) = T)) /\
      (out 0 = (q5 0 => (inp 0 => ~q2 0 | q2 0) | T)) /\
      (out(SUC t) =
       (q5(SUC t) => (inp(SUC t) => ~q2(SUC t) | q2(SUC t)) | T))) =
   (!t. (out 0 = T) /\ (out(SUC t) = (inp(SUC t) = ~out t)))")
```

Applying seq-elim for eliminating q_2, now yields the formula in session box (8):

```
([],                                                                    8
 "!inp out.
   (?q5.
     !t.
      ((q5 0 = F) /\ (q5(SUC t) = T)) /\
      (out 0 = (q5 0 => (inp 0 => ~F | F) | T)) /\
      (out(SUC t) = (q5(SUC t) => (inp(SUC t) => ~out t | out t) | T))) =
   (!t. (out 0 = T) /\ (out(SUC t) = (inp(SUC t) = ~out t)))")
```

Since the formula in session box (8) already contains the variable q_5 in the required form, it can also be eliminated by applying seq-elim to generate the formula in session box (9):

```
([],                                                                    9
 "!inp out.
   (!t.
     (out 0 = (F => (inp 0 => ~F | F) | T)) /\
     (out(SUC t) = (T => (inp(SUC t) => ~out t | out t) | T))) =
   (!t. (out 0 = T) /\ (out(SUC t) = (inp(SUC t) = ~out t)))")
```

The elimination rules are repeatedly applied on the subgoal until they are not applicable any more. Usually not all sequential internal lines can be removed, since the preconditions for the seq-elim rule application are sometimes violated, i.e., if Φ_n contains the variable q. Internal line variables $\vec{q}$, which cannot be eliminated, represent the state variables of the circuit and can thus occur only if the circuit is not combinational.

3.4.3 Further simplifications

The formula resulting after the internal line eliminations can be further simplified by reducing its equivalences of the corresponding output line definitions. When no more elimination rules are applicable, logical simplifications take place. This includes the elimination of constants T and F, which appear in the formulae due to the initializations of the sequential components, simplifications of conditionals, etc. Furthermore, if the goal is an equivalence, it is replaced by two implications that are rewritten with their assumptions. All the simplifications given in step 3.4 are contained in the function `simplify`:

```
#let goal2=simplify goal1;;                                            10
goal2 =
([],
 "((inp(SUC t) = ~out t) =
   (inp(SUC t) ==> ~out t) /\ (~inp(SUC t) ==> out t)) /\
  ((inp(SUC t) ==> ~out t) /\ (~inp(SUC t) ==> out t) =
   (inp(SUC t) = ~out t))")
: goal
Run time: 0.1s
```

3.5 Step 5: Automatic proof of the simplified subgoals

Having reduced the subgoals, an automated theorem prover can then be used to prove each subgoal separately. The next section is devoted to a description of the theory behind such a prover, called FAUST. In most cases, FAUST is capable of proving these subgoals without further aid. However, to speed up the proof process and to contain the space and runtime requirements, it may be useful to guide the proof process by adding some lemmas that can in turn be automatically proved by FAUST. However, in this example a single invocation of FAUST is sufficient to prove the remaining goal.

11

```
#faust_prove goal3;;
|- ((inp(SUC t) = ~out t) =
    (inp(SUC t) ==> ~out t) /\ (~inp(SUC t) ==> out t)) /\
   ((inp(SUC t) ==> ~out t) /\ (~inp(SUC t) ==> out t) =
    (inp(SUC t) = ~out t))
Run time: 0.1s
Intermediate theorems generated: 1
```

3.6 Step 6: Update library for future use

The specification and the correctness theorems generated for the design can then be stored within a library and recalled later while designing components that use the currently verified component. A hierarchy can thus be achieved in future proofs since the specification and the correctness theorems are sufficient and the current component need not be broken down to its implementation anymore. This reduces the complexity of the verification process since the specifications are more compact than the implementations.

12

```
#save_thm(`SPARITY_CORRECT`,(mk_thm goal));;
|- SPARITY_IMP(inp,out) = SPARITY_SPEC(inp,out)
Run time: 0.1s
Intermediate theorems generated: 1

#close_theory();;
() : void
Run time: 0.1s
Intermediate theorems generated: 1
```

```
#print_theory `parity`;;
The Theory parity
Parents --  HOL     basics
Constants --
  TOGGLE ":bool # bool -> bool"
  SPARITY_SPEC ":(num -> bool) # (num -> bool) -> bool"
  SPARITY_IMP ":(num -> bool) # (num -> bool) -> bool"
Definitions --
  TOGGLE  |- !inp out. TOGGLE(inp,out) = (inp = ~out)
  SPARITY_SPEC
    |- !inp out.
        SPARITY_SPEC(inp,out) =
        (!t. (out 0 = T) /\ (out(SUC t) = TOGGLE(inp(SUC t),out t)))
  SPARITY_IMP
    |- !inp out.
        SPARITY_IMP(inp,out) =
        (?l1 q2 l3 l4 q5.
          !t.
           NOT(q2 t,l1 t) /\
           MUX(inp t,l1 t,q2 t,l3 t) /\
           DELAY(out,q2) /\
           ONE(l4 t) /\
           DELAY(l4,q5) /\
           MUX(q5 t,l3 t,l4 t,out t))

Theorems --
  SPARITY_CORRECT  |- SPARITY_IMP(inp,out) = SPARITY_SPEC(inp,out)
******************** parity ********************
```

3.7 Embedment of MEPHISTO and FAUST

The verification system MEPHISTO, which contains the prover FAUST, is being integrated in a CAD design environment (CADENCE) so that the designer can continue to work in a conventional manner. The hierarchy tree (figure 1) is used by the hierarchy manager for interaction with MEPHISTO. By user interactions, the hierarchy manager executes the following actions

- Opening of a new HOL-theory.
- Loading the required parent theories for using the preproved components used within the implementation of this node.
- Converting the EDIF-netlist into a higher-order logic description and its storage in the theory.
- Reading the specification given by the designer and its storage in the theory.
- Formulating an appropriate goal.
- Starting MEPHISTO, which proves the goal with the help of FAUST, and writing the constructed proof tree in a protocol file.
- Storing the proved theorem in the theory for future use.

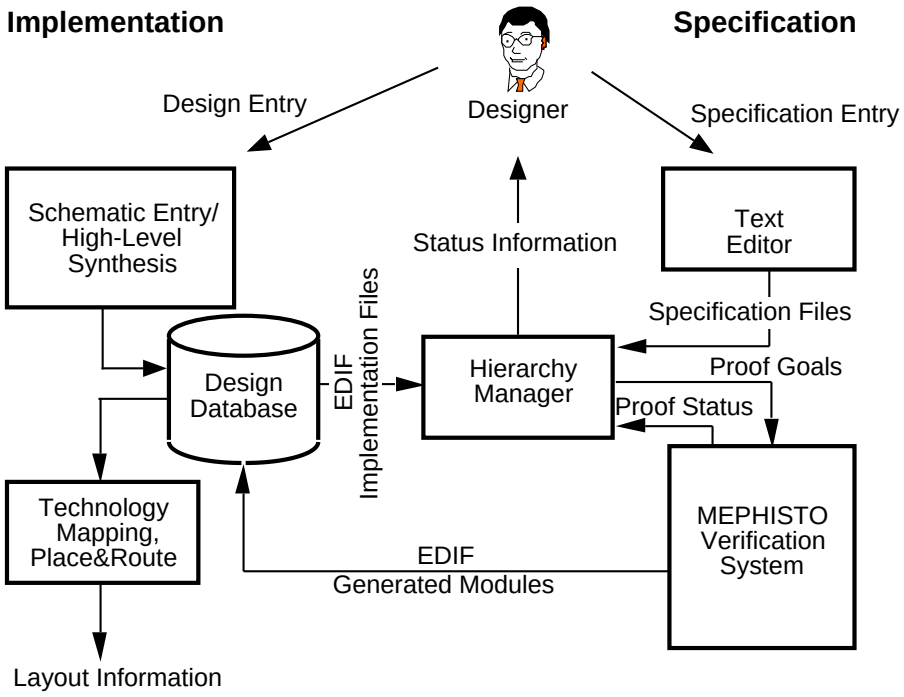


Figure 5: The embedment of the proof system in a CAD environment.

Figure 5 shows the components of the entire system.

Furthermore, it is also possible to specify and verify generalized n-bit structures as described in [40] and the corresponding instantiated netlists can be also generated.

4 A calculus for automating the proofs of subgoals

Due to the use of restricted higher-order logic for hardware descriptions, it is possible to use first-order techniques for hardware proofs. For example, the second-order goal

$$\forall f.\exists q.(q(t) \leftrightarrow f(t)) \wedge P(q(t)) \leftrightarrow P(f(t))$$

can be automatically solved by FAUST, since the required instance $[q \leftarrow f]$ can be computed by a simple unification. This unification is a first-order unification rather than a higher-order unification,[14] although it allows the substitution of function and predicate variables by other function and predicate variables of the same type. Solving such goals interactively within HOL can be quite tedious, especially for circuit designers. FAUST, the general-purpose, first-order prover, is based on a modified form of sequent calculus. This calculus was chosen because it is the closest to natural deduction, on which HOL is based. Sequent calculus and the related tableau calculus also have other advantages as compared to resolution, namely,

- readability of proofs,

- no need for normal forms,
- no need for skolemization,[15]
- easily extensible to modal, intuitionistic, and multivalued logics, and
- domain-specific rules, e.g., hardware specific, can be easily incorporated within the calculus.

These advantages motivated us to implement a first-order, automatic prover based on sequent calculus, although sequent calculus has its own problems. This section is devoted to the elimination of these problems so as to obtain an efficient implementation.

4.1 Brief introduction to sequent calculus

The normal sequent calculus ($\mathcal{SEQ}$) [41] uses the notion of sequents, a set of rules and an axiom scheme for proving formulae. A sequent can be defined as follows:

Definition 4.1 (Sequent) *A sequent is an ordered pair* (Γ, Δ) *of finite (possibly empty) sets of formulae* $\Gamma := \{\varphi_1, \ldots, \varphi_m\}, \Delta := \{\psi_1, \ldots, \psi_n\}$. *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 [42], [39] and are omitted here. Intuitively, a sequent is valid if the corresponding formula[16]

$$\left(\bigwedge_{i=1}^{m} \varphi_i\right) \rightarrow \left(\bigvee_{j=1}^{n} \psi_j\right)$$

is valid. The calculus based on such sequents contains several rules that reflect the semantics of the various operators (including quantifiers), and an axiom scheme that is a sequent $\Gamma \vdash \Delta$, such that Γ and Δ contain some common formula (i.e., $\Gamma \cap \Delta \neq \{\}$). An informal reason for the axiom scheme corresponds to the statement $\Phi \vdash \Phi$. Proving the correctness of any statement within $\mathcal{SEQ}$ then corresponds to iterative rule applications, which decompose the original sequent into simpler sequents such that axioms are obtained. This process can be visualized as a proof tree $\mathcal{P}$ (figure 6); a closed proof tree is one which has an axiom at each leaf node.

The rules can be classified into four types — α, β, γ and δ (cf. section 4.3). α- and β-rules are used to eliminate propositional junctors of the logic. β-rules branch a path but α-rules do not. δ-rules eliminate quantifiers by instantiating an *arbitrary new* variable. Thus $\alpha-, \beta-$ and δ-rules can be applied only finitely often on a specific sequent. Moreover, they are often called uncritical, since they can be applied deterministically. γ-rules, on the other hand, which add instances of a quantified formula, can be applied infinitely often, and the choice of the best term used for the instantiation is unknown at the time of rule application. This choice, however, influences the size and the closure of the proof tree. This rule poses a major hurdle in the efficient implementation of $\mathcal{SEQ}$. The problems with critical rule application also appear in the implementation of tableaux-based, first-order provers like HARP [43] which overcome these problems by using good heuristics in guessing the right term for substitution.

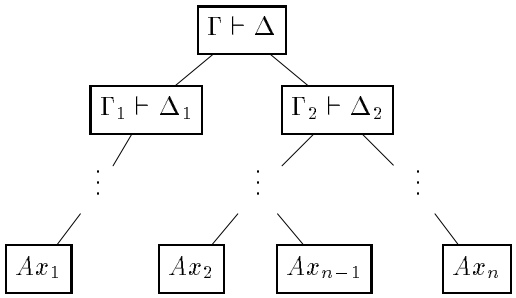


Figure 6: A Closed Proof Tree in $\mathcal{SEQ}$.

We on the other hand, use an exact approach like [44] or [39] by postponing the choice of the exact term to an appropriate stage. When the proof tree construction process is ripened, we use *first-order unification* for computing the terms for instantiation. This concept can be thought of as being similar to lazy evaluation within functional language implementations. Before delving into the details of our calculus, the notations used are stated:

For	Set of all first-order formulae
$\mathcal{T}$	Set of all first-order terms
$\mathcal{V}$	Set of all variables
$[]_x^t$	Substitution of a variable x by the term $t \in \mathcal{T}$
$\sigma = [\ldots []_{x_1}^{t_1} \ldots]_{x_n}^{t_n}$	A multiple simultaneous substitution

4.2 Modifications to $\mathcal{SEQ}$

4.2.1 Metavariables

The problem of computing the right choice of a term during the γ-rule application is undecidable, since the existence of an algorithm for term computation would lead to a decision procedure for first-order logic, which contradicts Church's Theorem [45]. However, this problem can be circumvented by plugging in a place-holder for the term and proceeding further with rule applications, until one reaches a stage where an appropriate term can be computed using unification. This place-holder is called a *metavariable*, since it does not belong to the language of first-order logic. Metavariables are not variables for the objects belonging to the domain of discourse, but are variables for the syntactic objects of the language, i.e., for terms.[17] However, for syntactical convenience, the set of variables $\mathcal{V}$ is split into the set $\mathcal{V}_\mathcal{I}$ and $\mathcal{V}_\mathcal{M}$. It is to be noted that the terms and formulae of the language must be built up exclusively from variables belonging to $\mathcal{V}_\mathcal{I}$, since metavariables are available only for the proof process.

4.2.2 Restricted sequents

The introduction of metavariables introduces new problems as far as the application of δ-rules is concerned. A δ-rule application requires that the variable[18] introduced for the quantifier elimination is new [42]. However, at the present juncture, we do not know the terms that instantiate the metavariables existing so far; hence the newly introduced variables should not appear in the terms to be instantiated for these metavariables. Therefore, restrictions are placed on the terms that the existing metavariables can take. The use of such restrictions led us to christen our calculus as $\mathcal{RSEQ}$ or *restricted sequent calculus.*

For each currently existing metavariable $m \in \mathcal{V}_{\mathcal{M}}$, all new variables $\in \mathcal{V}_{\mathcal{I}}$ introduced after the creation of that metavariable m are stored in its forbidden set $fs_m \subseteq \mathcal{V}_{\mathcal{I}}$. Using these concepts, the definition of the modified sequent, called a restricted sequent, can now be given as follows:

Definition 4.2 (Restricted sequent) *Given that $\Gamma, \Delta \subseteq$ For (sets of formulae) and $R \subseteq \mathcal{V}_{\mathcal{M}} \times 2^{\mathcal{V}_{\mathcal{I}}}$, then $\Gamma \vdash \Delta \parallel R$ is a restricted sequent. Γ is called the antecedent, Δ is called the succedent, and R is called the restriction of the sequent. $\parallel$ binds the restriction R to the sequent. The restriction R contains for each metavariable occurring in $\Gamma \cup \Delta$ an ordered pair $(m, fs_m) \subseteq \mathcal{V}_{\mathcal{M}} \times 2^{\mathcal{V}_{\mathcal{I}}}$, where fs_m is the forbidden set of m.*

In order to explain the semantics of restricted sequents, the following definitions are needed.

Definition 4.3 (Application of a substitution) *The application of a substitution σ on a restricted sequent is defined as: $\sigma(\Gamma \vdash \Delta \parallel R) := \sigma(\Gamma) \vdash \sigma(\Delta) \parallel R$.*

Definition 4.4 (Allowed metasubstitution) *A substitution σ is an allowed metasubstitution for a restricted sequent $\Gamma \vdash \Delta \parallel R$, iff*

1. *σ substitutes each metavariable and only metavariables of the restricted sequent by metavariable-free terms, and*
2. *No $y \in fs_m$ does occur in $\sigma(m)$ for each (m, fs_m).*

Definition 4.5 (Closed restricted sequent) *An allowed metasubstitution σ closes the restricted sequent $\Gamma \vdash \Delta \parallel R$, if $\sigma(\Gamma) \cap \sigma(\Delta) \neq \{\}$.*

Informally, the semantics of the restricted sequent can now be stated as follows: Given an arbitrary restricted sequent $\Gamma \vdash \Delta \parallel R$, then $\Gamma \vdash \Delta \parallel R$ is valid in $\mathcal{RSEQ}$, iff $\sigma(\Gamma) \vdash \sigma(\Delta)$ is valid in $\mathcal{SEQ}$ for each allowed metasubstitution σ.

The allowed metasubstitution σ is computed using *metaunification.* This metaunification is obtained by modifying any unification algorithm, e.g., Robinson's [46], in such a manner that only metavariables are considered as substitutable subterms. The computation of metaunifiers is given in section 4.4. Having defined the basic concepts of $\mathcal{RSEQ}$, we can now formulate the rules of $\mathcal{RSEQ}$. From now on we use the terms *restricted sequent* and *sequent* interchangeably, and it should be clear from the context what is being referred to.

4.3 Rules and Proof Trees in $\mathcal{RSEQ}$

In figure 7, Γ, Δ are sets of formulae, φ, and ψ are formulae, m is a metavariable,and x and y are variables. We use the notation φ, Γ instead of $\varphi \cup \Gamma$. Both the variable y and the metavariable m are new, i.e., they must not appear in the sequent until this point of time. The function ϱ_y, used for updating the restrictions of the existing metavariables, is defined recursively as follows:

$$\varrho_y(R) := \begin{cases} \{\} & if\ R = \{\} \\ \{(m, \{y\} \cup fs_m)\} \cup \varrho_y(R') & if\ R = \{(m, fs_m)\} \cup R' \end{cases}$$

The rules are to be read as follows:— given a sequent above the line, the rule converts it to the sequent(s) below the line. Considering semantical aspects, if the sequents below the line are valid, then the sequent above the line is also valid.

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

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

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

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

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

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

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

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

IFF_LEFT
$$\frac{\phi \leftrightarrow \psi, \Gamma \vdash \Delta \parallel R}{\Gamma \vdash \phi, \psi, \Delta \parallel R \quad \psi, \phi, \Gamma \vdash \Delta \parallel R}$$

IFF_RIGHT
$$\frac{\Gamma \vdash \phi \leftrightarrow \psi, \Delta \parallel R}{\phi, \Gamma \vdash \psi, \Delta \parallel R \quad \psi, \Gamma \vdash \phi, \Delta \parallel R}$$

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

FORALL_RIGHT
$$\frac{\Gamma \vdash \forall x.\phi, \Delta \parallel R}{\Gamma \vdash [\phi]^y_x, \Delta \parallel \varrho_y(R)}$$

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

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

Figure 7: Rules of $\mathcal{RSEQ}$.

Similar to the usual sequent calculus the rules of $\mathcal{RSEQ}$ can be classified as follows:

α-**Rules:** NOT_LEFT, NOT_RIGHT, AND_LEFT, OR_RIGHT, IMP_RIGHT
β-**Rules:** AND_RIGHT, OR_LEFT, IMP_LEFT, IFF_LEFT, IFF_RIGHT
γ-**Rules:** FORALL_LEFT and EXISTS_RIGHT
δ-**Rules:** FORALL_RIGHT and EXISTS_LEFT

Often the type of the rule and not the rule itself plays an important role, and the sequents obtained after the rule applications can be represented as follows:

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

Iterative application of the rules lead to proof trees, which are defined as follows:

Definition 4.6 (Proof trees) *Proof trees, which are binary trees whose nodes are labelled with restricted sequents, are defined recursively as follows:*

1. $\Gamma \vdash \Delta \parallel \{\}$ *is a proof tree.*
2. *Given that* Υ *is a path of a proof tree* $\mathcal{P}$ *with a leaf node* $\mathcal{S}$*, then* $\mathcal{P}'$ *is also a proof tree, where*
   - $\mathcal{P}'$ *is obtained by extending* Υ *by* α_1*, after an* α*-rule application on* $\mathcal{S}$*.*
   - $\mathcal{P}'$ *is obtained by branching* Υ *to* β_1 *and* β_2*, after a* β*-rule application on* $\mathcal{S}$*.*
   - $\mathcal{P}'$ *is obtained by extending* Υ *by* $\delta(y)$*, after a* δ*-rule application on* $\mathcal{S}$*, provided that the variable* y *does not appear in* $\mathcal{P}$*.*
   - $\mathcal{P}'$ *is obtained by extending* Υ *by* $\gamma(m)$*, after a* γ*-rule application on* $\mathcal{S}$*, provided that the metavariable* m *does not appear in* $\mathcal{P}$*.*

Definition 4.7 (Closed proof tree) *A proof tree* $\mathcal{P}$ *is called closed under the substitution* σ*, if* σ *closes each leaf node.*

The example in figure 8 illustrates the concepts explained so far. It is to be noted here, that since the formula to be proved does not need the application of δ-rules, all restrictions are empty and are therefore omitted. The substitution $\sigma = [[\ldots]^{a}_{m_1}]^{f(a)}_{m_2}$, for instance, closes the above-mentioned proof tree. Subsection 4.4 shows how such substitutions that close the proof tree can be computed.

Figure 9 shows another example, where the necessity of restrictions is illustrated. The formula $\exists x.\forall y.Pyx \rightarrow Pxx$ is not a theorem, since its negation, which is equivalent to $\forall x.\exists y.Pyx \wedge \neg Pxx$, has a model: Assume that Pyx means y is the father of x; then it is clear that each person x has a father y, but x is not the father of x. Thus the formula $\exists x.\forall y.Pyx \rightarrow Pxx$ cannot be derived in any sound calculus. Ignoring restrictions, however, would lead to a closed proof tree given in the upper half of figure 9, while the lower half shows an infinitely growing (sound) proof tree using restrictions.

Having given the concepts and illustrations of $\mathcal{RSEQ}$, we shall now state the soundness and completeness theorems. These two theorems can be proved as shown in [47]. It can also be derived from the correctness and completeness theorems of the normal sequent calculus [42], since the instantiation of the metavariables in proof trees of $\mathcal{RSEQ}$ leads to proof trees in $\mathcal{SEQ}$.

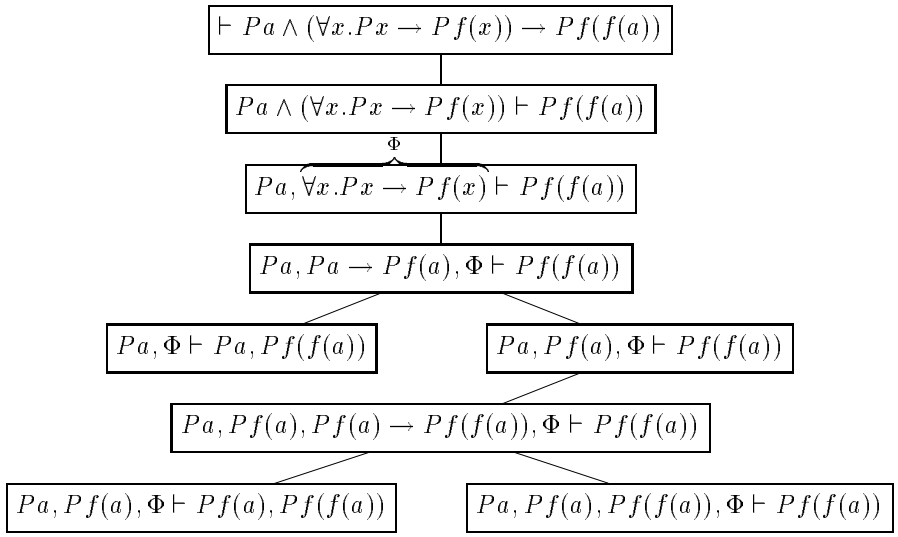


Figure 8: Example illustrating the concepts and rules of $\mathcal{RSEQ}$.

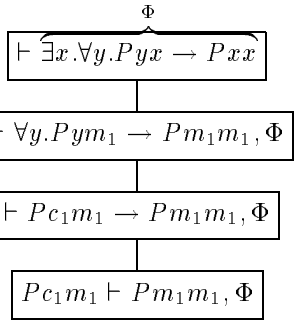
Theorem 4.1 (Completeness Theorem) *Given a valid formula φ, one can always construct a closed proof tree with root $\{\} \vdash \{\varphi\} \parallel \{\}$.*

However, it has to be noted that not every proof tree with the root $\{\} \vdash \{\varphi\} \parallel \{\}$ can be closed after a finite number of rule applications even if φ is valid, as shown in figure 10. Nevertheless, in subsection 5.1 we formulate sufficient conditions under which every proof tree construction for valid formulae can be closed after a finite number of rule applications. The converse of the theorem above is the correctness theorem:

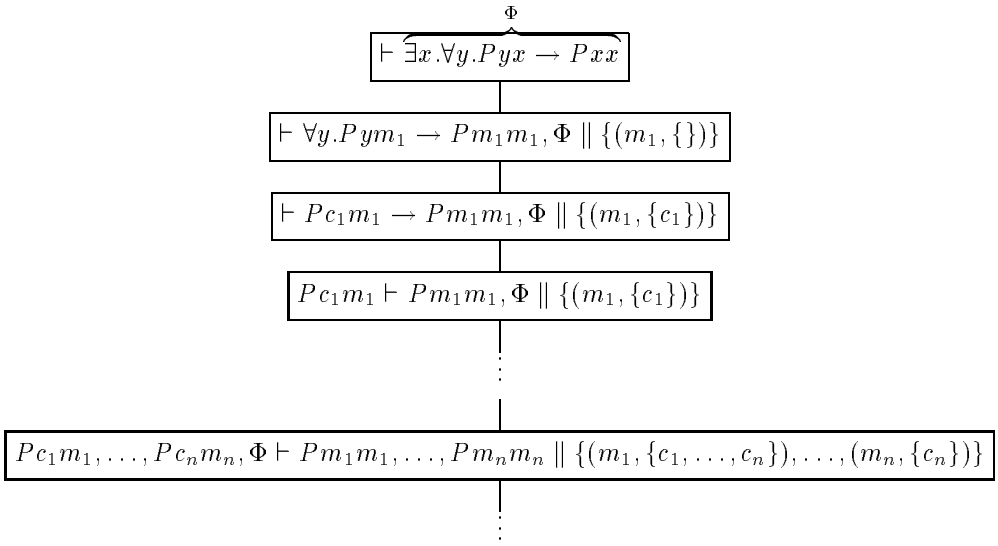
Theorem 4.2 (Correctness theorem) *φ is valid if there exists a closed proof tree with root $\{\} \vdash \{\varphi\} \parallel \{\}$.*

4.4 Computation of closing substitutions

The computation of closing substitutions depends upon the approach taken for realizing the proof tree, i.e., depth-first or breadth-first. In the depth-first approach, since the closed branches are not revisited in our implementation, all the necessary information (i.e., all possible allowed closing metasubstitutions) has to be carried over to the other unclosed branches. Contrariwise, the breadth-first approach simultaneously contains all the leaves to be closed, and therefore a single closing substitution can be calculated. In case of the depth-first proof strategy, a list of all possible allowable metasubstitutions are generated at each sequent before moving to the next one. Given the antecedent $\Gamma = \{\varphi_1, \ldots, \varphi_n\}$ and the succedent $\Delta = \{\psi_1, \ldots, \psi_m\}$ of a restricted sequent, the most general allowed



An unsound proof found by neglecting restrictions.



The infinitely growing sound proof tree with restrictions.

Figure 9: Example illustrating the need for restrictions.

metaunifiers σ_{ij} can be computed for each pair (φ_i, ψ_j), if φ_i and ψ_j are unifiable. 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 $\sigma_{ij}\eta$ continues to unify the pair (φ_i, ψ_j), since σ_{ij} is more general than $\sigma_{ij}\eta$. The aim is now to choose an appropriate refinement η that results in the closure of further sequents in the overall proof tree. Such closed sequents are all valid in $\mathcal{SEQ}$ since they correspond to axioms by definition. In section 5.2, the integration of such metaunification in the depth-first proof strategy is illustrated.

4.5 Comparing $\mathcal{RSEQ}$ with Fitting's Approach

As described in earlier sections, the introduction of metavariables and the restriction R effectively remove the nondeterminism in the application of γ-rules. A "dual approach" (Free variable Semantic Tableaux) has been given by Fitting [39]. In this approach, the application of a γ-rule introduces a new free variable,[19] and a δ-rule introduces a new function symbol, called a *Skolem function symbol*, whose arguments are all free variables existing in the branch (where the δ-rule has been applied). Since these variables can be detected at the time of the δ-rule application, these restrictions are static and not *dynamic* as in our approach.

Although both these approaches are equivalent, differences occur during the implementation and in general, it cannot be said which approach is the more efficient one. In Fitting's approach, if a δ-rule application takes place after a large number of free variables have been introduced, the Skolem function symbol turns out to be a complex term that has to be propagated down to all the future branches and furthermore increases the complexity of unification. On the other hand, if the number of free variables is small, then Fitting's approach would be more efficient than ours, since the restrictions are to be updated dynamically. However, a more efficient static implementation of restrictions can be implemented, as described in [40].

5 Implementation of FAUST

In order to obtain an efficient implementation of the prover, certain additional concepts are needed. For example, it can be observed that although the formula to be proved in figure 10 is valid, its validity cannot be shown if the rules are applied in a haphazard manner. It is therefore necessary that a certain precedence order for the rule application exists. This motivated us to use a fair application of rules as elaborated in the following subsection.

5.1 Fairness of rule application

Definition 5.1 (fair application) *An application of the rules is defined to be fair if no rule gets a continuing precedence over the others.*

In order to achieve a complete prover, a precedence hierarchy for the application of rules is defined as follows: $\alpha \gg \delta \gg \beta \gg \gamma$. Such a hierarchy ensures a fair application of the

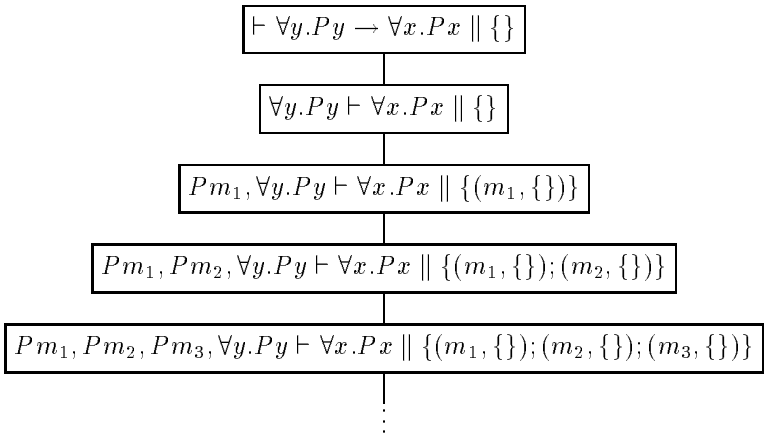


Figure 10: Infinite growth of a proof tree due to unfair rule applications.

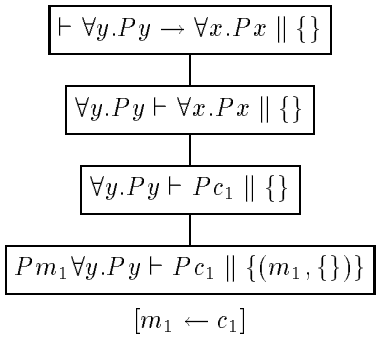


Figure 11: Fair application of rules to contain the growth of the proof tree.

rules. The uncritical rules (α, β, δ) 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, since further γ-rule applications are done on quantified variables that have not been instantiated so far. If no further γ-rules can be applied directly on the sequent 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 if the level saturation condition [48] is observed. The proof of this statement is given in [47]. Revisiting the example shown in figure 10, it can be seen that a fair application of rules generates a closed proof tree (figure 11).

5.2 Depth-first construction of the proof tree

The unification algorithm produces a 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 $\eta \circ \sigma$) is still a unifier for the two original formulae; however, it is 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 leftmost leaf 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 $\in \mathcal{V}_{\mathcal{I}}$, i.e., y does not occur in $\mathcal{S}$.

   (c) If a β-rule is applicable and neither an α-rule nor 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 an arbitrary new metavariable. The queue local to the sequent S is updated as stated in section 5.1.

   (e) $\mathcal{S} := \Gamma \vdash \Delta \parallel R$ now contains only atomic formulae and no more rules can be applied. Given $\Sigma_n = \{\sigma_1, \ldots, \sigma_k\}$, 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 allowed meta-substitutions for $\mathcal{S}$ using σ_i, represented as $\Pi_i = \{\pi_1^{(i)}, \ldots, \pi_{l_i}^{(i)}\}$. Now there are two possibilities, the first of which being that all Π_i's are empty. In this case, the sequent $\mathcal{S}$ cannot be closed at this step and we proceed to step 2(f). On the other hand, if one of the Π_i's is not empty, the substitution set Σ_{n+1} is calculated as follows:

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

   It is to be noted that each substitution belonging to Σ_{n+1} continues to unify the leaf sequents considered so far, as well as $\mathcal{S}$. $\mathcal{P}_{n+1}$ is now obtained by declaring the sequent $\mathcal{S}$ as closed and step 2 of the proof construction is continued with the next leftmost 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.

(f) When no substitutions that close the leaf $\mathcal{S}$ are found in step 2(e), i.e., all Π_i's are empty, 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(a).

We have also implemented a breadth-first algorithm within FAUST. Although the breadth-first algorithm is in general much slower than the depth-first algorithm, certain problems that are not solvable using a depth-first approach are provable using the breadth-first prover. 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.

5.3 Optimizations in FAUST

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 [47] for details):

- Metavariables are managed locally within the branches; thus the unifiers corresponding to those metavariables that do not appear in the current branch need not be refined.
- A partial order on the set of unifiers of a leaf is defined which is the generality of the unifiers. This ordering reduces the number of unifiers that are to be carried over to the next branch as only the most general ones are considered.
- The sequent is split up right at the start into different sets, each corresponding to the different rule types. 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 proved is not valid, since first-order predicate logic is only semi-decidable. Hence some upper bound of rule applications for the proof process can be set which reflects the complexity of the goal. If the bound is reached, the proof procedure terminates and reports that no proof has been found.

5.4 Embedding FAUST in HOL

FAUST has been implemented using the ML within HOL. In the beginning, we experimented with the idea of implementing $\mathcal{RSEQ}$ using tactics and the subgoal package of HOL. This, however, proved to be extremely slow, so we therefore resorted to the following two-pass method:

1. Prove the validity of statements outside the formal framework of HOL.

2. Generate a rule from the generated closed proof-tree as justification within HOL.

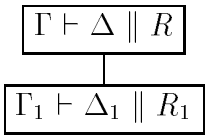
The validity of the statement proved during the first pass can be used within HOL by smuggling it in via the function 'mk_thm'. This result could be used by HOL users during the development phase as it is faster than generating a theorem by using the generated rule, as shown in table 2.

5.4.1 Relationship between $\mathcal{SEQ}$ and $\mathcal{RSEQ}$

In order to formally prove the statements (already proved by FAUST) easily in HOL, we need to find a mapping between $\mathcal{RSEQ}$ and $\mathcal{SEQ}$, since it is not possible to implement the rules of $\mathcal{RSEQ}$ easily within HOL. This mapping can then be used to generate a rule in HOL for proving the goal. As explained in section 4, the rules of $\mathcal{SEQ}$ and $\mathcal{RSEQ}$ differ only as far the δ and γ-rules are concerned. If these rules in $\mathcal{SEQ}$ are additionally supplied with parameters corresponding to the terms that are to be used for substitution, then the effect of the rule application in $\mathcal{SEQ}$ is exactly the same as that in $\mathcal{RSEQ}$. Each rule of $\mathcal{SEQ}$ is then coded as a rule of HOL, which is parametrized by the terms to be instantiated. These rules can then be combined in order to obtain a forward proof[20] for the whole theorem. A complete list of rules implemented in HOL can be found in appendix C.

5.4.2 Rule generation

The process of rule generation uses a closed proof tree generated by FAUST and two lists l_δ and l_γ, which contain the the instantiated terms of the δ- and γ-rules, respectively. It starts by proving theorems corresponding to each leaf by using the AXIOM_RULE. The rest of the proof is constructed by recursively composing the ALPHA_RULE, BETA_RULE, GAMMA_RULE and DELTA_RULE, as shown in figures 12 and 13.

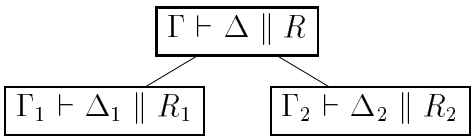


$$\text{ALPHA\_RULE}(\Gamma_1 \vdash \Delta_1 \parallel R_1) := \Gamma \vdash \Delta \parallel R$$
$$\text{GAMMA\_RULE } l_\gamma\ (\Gamma_1 \vdash \Delta_1 \parallel R_1) := \Gamma \vdash \Delta \parallel R$$
$$\text{DELTA\_RULE } l_\delta\ (\Gamma_1 \vdash \Delta_1 \parallel R_1) := \Gamma \vdash \Delta \parallel R$$

Figure 12: Rule generation for α, γ and δ-rules

6 Experimental Results

The prover FAUST[21] has been implemented in ML, available in the HOL proof assistant [18]. This choice has been made, since our prover has been primarily developed for reliev-



$$\text{BETA\_RULE}\ \ (\Gamma_1 \vdash \Delta_1 \parallel R_1)\ \ (\Gamma_2 \vdash \Delta_2 \parallel R_2) := \Gamma \vdash \Delta \parallel R$$

Figure 13: Rule generation for β-rules

ing a HOL user in proving simple but tedious first-order formulae. The set of benchmarks given by Pelletier [49] has been proved. All the benchmarks except Schubert's Steamroller Problem and the first-order formulae with equality (since our application domain does not require them) have been proved in acceptable times, although the ML used for implementation, is itself bootstrapped using an underlying Lisp. Andrew's Challenge was solved in 16.4 seconds, and the number of branches of the generated proof tree was 86. Having gained confidence about the correctness of FAUST, we then used MEPHISTO in combination with FAUST to prove some small circuits which were all automatically proved in a matter of seconds (table 2).

The columns in the table contain the following data: type of the goal (partial : $\rightarrow$; complete: $\leftrightarrow$); required times for expansion, simplification, and FAUST-proof of the goal; time required by the generated HOL-rule; and the number of theorems generated by it. None of the listed circuits required any manual interaction!

At present we have proved the correctness of sequential circuits such as parity, serial adder, flipflops, shift registers, twisted ring counter, sequence detectors, n-bit comparators, n-bit adders, and minmax. Currently we are in the midst of proving real-sized chips.

7 Conclusions and future work

In this article it has been shown that most parts of hardware proofs at the register-transfer level can be mechanized by taking benefit of two observations: firstly, the exploitation of the definite pattern existing in hardware proofs; and secondly, the use of a restricted higher-order language for hardware descriptions. The former has resulted in MEPHISTO, which breaks the original goal into easily solvable subgoals by following the sequence of steps given in section 3. The creative steps involved in proving the correctness can therefore be clearly identified, and the remaining steps can be automated. Furthermore, we have elucidated that, although higher-order language is essential for specifying hardware, it is a restricted form that can in general be handled by first-order proving techniques. For this purpose, a modified form of sequent calculus has been proposed. Based on this calculus, an implementation of the prover FAUST has been presented. We are currently embedding our approach within the CADENCE design framework, so that verification proceeds hand in hand with circuit design. The combination of MEPHISTO and FAUST results in complete automation for simple circuits. 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 proofs!

The main work for improving the hardware verification work in HOL so far has been

circuit	goal-type	expand [sec]	simplify [sec]	FAUST [sec]	HOL-proof [sec]	HOL-proof theorems
ADD2	$\leftrightarrow$	0.8	0.2	0.6	16.8	13869
BCD_CORRECT	$\rightarrow$	0.8	0.1	0.1	0.2	208
C_COUNT	$\rightarrow$	1.1	0.3	0.2	3.7	2762
DETECT11	$\rightarrow$	0.8	0.1	–	–	–
DETECT110	$\rightarrow$	0.5	0.2	525.7	702.2	25830
DMUX	$\rightarrow$	0.8	0.1	1.1	2.0	749
JK1	$\leftrightarrow$	0.8	< 0.1	< 0.1	0.6	437
JK2	$\leftrightarrow$	0.4	0.1	< 0.1	0.3	289
MUX	$\leftrightarrow$	0.7	0.1	< 0.1	1.1	729
SPARITY	$\leftrightarrow$	1.1	0.1	< 0.1	0.8	605
PAR_SER	$\rightarrow$	0.8	0.3	–	–	–
RESET_REG	$\leftrightarrow$	0.7	< 0.1	–	–	–
SADDER	$\leftrightarrow$	0.8	0.1	–	–	–
SAMPLER	$\rightarrow$	1.2	0.1	–	–	–
SAMPLER1	$\rightarrow$	0.8	0.5	–	–	–
SREG4	$\rightarrow$	0.8	0.2	43.0	95.4	9765
TRC_CORRECT	$\leftrightarrow$	0.9	0.2	–	–	–
TRC_011	$\leftrightarrow$	0.6	0.1	22.3	34.9	8446
TRC_100	$\leftrightarrow$	0.6	0.1	186.1	224.5	16828
XOR	$\rightarrow$	0.8	0.1	0.1	0.6	437
HAZARD	$\rightarrow$	0.9	0.2	21.9	24.5	1785

Table 2: Run times of various circuits

in using abstractions to make specifications and proofs more manageable [28, 50]. The contribution of MEPHISTO and FAUST represents a new direction in the problem of managing and solving hardware proofs, so that the user is free to concentrate on higher-level problems.

Recently, we have implemented a new version of FAUST in the SML under HOL90. This prover has been extended by a variety of further concepts to improve its efficiency. Actually, it is no longer based on sequent calculus, but on a related calculus that we called *tableau graph calculus* [51].

On the hardware domain, we are working on providing module generators for parametrizable components [40] by using a form of abstract datas types for bitvectors. These will be accompanied by a set of predicates that define operations on one's- and two's-complement representations of natural numbers and integers. Additionally, specialized proof procedures for finite-state machines, microprocessors, signal processors, etc. will be developed for enriching MEPHISTO, thus relaxing the present restriction to proofs at the register-transfer level. We are also working on the integration of hardware description languages as an alternate means of specification within the proof environment. Furthermore, we are investigating the possibilities of using more complex timing models. A first attempt at proving the existence of hazards in asynchronous circuits, is shown in appendix G.

Acknowledgments

The quality of this article has seen major improvement due to the the constructive criticisms given by the reviewer. We gratefully acknowledge these comments. This project has been partly financed by the "Deutsche Forschungsgemeinschaft", No. SFB-358.

Notes

1. Our studies have shown that resolution is not well suited for hardware verification and one uses mostly the rewriting mechanisms in Otter, for hardware proofs [52]. In [53] also, mostly rewriting has been used.
2. In backward proofs the goal is placed on a goal stack and is repeatedly split into subgoals by tactics. Solving all the subgoals then corresponds to the solution of the original goal. This is the most commonly used proof technique for solving goals within HOL. Additionally forward proofs can also be performed in HOL.
3. Some limited form of automation is available in HOL by using tactics.
4. Acronym for Managing Exhaustive Proofs of Hardware for Integrated circuit designers by Structuring Theorem-proving Operations
5. Acronym for First-order Automation using Unification in a modified Sequent calculus Technique
6. $\mathbf{N}$ denotes the set of natural number and $\mathbf{B}$ denotes the set of booleans.
7. For readability reasons, we use vectorized formulae, which should be read as
$$\left[\vec{\ell}t \leftrightarrow \vec{\Delta}(\vec{i}t, \vec{\ell}t, \vec{q}t)\right] \Leftrightarrow \bigwedge_{i=1}^{k} \left[\ell_i t \leftrightarrow \Delta_i(\vec{i}t, \vec{\ell}t, \vec{q}t)\right].$$
8. Although a specification using natural numbers would be more appropriate here, we shall not do so, since it would cloud the focus of this article. The details of hardware specification and verification using more complex data types are given in [40].
9. The netlists can be generated by a schematic entry tool or automatic synthesis programs.
10. The HOL notation for the logical connectives is given in appendix B.
11. The rules can be easily implemented in the HOL system. Some of them directly correspond to existing functions: rule-1 is `FORALL_CONJ_CONV`, rule-2 is `SPEC`, rule-4 is `LEFT_AND_EXISTS_CONV`, and rule-5 is a modification of `CONJ_FORALL_CONV`.
12. At the beginning there are only finitely many variables corresponding to internal lines. Each application of the comb-elim-rule eliminates exactly one variable, thus after a finite number of rule applications, no combinational internal line variable can exist anymore.
13. The seq-elim rule is applicable even if one or more arguments in P and the corresponding initializations are missing. Furthermore, additional arguments corresponding to the input/output and other internal lines can occur in Φ_i $(1 \leq i \leq n)$.
14. Higher-order unification is undecidable [54, 55].
15. Skolemization leads to terms that can increase the complexity of unification [56].
16. For a proper definition of the semantics we also have to define
$$\bigwedge_{i=1}^{0} \varphi_i = T \text{ and } \bigvee_{j=1}^{0} \varphi_j = F$$
17. Metavariables do not have an interpretation (i.e., a mapping into the domain of discourse) and must be substituted by metavariable-free terms before interpretation.
18. In [42] new *constants* are introduced during a δ-rule application. This implies that the signature contains an infinite number of constants. Equivalently, one could use a new variable instead of a new constant.
19. All free variables in this approach are also instantiated at an appropriate stage using unification.
20. Actually, we first generated tactics for backward proofs, which, however, turned out to be quite slow. The rule generation to perform forward proofs was suggested by [57].
21. A public domain version of FAUST embedded in HOL is available via anonymous ftp from i80s12.ira.uka.de (129.13.18.22).

References

[1] P.A. Subrahmanyam G. Birtwistle, editor. *Current Trends in Hardware Verification and Automated Theorem Proving.* Springer Verlag, 1988.

[2] P. Camurati and P. Prinetto. Formal verification of hardware correctness: Introduction and survey of current research. *IEEE-Computer*, pages 8–19, 1988.

[3] V. Stavridou, H. Barringer, and D.A. Edwards. Formal specification and verification of hardware: A comparative case study. In *25th Design Automation Conference*, pages 197–204, 1988.

[4] D. Gajski and R. Kuhn. Guest editors' introduction: New VLSI tools. *Computer*, 16(12):11–14, December 1983.

[5] R.A. Walker and D.E. Thomas. A model of design representation and synthesis. In *22th Design Automation Conference*, pages 453–457, 1985.

[6] A. Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning*, 5:127–139, 1989.

[7] O. Coudert, C. Berthet, and J.C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Workshop on Automatic Verification Methods for Finite State Systems*, June 1989.

[8] J.R. Burch, E.M. Clarke, K.L. McMilian, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *5th Annual Symposium on Logic in Computer Science*, 1990.

[9] Th. Kropf and H.-J. Wunderlich. A common approach to test generation and hardware-verification based on temporal logic. In *Proceedings of the International Test Conference*, pages 57–66, October 1991.

[10] J.R. Burch, E.M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *28th Design Automation Conference*, pages 403–407, 1991.

[11] N. C. E. Srinivas and V. D. Agrawal. Prove: Prolog based verifier. In *International Conference on Computer Aided Design*, pages 306–309, 1986.

[12] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook.* Academic Press, 1988.

[13] L. Wos, R. Overbeek, E. Lusk, and J. Boyle. *Automated Reasoning: Introduction and Applications.* Prentice-Hall, 1984.

[14] C. M. Angelo, D. Verkest, L. Claesen, and H. De Man. Formal hardware verification in HOL and in Boyer-Moore: A comparative analysis. In *International Workshop on the HOL Theorem Proving System and its Applications*, pages 340–347. IEEE Press, 1991.

[15] F.K. Hanna and N. Daeche. Specification and verification of digital systems using higher-order predicate logic. *IEE Proc. Pt. E*, 133:242–254, 1986.

[16] M.J.C. Gordon. A proof generating system for higher-order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*. Kluwer, 1988.

[17] M.J.C. Gordon, R. Milner, and C.P. Wadsworth. *A Mechanized Logic of Computation*. Number 78 in Lecture Notes in Computer Science. Springer Verlag, 1979.

[18] M.J.C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. Milne and P.A. Subrahmanyam, editors, *Formal aspects of VLSI Design*. North-Holland, 1986.

[19] J. Joyce. More reasons why higher-order logic is a good formalism for specifying and verifying hardware. In *International Workshop on Formal Methods in VLSI Design*, Miami,1991.

[20] *Third HOL Users Meeting*. unpublished, 1990.

[21] *International Workshop on the HOL Theorem Proving System and its Applications*, University of California, Davis, August 1991. IEEE Press.

[22] A. Cohn. A proof of correctness of the viper microprocessor: The first level. In P.A. Subrahmanyam G. Birtwistle, editor, *VLSI Specification, Verification and Synthesis*. Kluwer, 1988.

[23] A. Cohn. Correctness properties of the viper block model: The second level. In *Current Trends in Hardware Verification and Automated Theorem Proving*. Springer Verlag, 1988.

[24] W.J. Cullyer. *Implementing Safety Critical Systems: The VIPER Microprocessor*, chapter VLSI Specification, Verification and Synthesis, G. Birtwistle and P.A. Subrahmanyam. Kluwer, 1988.

[25] J. Joyce. Formal verification and implementation of a microprocessor. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*. Kluwer, 1988.

[26] F.K. Hanna, M. Longley, and N. Daeche. Formal synthesis of digital systems. In L. Claesen, editor, *Applied Formal Methods for Correct VLSI Design*. North Holland, 1989.

[27] M. Aargaard and M. Leeser. A methodology for reusable hardware-proofs. In *International Workshop on Higher Order Logic and its Applications*, pages 177–196, 1992.

[28] P. Windley. Microprocessor verification. In *International Workshop on the HOL Theorem Proving System and its Applications*, pages 32–37. IEEE Press, 1991.

[29] S. Finn, M. Fourman, M. Francis, and B. Harris. Formal system design - interactive synthesis based on computer assisted formal reasoning. In *Intern. Workshop on Applied Formal Methods for Correct VLSI Design*, November 1989. Leuven.

[30] E. Mayger and M. P. Fourman. Integration of formal methods with system design. In *International Conference on Very Large Scale Integration*, pages 3a.2.1–3a.2.11, 1991. Edinburgh.

[31] K. Schneider, R. Kumar, and Th. Kropf. Structuring hardware proofs: First steps towards automation in a higher-order environment. In P.B. Denyer A. Halaas, editor, *International Conference on Very Large Scale Integration*, pages 81–90. North Holland, Edingburgh, 1991.

[32] K. Schneider, R. Kumar, and Th. Kropf. Automating most parts of hardware proofs in HOL. In A. Skou K.G. Larsen, editor, *Workshop on Computer Aided Verification*, number 575 in Lecture Notes in Computer Science, pages 365–375. Springer Verlag, 1991.

[33] R. Kumar, Th. Kropf, and K. Schneider. First steps towards automating hardware proofs in hol. In *International Workshop on the HOL Theorem Proving System and its Applications*, pages 190–193. IEEE Press, 1991.

[34] R. Kumar, Th. Kropf, and K. Schneider. Integrating a first-order automatic prover in the hol environment. In *International Workshop on the HOL Theorem Proving System and its Applications*, pages 170–176. IEEE Press, 1991.

[35] K. Schneider, R. Kumar, and Th. Kropf. The FAUST prover. In D. Kapur, editor, *11th Conference on Automated Deduction*, number 607 in Lecture Notes in Computer Science, pages 766–770. Springer Verlag, Albany, New York,1992.

[36] R. Boulton, M. Gordon, J. Herbert, and J. van Tassel. The HOL verification of ELLA designs. In *International Workshop on Formal Methods in VLSI Design*, January 1991.

[37] S. Kalvala, M. Archer, and K. Levitt. A methodology for integrating hardware design and verification. In *International Workshop on Formal Methods in VLSI Design*, January Miami,1991.

[38] M.J.C. Gordon, Th. Melham, D. Sheperd, and A. Boulton. *The UNWIND Library.* Manual part of the HOL system, 1988.

[39] M. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer Verlag, 1990.

[40] K. Schneider, R. Kumar, and Th. Kropf. Modelling generic hardware structures by abstract datatypes. In M. Gordon L. Claesen, editor, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 419–429. Elsevier Science Publishers, 1992.

[41] G. Gentzen. Untersuchungen über das logisches Schließen. *Mathematische Zeitschrift*, 1:176–210, 1935.

[42] J.H. Gallier. *Logic for Computer Science—Foundations of Automated Theorem Proving*. Number 5 in Computer Science and Technology Series. Harper & Row, 1986.

[43] F. Oppacher and E. Suen. HARP: A tableau-based theorem prover. *Journal of Automated Reasoning*, 4:69–100, 1988.

[44] S.V. Reeves. Semantic tableau as a framework for automated theorem proving. Technical report, Department of Computer Science, Queen Mary College, University of London, 1987.

[45] A. Church. A note on the Entscheidungsproblem. *Journal of Symbolic Computation*, 1, 1936.

[46] J.A. Robinson. A machine oriented logic based on the resolution principle. *Journal of Automated Reasoning*, 12(1):23–41, 1965.

[47] K. Schneider. Ein Sequenzenkalkül für HOL. Diploma thesis, Unversity of Karlsruhe, 1991.

[48] R. Manthey and F. Bry. SATCHMO: A theorem prover implemented in prolog. In *Proceedings of 9th Conference on Automated Deduction*, Lecture Notes in Computer Science. Springer Verlag, 1988.

[49] F.J. Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2:191–216, 1986.

[50] J. Joyce. *Multilevel Verification of Microprocessor-Based Systems*. PhD thesis, Unversity of Cambridge, December 1989.

[51] K. Schneider, R. Kumar, and Th. Kropf. Efficient representation and computation of tableau proofs. In M. Gordon L. Claesen, editor, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 471–492. Elsevier Science Publishers, Leuven, 1992.

[52] H. Vogelsang. Hardware verifikation mit Otter. Studienarbeit, Institut für Rechnerentwurf und Fehlertoleranz, Universität Karlsruhe, June 1991.

[53] P. Camurati, T. Margaria, and P. Prinetto. Resolution based correctness proofs of synchronous circuits. In *European Design Automation Conference*, pages 11–15, 1991. Amsterdam.

[54] G. Huet. The undecidability of unification in third order logic. *Information and Control*, 22(3):257 – 367, 1973.

[55] D. Goldfarb. The undecidability of the second order unification. *Journal of Theoretical Computer Science*, 13:225–230, 1981.

[56] W. Bibel. *Automated Theorem Proving.* Vieweg Verlag, 1987.

[57] R. Arthan. Private communication, 1991.

A Specification of basic components

Component	Definition
ONE(out)	out
ZERO(out)	$out \leftrightarrow F$
NOT(in,out)	$out \leftrightarrow \neg in$
AND(e_1,e_2,a)	$a \leftrightarrow (e_1 \wedge e_2)$
OR(e_1,e_2,a)	$a \leftrightarrow (e_1 \vee e_2)$
EQUIV(e_1,e_2,a)	$a \leftrightarrow (e_1 \leftrightarrow e_2)$
NAND(e_1,e_2,a)	$a \leftrightarrow \neg(e_1 \wedge e_2)$
NOR(e_1,e_2,a)	$a \leftrightarrow \neg(e_1 \vee e_2)$
XOR(e_1,e_2,a)	$a \leftrightarrow \neg(e_1 \leftrightarrow e_2)$
MUX(s,e_1,e_2,a)	$a \leftrightarrow (s \Rightarrow e_1 \mid e_2)$
DELAY(e,a)	$\forall t.(a\ 0 \leftrightarrow F) \wedge (a(suc\ t) \leftrightarrow e\ t))$
JK(j,k, q)	$\forall t.(q\ 0 \leftrightarrow F) \wedge (q(suc\ t) \leftrightarrow (j\ t \wedge q\ t) \vee (\neg k\ t \wedge q\ t))$
T(e,a)	$\forall t.(q\ 0 \leftrightarrow F) \wedge (q(suc\ t) \leftrightarrow \neg(e\ t \leftrightarrow q\ t))$
DR(r,e,q)	$\forall t.(q\ 0 \leftrightarrow F) \wedge (q(suc\ t) \leftrightarrow (\neg r\ t \wedge e\ t))$

Table 3: Basic components

B HOL-Syntax

Symbol	HOL-equivalent
$\neg$	~
$\wedge$	/\
$\vee$	\/
$\rightarrow$	==>
$\leftrightarrow$	=
$\forall$	!
$\exists$	?

Table 4: HOL-Syntax

C Implemented $\mathcal{SEQ}$-rules

NOT_LEFT_RULE
$$\frac{\Gamma \vdash \phi, \Delta}{\neg\phi, \Gamma \vdash \Delta}$$

NOT_RIGHT_RULE
$$\frac{\phi, \Gamma \vdash \Delta}{\Gamma \vdash \neg\phi, \Delta}$$

AND_LEFT_RULE
$$\frac{\phi, \psi, \Gamma \vdash \Delta}{\phi \wedge \psi, \Gamma \vdash \Delta}$$

AND_RIGHT_RULE
$$\frac{\Gamma \vdash \phi, \Delta \quad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \wedge \psi, \Delta}$$

OR_LEFT_RULE
$$\frac{\phi, \Gamma \vdash \Delta \quad \psi, \Gamma \vdash \Delta}{\phi \vee \psi, \Gamma \vdash \Delta}$$

OR_RIGHT_RULE
$$\frac{\Gamma \vdash \phi, \psi, \Delta}{\Gamma \vdash \phi \vee \psi, \Delta}$$

IMP_LEFT_RULE
$$\frac{\Gamma \vdash \phi, \Delta \quad \psi, \Gamma \vdash \Delta}{\phi \rightarrow \psi, \Gamma \vdash \Delta}$$

IMP_RIGHT_RULE
$$\frac{\phi, \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \rightarrow \psi, \Delta}$$

IFF_LEFT_RULE
$$\frac{\Gamma \vdash \phi, \psi, \Delta \quad \psi, \phi, \Gamma \vdash \Delta}{\phi \leftrightarrow \psi, \Gamma \vdash \Delta}$$

IFF_RIGHT_RULE
$$\frac{\phi, \Gamma \vdash \psi, \Delta \quad \psi, \Gamma \vdash \phi, \Delta}{\Gamma \vdash \phi \leftrightarrow \psi, \Delta}$$

FORALL_LEFT_RULE τ
$$\frac{[\phi]^{\tau}_{x}, \forall x.\phi, \Gamma \vdash \Delta}{\forall x.\phi, \Gamma \vdash \Delta}$$

FORALL_RIGHT_RULE y
$$\frac{\Gamma \vdash [\phi]^{y}_{x}, \Delta}{\Gamma \vdash \forall x.\phi, \Delta}$$

EXISTS_LEFT_RULE y
$$\frac{[\phi]^{y}_{x}, \Gamma \vdash \Delta}{\exists x.\phi, \Gamma \vdash \Delta}$$

EXISTS_RIGHT_RULE τ
$$\frac{\Gamma \vdash [\phi]^{\tau}_{x}, \exists x.\phi, \Delta}{\Gamma \vdash \exists x.\phi, \Delta}$$

Figure 14: Rules of $\mathcal{SEQ}$

D Example circuits

D.1 Two bit full adder

```
ADD2_SPEC(i0,i1,j0,j1,s0,s1,c_out) =
  (s0 = i0 /\ ~j0 \/ ~i0 /\ j0) /\
  (s1 =
   (i1 /\ ~j1 \/ ~i1 /\ j1) /\ ~(i0 /\ j0) \/
   ~(i1 /\ ~j1 \/ ~i1 /\ j1) /\ i0 /\ j0) /\
  (c_out =
   i0 /\ i1 /\ (j1 \/ j0) \/
   (i1 \/ i0) /\ j1 /\ j0 \/
   i1 /\ ~i0 /\ j1 /\ ~j0)
```

```
ADD2_IMP(i0,i1,j0,j1,s0,s1,c_out) =
   (?w1 w2 w3 w4.
     XOR(i0,j0,s0) /\
     AND(i0,j0,w1) /\
     XOR(i1,j1,w2) /\
     AND(i1,j1,w3) /\
     XOR(w1,w2,s1) /\
     AND(w1,w2,w4) /\
     OR(w4,w3,c_out))
```

```
ADD2_IMP (i0,i1,j0,j1,s0,s1,c_out) = ADD2_SPEC (i0,i1,j0,j1,s0,s1,c_out)
```

D.2 BCD-Code Checker

```
BCD_CORRECT_SPEC(inp,out) =
(!t. out(SUC(SUC t)) =
       ~inp(SUC(SUC t)) \/
       ~inp(SUC t) /\ ~inp t)
```

```
BCD_CORRECT_IMP(inp,out) =
 (?w1 w2 w3 w4.
   !t.
    DELAY(inp,w1) /\
    DELAY(w1,w2) /\
    OR(w1 t,w2 t,w3 t) /\
    AND(inp t,w3 t,w4 t) /\
    NOT(w4 t,out t))
```

```
BCD_CORRECT_IMP(inp,out) ==> BCD_CORRECT_SPEC(inp,out)
```

D.3 A Counter

```
C_COUNT_SPEC(ctl,out_0,out_1,c_out) =
 (!t.
   (out_0(SUC t) =
       ~out_0 t /\ ~out_1 t \/
         ctl t /\ ~out_0 t) /\
   (out_1(SUC t) =
         out_0 t /\ ~out_1 t \/
         ctl t  /\ ~out_0 t /\ out_1 t) /\
   (c_out t =
         ~ctl t /\ ~out_0 t /\ out_1 t \/
           ctl t /\ out_0 t /\ out_1 t))
```

```
C_COUNT_IMP(ctl,out_0,out_1,c_out) =
   (?w2 w3 w4 w5 w6 w7 w8 w9 w10 w11 w12.
     !t.
      NOT(ctl t,w4 t) /\
      AND3(w4 t,out_1 t,w6 t,w3 t) /\
      AND3(out_1 t,ctl t,w6 t,w7 t) /\
      NOT(out_1 t,w5 t) /\
      AND(w5 t,out_0 t,w8 t) /\
      AND(ctl t,w6 t,w9 t) /\
      NOT(out_0 t,w6 t) /\
      AND(w6 t,w5 t,w10 t) /\
      OR(w2 t,w3 t,c_out t) /\
      OR(w7 t,w8 t,w11 t) /\
      OR(w9 t,w10 t,w12 t) /\
      DELAY(w11,out_1) /\
      DELAY(w12,out_0))
```

```
"C_COUNT_IMP(ctl,out_0,out_1,c_out) ==> C_COUNT_SPEC(ctl,out_0,out_1,c_out)
```

D.4 Detector circuits

D.4.1 11-detector

```
DETECT11_SPEC(e,a) =
       (!t. a(SUC t) = e t /\ e(SUC t))
```

```
DETECT11_IMP(e,a) =
        (?q. !t. DELAY(e,q) /\ AND(q t,e t,a t))
```

```
DETECT11_IMP(e,a)==>DETECT11_SPEC(e,a)
```

D.4.2 110-detector

```
  DETECT110_SPEC(e,a) =
    (!t. a(SUC(SUC t)) =
         ~e(SUC(SUC t)) /\ e(SUC t) /\ e t)
```

```
 DETECT110_IMP(e,a) =
    (?l1 l2 l3 q4 l5 l6 q7.
      !t.
       NOT(e t,l1 t) /\
       NOT(l5 t,l6 t) /\
       AND(e t,l5 t,l2 t) /\
       AND(e t,l6 t,l3 t) /\
       DELAY(l2,q4) /\
       DELAY(l3,q7) /\
       AND(l1 t,q4 t,a t) /\
       OR(q4 t,q7 t,l5 t))
```

```
DETECT110_IMP(e,a) ==> DETECT110_SPEC(e,a)
```

D.5 A Multiplexer with a Delay Time

```
DMUX_SPEC(sel,in1,in2,out:num->bool) =
    !t. sel t =>
        (out (SUC t) = in1 t) |
        (out (SUC t) = in2 t)
```

```
DMUX_IMP(sel,in1,in2,out) =
    (?l1 l2 l3 l4.
      !t.
       AND(sel t,in1 t,l1 t) /\
       NOT(sel t,l3 t) /\
       AND(l3 t,in2 t,l2 t) /\
       OR(l1 t,l2 t,l4 t) /\
       DELAY(l4,out))
```

```
DMUX_IMP(sel,in1,in2,out) ==> DMUX_SPEC(sel,in1,in2,out)
```

D.6 A Multiplexer with a Static Hazard

```
 HAZARD_IMP(in1,in2,in3,out) =
    (?l1 l2 l3 l4 l5 q6 q7 q8.
      !t.
       NOT(in1 t,l1 t) /\
       DELAY(l1,l2) /\
       AND(l2 t,in2 t,l3 t) /\
       DELAY(l3,l4) /\
       AND(in1 t,in3 t,l5 t) /\
       DELAY(l5,q6) /\
       DELAY(q6,q7) /\
       DELAY(q7,q8) /\
       OR(l4 t,q8 t,out t))

HAZARD_IMP(in1,in2,in3,out) ==>
        (in3 0 /\ in2 0 /\ ~in1 0)  /\
        ( (!t.in3 (SUC t)) /\ (!t. in2(SUC t)) /\ (!t. in1 (SUC t)) )
        ==> (?t. out t /\ ~out(SUC t) /\ out(SUC(SUC t)) )
```

D.7 Flipflops

```
JK(j,j,q) = TFF(j,q)

(?k. !t. (NOT(j t,k t) /\ (JK (j,k,q)))) = DELAY(j,q)
```

D.8 Multiplexer without a Delay Time

```
   MUX_IMP(sel,in1,in2,out) =
    (?l1 l2 l3.
      AND(sel,in1,l1) /\
      NOT(sel,l3) /\
      AND(l3,in2,l2) /\
      OR(l1,l2,out))

MUX_IMP(sel,in1,in2,out) = MUX(sel,in1,in2,out)
```

D.9 A Serial Even Parity Checker

```
SPARITY_SPEC(inp,out) =
 !t.(out 0=T) /\
     (out(SUC t) =  TOGGLE(inp(SUC t),(out t)))
where
     TOGGLE(inp,out) = (inp = ~out)
```

```
SPARITY_IMP(inp,out) =
    (?l1 q2 l3 l4 q5.
      !t.
       NOT(q2 t,l1 t) /\
       MUX(inp t,l1 t,q2 t,l3 t) /\
       DELAY(out,q2) /\
       ONE(l4 t) /\
       DELAY(l4,q5) /\
       MUX(q5 t,l3 t,l4 t,out t))

SPARITY_IMP(inp,out) = SPARITY_SPEC(inp,out)
```

D.10 A 3-bit Parallel to Serial Converter

```
PAR_SER_SPEC(ctl,s_in,inp1,
             inp2,inp3,out) =
    (!t.
      (out 0 = F) /\
      (out(SUC(SUC(SUC t))) =
       (ctl(SUC(SUC t)) =>
        inp1(SUC(SUC t)) |
        (ctl(SUC t) => inp2(SUC t) |
           (ctl t => inp3 t | s_in t)))))
```

```
PAR_SER_IMP(ctl,s_in,inp1,
             inp2,inp3,out) =
  (?w1 w2 w3 w4 w5.
    !t.
     MUX(ctl t,inp3 t,s_in t,w1 t) /\
     DELAY(w1,w2) /\
     MUX(ctl t,inp2 t,w2 t,w3 t) /\
     DELAY(w3,w4) /\
     MUX(ctl t,inp1 t,w4 t,w5 t) /\
     DELAY(w5,out))
```

```
PAR_SER_IMP(ctl,s_in,inp1,inp2,inp3,out) ==>
              PAR_SER_SPEC(ctl,s_in,inp1,inp2,inp3,out)
```

D.11 A resetable One bit Register

```
RESET_REG_SPEC(reset,in,out) =
 (!t.(out 0 = F) /\
     (out(SUC t) =
(reset t => T | in t)))
```

```
RESET_REG_IMP(reset,in,out) =
(?l1 l2.!t.
   ONE(l1 t) /\
   MUX(reset t,l1 t,in t,l2 t) /\
   DELAY(l2,out))
```

```
RESET_REG_IMP(reset,in,out) = RESET_REG_SPEC(reset,in,out)
```

D.12 A Serial Adder

```
SADDER_SPEC(in1,in2,out) =
 (?q.!t.
    (q 0 = F) /\
    (q(SUC t) =
       (q t => (in1 t \/ in2 t) |
               (in1 t /\ in2 t))) /\
    (out t = ((in1 t = in2 t) = q t)))
```

```
SADDER_IMP(in1,in2,out) =
   (?l1 l2 l3 l4 q5.
      !t.
       OR(in1 t,in2 t,l1 t) /\
       AND(in1 t,in2 t,l2 t) /\
       EQUIV(in1 t,in2 t,l3 t) /\
       MUX(q5 t,l1 t,l2 t,l4 t) /\
       DELAY(l4,q5) /\
       EQUIV(l3 t,q5 t,out t))
```

```
SADDER_IMP(in1,in2,out) = SADDER_SPEC(in1,in2,out)
```

D.13 A Sampler

```
SAMPLER_SPEC(sel,in,out_0,
out_1,out_2,out_3) =
(!t.
  (out_0(SUC(SUC(SUC(SUC t)))) =
     (sel(SUC(SUC(SUC(SUC t))))
=> in t | F)) /\
  (out_1(SUC(SUC(SUC t))) =
     (sel(SUC(SUC(SUC t)))
=> in t | F)) /\
  (out_2(SUC(SUC t)) =
     (sel(SUC(SUC t))
=> in t | F)) /\
  (out_3(SUC t) =
     (sel(SUC t) => in t | F)))
```

```
SAMPLER_IMP1(seq,in,out_0,out_1,out_2,out_3) =
 (?q1 q2 q3 q4.
  !t.
   DELAY(in,q1) /\
   DELAY(q1,q2) /\
   DELAY(q2,q3) /\
   DELAY(q3,q4) /\
   CONDG(seq t,q1 t,out_3 t) /\
   CONDG(seq t,q2 t,out_2 t) /\
   CONDG(seq t,q3 t,out_1 t) /\
   CONDG(seq t,q4 t,out_0 t))
```

```
SAMPLER_IMP1(sel,in,out_0,out_1,out_2,out_3) ==>
       SAMPLER_SPEC(sel,in,out_0,out_1,out_2,out_3)
```

D.14 Sampler implemented by Multiplexers

```
SAMPLER_SPEC(sel,in,out_0,
       out_1,out_2,out_3) =
  (!t.
    (out_0(SUC(SUC(SUC(SUC t)))) =
      (sel(SUC(SUC(SUC(SUC t))))
=> in t | F)) /\
    (out_1(SUC(SUC(SUC t))) =
      (sel(SUC(SUC(SUC t)))
=> in t | F)) /\
    (out_2(SUC(SUC t)) =
      (sel(SUC(SUC t))
=> in t | F)) /\
    (out_3(SUC t) =
      (sel(SUC t) => in t | F)))
```

```
SAMPLER_IMP2(seq,in,out_0,out_1,out_2,out_3) =
 (?q1 q2 q3 q4 l5.
  !t.
    DELAY(in,q1) /\
    DELAY(q1,q2) /\
    DELAY(q2,q3) /\
    DELAY(q3,q4) /\
    ZERO(l5 t) /\
    MUX(seq t,q1 t,l5 t,out_3 t) /\
    MUX(seq t,q2 t,l5 t,out_2 t) /\
    MUX(seq t,q3 t,l5 t,out_1 t) /\
    MUX(seq t,q4 t,l5 t,out_0 t))
```

```
SAMPLER_IMP1(sel,in,out_0,out_1,out_2,out_3) ==> SAMPLER_SPEC(sel,in,out_0,out_1,out_2,out_3)
```

D.15 A Shift Register

```
SREG4_SPEC(reset,inp,out_0,
     out_1,out_2,out_3) =
    (!t.
      (reset t =>
       ((out_0(SUC t) = F) /\
        (out_1(SUC t) = F) /\
        (out_2(SUC t) = F) /\
        (out_3(SUC t) = F)) |
       ((out_0(SUC t) = inp t) /\
        (out_1(SUC t) = out_0 t) /\
        (out_2(SUC t) = out_1 t) /\
        (out_3(SUC t) = out_2 t))))
```

```
SREG4_IMP(reset,inp,out_0,out_1,out_2,out_3) =
    (!t.
      DFF_RES(reset,inp,out_0) /\
      DFF_RES(reset,out_0,out_1) /\
      DFF_RES(reset,out_1,out_2) /\
      DFF_RES(reset,out_2,out_3))
```

```
SREG4_IMP(reset,inp,out_0,out_1,out_2,out_3) ==>
  SREG4_SPEC(reset,inp,out_0,out_1,out_2,out_3)
```

D.16 A Twisted Ring Counter

```
TRC_SPEC(reset,a,b,c) =
 (!t.
    (a(SUC t) = (reset t => F | b t)) /\
    (b(SUC t) = (reset t => F | c t)) /\
    (c(SUC t) = (reset t => F | (~a t /\
                 (~b t \/ c t)))))
```

```
TRC_IMP(reset,a,b,c) =
 (?aBar bBar cBar l1 l2.
   !t.
    DTYPE_RESET(reset,b,a,aBar) /\
    DTYPE_RESET(reset,c,b,bBar) /\
    DTYPE_RESET(reset,l1,c,cBar) /\
    AND(aBar t,l2 t,l1 t) /\
    OR(bBar t,c t,l2 t))
```

```
TRC_IMP(reset,a,b,c) ==> TRC_SPEC(reset,a,b,c)
```

```
TRC_IMP(reset,a,b,c) /\
    (reset t = F) /\
    (a t = F) /\
    (b t = T) /\
    (c t = F) ==>
    (a(SUC t) = T) /\
    (b(SUC t) = F) /\
    (c(SUC t) = F)
```

```
TRC_IMP(reset,a,b,c) /\
  (reset t = F) /\
  (reset(SUC t) = F) /\
  (a t = T) /\
  (b t = F) /\
  (c t = T) ==>
  (a(SUC(SUC t)) = T) /\
  (b(SUC(SUC t)) = F) /\
  (c(SUC(SUC t)) = F)
```

D.17 An XOR-Gate

```
XOR_IMP(in1,in2,out) =
        (?l1 l2 l3 l4.
          NOT(in2,l1) /\
          NOT(in1,l2) /\
          AND(in1,l1,l3) /\
          AND(in2,l2,l4) /\
          OR(l3,l4,out))
```

```
XOR_IMP(in1,in2,out) = XOR(in1,in2,out)
```

E Verification of a Detector

```
i80s25|THEORIES: hol                                                    14


          _  _    __    _      __    __
   |___   |__|   |  |   |     |__|  |__|
   |      |  |   |__|   |__   |__|  |__|

          Version 2.0, built on 14/10/91

#timer true;;
false : bool
Run time: 0.0s

#set_search_path ([‘~/EXAMPLES/THEORIES/‘;‘~/FAUST/‘]@(search_path()));;
() : void
Run time: 0.0s

#loadf ‘hardprover‘;;
.....................................() : void
Run time: 1.6s

#new_theory ‘detect‘;;
() : void
Run time: 0.0s
Intermediate theorems generated: 1

#new_parent‘basics‘;;
Theory basics loaded
() : void
Run time: 0.0s
```

```
#new_definition(‘DETECT110_SPEC‘,                                       15
        "!e a.DETECT110_SPEC(e,a)=
          !(t:num).a (SUC (SUC t))=~e(SUC(SUC t))/\e(SUC t)/\e t");;
|- !e a.
    DETECT110_SPEC(e,a) =
    (!t. a(SUC(SUC t)) = ~e(SUC(SUC t)) /\ e(SUC t) /\ e t)
Run time: 0.2s
Intermediate theorems generated: 2
```

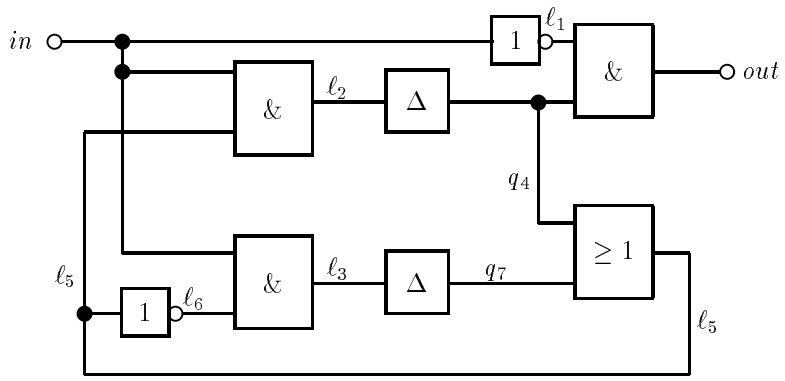


Figure 15: Implementation of a 110-detector

```
#loadt `~/EXAMPLES/HW_EXAMPLES/DETECT/detect_imp.ml`;;                    16

|- !e a.
    DETECT110_IMP(e,a) =
    (?l1 l2 l3 q4 l5 l6 q7.
      !t.
       NOT(e t,l1 t) /\
       NOT(l5 t,l6 t) /\
       AND(e t,l5 t,l2 t) /\
       AND(e t,l6 t,l3 t) /\
       DELAY(l2,q4) /\
       DELAY(l3,q7) /\
       AND(l1 t,q4 t,a t) /\
       OR(q4 t,q7 t,l5 t))
Run time: 0.3s
Intermediate theorems generated: 2


File ~/EXAMPLES/HW_EXAMPLES/DETECT/detect_imp.ml loaded
() : void
Run time: 1.1s
Intermediate theorems generated: 2
```

17

```
#let goal  = ([],"DETECT110_IMP(e,a)==>DETECT110_SPEC(e,a)");;
goal =
([], "DETECT110_IMP(e,a) ==> DETECT110_SPEC(e,a)")
: (* list # term)
Run time: 0.0s

#let goal1 = expand_all goal;;
goal1 =
([],
 "(?l1 l2 l3 q4 l5 l6 q7.
    !t.
     (l1 t = ~e t) /\
     (l6 t = ~l5 t) /\
     (l2 t = e t /\ l5 t) /\
     (l3 t = e t /\ l6 t) /\
     (!t. (q4 0 = F) /\ (q4(SUC t) = l2 t)) /\
     (!t. (q7 0 = F) /\ (q7(SUC t) = l3 t)) /\
     (a t = l1 t /\ q4 t) /\
     (l5 t = q4 t \/ q7 t)) ==>
  (!t. a(SUC(SUC t)) = ~e(SUC(SUC t)) /\ e(SUC t) /\ e t)")
: (* list # term)
Run time: 0.5s
Intermediate theorems generated: 22
```

18

```
#let goal2 = simplify goal1;;
goal2 =
([],
 "(?q4 q7.
    !t.
     ~q4 0 /\
     ~q7 0 /\
     (q4(SUC t) = e t /\ (q4 t \/ q7 t)) /\
     (q7(SUC t) = e t /\ ~(q4 t \/ q7 t)) /\
     (a t = ~e t /\ q4 t)) ==>
  (!t. a(SUC(SUC t)) = ~e(SUC(SUC t)) /\ e(SUC t) /\ e t)")
: goal
Run time: 0.1s
```

19

```
#pred_prove goal2;;
|- (?q4 q7.
     !t.
      ~q4 0 /\
      ~q7 0 /\
      (q4(SUC t) = e t /\ (q4 t \/ q7 t)) /\
      (q7(SUC t) = e t /\ ~(q4 t \/ q7 t)) /\
      (a t = ~e t /\ q4 t)) ==>
   (!t. a(SUC(SUC t)) = ~e(SUC(SUC t)) /\ e(SUC t) /\ e t)
Run time: 525.7s
Intermediate theorems generated: 1
Run time: 0.0s
```

```
#save_thm(`DETECT110_CORRECT`,(mk_thm goal));;                    20
|- DETECT110_IMP(e,a) ==> DETECT110_SPEC(e,a)
Run time: 0.1s
Intermediate theorems generated: 1

#close_theory();;
() : void
Run time: 0.1s
Intermediate theorems generated: 1
```

```
#print_theory`-`;;                                                21
The Theory detect
Parents --  HOL     basics
Constants --
  DETECT110_IMP ":(num -> bool) # (num -> bool) -> bool"
  DETECT110_SPEC ":(num -> bool) # (num -> bool) -> bool"
Definitions --
  DETECT110_IMP
    |- !e a.
        DETECT110_IMP(e,a) =
        (?l1 l2 l3 q4 l5 l6 q7.
          !t.
           NOT(e t,l1 t) /\
           NOT(l5 t,l6 t) /\
           AND(e t,l5 t,l2 t) /\
           AND(e t,l6 t,l3 t) /\
           DELAY(l2,q4) /\
           DELAY(l3,q7) /\
           AND(l1 t,q4 t,a t) /\
           OR(q4 t,q7 t,l5 t))
  DETECT110_SPEC
    |- !e a.
        DETECT110_SPEC(e,a) =
        (!t. a(SUC(SUC t)) = ~e(SUC(SUC t)) /\ e(SUC t) /\ e t)

Theorems --
  DETECT110_CORRECT  |- DETECT110_IMP(e,a) ==> DETECT110_SPEC(e,a)

******************** detect ********************
```

F A faulty circuit

In figure 16 a faulty circuit for a two bit adder ADD2, is given. The dashed AND-gate should be an OR-gate.

An attempt to verify this circuit leads to the following session:

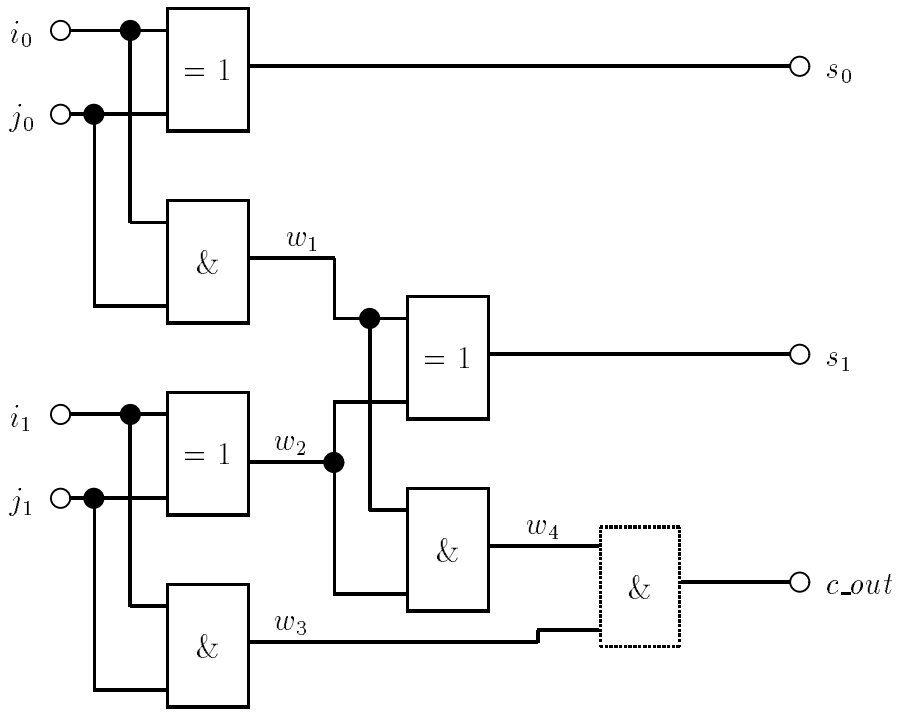


Figure 16: A wrong circuit

```
i80s12|test: hol                                                        22


          _  _    __    _      __    __
   |___   |__|   |  |   |     |__|  |__|
   |      |  |   |__|   |__   |__|  |__|

          Version 2.0, built on 14/10/91



File hol-init loaded
() : void

#loadf `hardprover_tpcd`;;
.....................................................() : void

#new_theory `add2_fault`;;
() : void

#new_parent `basics`;;
Theory basics loaded
() : void
```

23

```
# new_definition
  (‘ADD2_DEF‘,
   "ADD2 (i0,i1,j0,j1,s0,s1,c_out) =
     ((s0 = (i0 /\ ~j0) \/ (~i0 /\ j0)) /\
      (s1 = ((((i1 /\ ~j1) \/ (~i1 /\ j1)) /\ ~(i0 /\ j0)) \/
             (~((i1 /\ ~j1) \/ (~i1 /\ j1)) /\ (i0 /\ j0)))) /\
      (c_out = ((i0 /\ i1 /\ (j1 \/ j0)) \/
                ((i1 \/ i0) /\ j1 /\ j0) \/
                 (i1 /\ ~i0 /\ j1 /\ ~j0))))");;
|- !i0 i1 j0 j1 s0 s1 c_out.
    ADD2(i0,i1,j0,j1,s0,s1,c_out) =
    (s0 = i0 /\ ~j0 \/ ~i0 /\ j0) /\
    (s1 =
     (i1 /\ ~j1 \/ ~i1 /\ j1) /\ ~(i0 /\ j0) \/
     ~(i1 /\ ~j1 \/ ~i1 /\ j1) /\ i0 /\ j0) /\
    (c_out =
     i0 /\ i1 /\ (j1 \/ j0) \/
     (i1 \/ i0) /\ j1 /\ j0 \/
     i1 /\ ~i0 /\ j1 /\ ~j0)

# new_definition
  (‘ADD2_IMP_DEF‘,
   "ADD2_IMP (i0,i1,j0,j1,s0,s1,c_out) =
    ?w1 w2 w3 w4.
     XOR(i0,j0,s0) /\
     AND(i0,j0,w1) /\
     XOR(i1,j1,w2) /\
     AND(i1,j1,w3) /\
     XOR(w1,w2,s1) /\
     AND(w1,w2,w4) /\
     AND(w4,w3,c_out)");; %OR gate changed to AND%
|- !i0 i1 j0 j1 s0 s1 c_out.
    ADD2_IMP(i0,i1,j0,j1,s0,s1,c_out) =
    (?w1 w2 w3 w4.
      XOR(i0,j0,s0) /\
      AND(i0,j0,w1) /\
      XOR(i1,j1,w2) /\
      AND(i1,j1,w3) /\
      XOR(w1,w2,s1) /\
      AND(w1,w2,w4) /\
      AND(w4,w3,c_out))
```

24

```
#let goal = ([],"ADD2_IMP (i0,i1,j0,j1,s0,s1,c_out)
= ADD2 (i0,i1,j0,j1,s0,s1,c_out)");;
goal =
([],
 "ADD2_IMP(i0,i1,j0,j1,s0,s1,c_out) = ADD2(i0,i1,j0,j1,s0,s1,c_out)")
: (* list # term)
```

```
#let tg1 = expand_definitions tdefs goal;;                                    25
tg1 =
([],
 "(?w1 w2 w3 w4.
    (s0 = ~(i0 = j0)) /\
    (w1 = i0 /\ j0) /\
    (w2 = ~(i1 = j1)) /\
    (w3 = i1 /\ j1) /\
    (s1 = ~(w1 = w2)) /\
    (w4 = w1 /\ w2) /\
    (c_out = w4 /\ w3)) =
  (s0 = i0 /\ ~j0 \/ ~i0 /\ j0) /\
  (s1 =
   (i1 /\ ~j1 \/ ~i1 /\ j1) /\ ~(i0 /\ j0) \/
   ~(i1 /\ ~j1 \/ ~i1 /\ j1) /\ i0 /\ j0) /\
  (c_out =
   i0 /\ i1 /\ (j1 \/ j0) \/
   (i1 \/ i0) /\ j1 /\ j0 \/
   i1 /\ ~i0 /\ j1 /\ ~j0)")
: (* list # term)
```

```
#let tg2 = simplify tg1;;                                                    26
tg2 =
([],
 "(s0 = ~(i0 = j0)) /\
  (s1 = ~(i0 /\ j0 = ~(i1 = j1))) /\
  (c_out = ((i0 /\ j0) /\ ~(i1 = j1)) /\ i1 /\ j1) =
  (s0 = i0 /\ ~j0 \/ ~i0 /\ j0) /\
  (s1 =
   (i1 /\ ~j1 \/ ~i1 /\ j1) /\ ~(i0 /\ j0) \/
   ~(i1 /\ ~j1 \/ ~i1 /\ j1) /\ i0 /\ j0) /\
  (c_out =
   i0 /\ i1 /\ (j1 \/ j0) \/
   (i1 \/ i0) /\ j1 /\ j0 \/
   i1 /\ ~i0 /\ j1 /\ ~j0)")
: goal
```

```
#faust_prove tg2;;                                                           27
fault detected by setting:
variables set to 'T':
i1  j1
variables set to 'F':
s0  i0  j0  s1  c_out

() : void
```

FAUST has detected a counterexample for the given goal, thus it cannot be proved. Instead, one can use the countermodel generated to search for the error in the circuit. In this countermodel the variables have values corresponding to the addition of '10' and '10', such that a carry should occur. But it can be seen that `c_out` evaluates to T in the

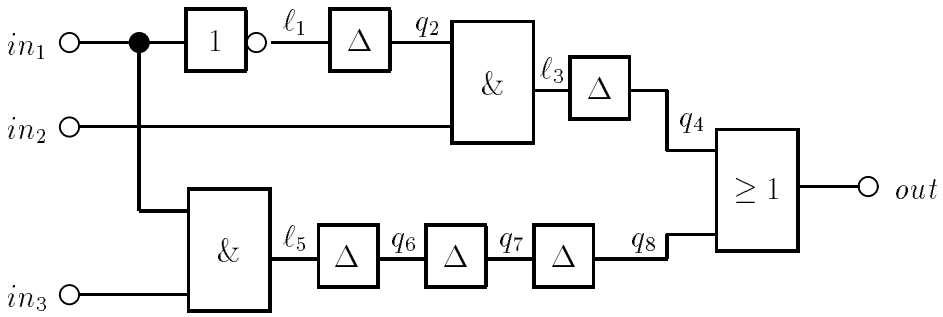


Figure 17: A asynchronous circuit with a hazard

specification and F in the implementation. Although the faulty gate cannot be identified in general, the countermodel is very useful in searching for errors.

G Hazard Detection

Our approach is not generally restricted to unit-delay timing conditions. At the moment we focus on more complex timing conditions and on asynchronous circuits. For an example, we list here the verification of a safety property, namely the presence of a hazard in a circuit. The delay times of the various circuits are modelled by separate DELAY-gates, i.e., if a circuit has a delay time which is three times larger than the delay time of another circuit, then we have to add three times more delay gates to this circuit than to the other one.

The circuit we concentrate on in this section is given in figure 17. In figure G we give impulse diagrams for the considered input signals. It can be observed that the output signal changes twice, although the input signals only change once, thus we have detected a hazard in the circuit.

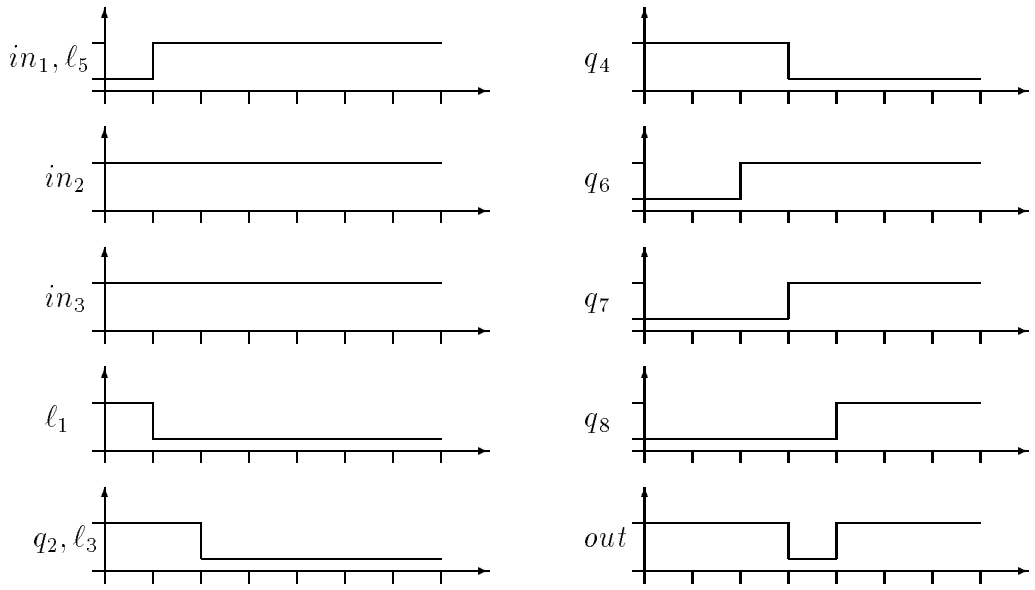


Figure 18: Impulse diagrams for detecting the hazard.


```
i80s25|THEORIES: hol


          _  _     __    _       __    __
   |___   |__|   |  |   |      |__|  |__|
   |      |  |   |__|   |__    |__|  |__|

          Version 2.0, built on 14/10/91

#timer true;;
false : bool
Run time: 0.0s

#set_search_path ([‘~/EXAMPLES/THEORIES/‘;‘~/FAUST/‘]@(search_path()));;
() : void
Run time: 0.0s

#loadf ‘hardprover‘;;
.....................................() : void
Run time: 1.6s

#new_theory ‘hazard‘;;
() : void
Run time: 0.1s
Intermediate theorems generated: 1

#new_parent ‘basics‘;;
Theory basics loaded
() : void
Run time: 0.1s
```

```
#loadt `~/EXAMPLES/HW_EXAMPLES/HAZARD/hazard_imp.ml`;;

|- !in1 in2 in3 out.
    HAZARD_IMP(in1,in2,in3,out) =
    (?l1 q2 l3 q4 l5 q6 q7 q8.
      !t.
       NOT(in1 t,l1 t) /\
       DELAY(l1,q2) /\
       AND(q2 t,in2 t,l3 t) /\
       DELAY(l3,q4) /\
       AND(in1 t,in3 t,l5 t) /\
       DELAY(l5,q6) /\
       DELAY(q6,q7) /\
       DELAY(q7,q8) /\
       OR(q4 t,q8 t,out t))
Run time: 0.4s
Intermediate theorems generated: 2


File ~/EXAMPLES/HW_EXAMPLES/HAZARD/hazard_imp.ml loaded
() : void
Run time: 0.9s
Intermediate theorems generated: 2
```



```
#let goal = ([],"HAZARD_IMP(in1,in2,in3,out) ==>
                 (in3 0 /\ in2 0 /\ ~in1 0)  /\
        ( (!t.in3 (SUC t)) /\ (!t. in2(SUC t)) /\ (!t. in1 (SUC t)) )
        ==> (?t. out t /\ ~out(SUC t) /\ out(SUC(SUC t)) )");;
goal =
([],
 "HAZARD_IMP(in1,in2,in3,out) ==>
  (in3 0 /\ in2 0 /\ ~in1 0) /\
  (!t. in3(SUC t)) /\
  (!t. in2(SUC t)) /\
  (!t. in1(SUC t)) ==>
  (?t. out t /\ ~out(SUC t) /\ out(SUC(SUC t)))")
: (* list # term)
Run time: 0.0s
```

31

```
#let goal1 = expand_all goal;;
goal1 =
([],
 "(?l1 q2 l3 q4 l5 q6 q7 q8.
    !t.
     (l1 t = ~in1 t) /\
     (!t. (q2 0 = F) /\ (q2(SUC t) = l1 t)) /\
     (l3 t = q2 t /\ in2 t) /\
     (!t. (q4 0 = F) /\ (q4(SUC t) = l3 t)) /\
     (l5 t = in1 t /\ in3 t) /\
     (!t. (q6 0 = F) /\ (q6(SUC t) = l5 t)) /\
     (!t. (q7 0 = F) /\ (q7(SUC t) = q6 t)) /\
     (!t. (q8 0 = F) /\ (q8(SUC t) = q7 t)) /\
     (out t = q4 t \/ q8 t)) ==>
  (in3 0 /\ in2 0 /\ ~in1 0) /\
  (!t. in3(SUC t)) /\
  (!t. in2(SUC t)) /\
  (!t. in1(SUC t)) ==>
  (?t. out t /\ ~out(SUC t) /\ out(SUC(SUC t)))")
: (* list # term)
Run time: 0.9s
Intermediate theorems generated: 19
```

32

```
#let goal2 = simplify goal1;;
goal2 =
([],
 "(!t.
    ~out 0 /\
    ~out(SUC 0) /\
    (out(SUC(SUC 0)) = ~in1 0 /\ in2(SUC 0)) /\
    (out(SUC(SUC(SUC t))) =
     ~in1(SUC t) /\ in2(SUC(SUC t)) \/ in1 t /\ in3 t)) ==>
  (in3 0 /\ in2 0 /\ ~in1 0) /\
  (!t. in3(SUC t)) /\
  (!t. in2(SUC t)) /\
  (!t. in1(SUC t)) ==>
  (?t. out t /\ ~out(SUC t) /\ out(SUC(SUC t)))")
: (* list # term)
Run time: 0.1s
```

```
#let goal3 = pred_prove goal2;;
goal3 =
|- (!t.
     ~out 0 /\
     ~out(SUC 0) /\
     (out(SUC(SUC 0)) = ~in1 0 /\ in2(SUC 0)) /\
     (out(SUC(SUC(SUC t))) =
      ~in1(SUC t) /\ in2(SUC(SUC t)) \/ in1 t /\ in3 t)) ==>
   (in3 0 /\ in2 0 /\ ~in1 0) /\
   (!t. in3(SUC t)) /\
   (!t. in2(SUC t)) /\
   (!t. in1(SUC t)) ==>
   (?t. out t /\ ~out(SUC t) /\ out(SUC(SUC t)))
Run time: 22.2s
Intermediate theorems generated: 1
```

```
#save_thm(`HAZARD`,(mk_thm goal));;
|- HAZARD_IMP(in1,in2,in3,out) ==>
   (in3 0 /\ in2 0 /\ ~in1 0) /\
   (!t. in3(SUC t)) /\
   (!t. in2(SUC t)) /\
   (!t. in1(SUC t)) ==>
   (?t. out t /\ ~out(SUC t) /\ out(SUC(SUC t)))
Run time: 0.2s
Intermediate theorems generated: 1

#close_theory ();;
() : void
Run time: 0.2s
Intermediate theorems generated: 1

#print_theory `hazard`;;
The Theory hazard
Parents --  HOL     basics
Constants --
  HAZARD_IMP
    ":(num -> bool) # ((num -> bool) # ((num -> bool) # (num -> bool))) ->
      bool"
Definitions --
  HAZARD_IMP
    |- !in1 in2 in3 out.
        HAZARD_IMP(in1,in2,in3,out) =
        (?l1 q2 l3 q4 l5 q6 q7 q8.
          !t.
           NOT(in1 t,l1 t) /\
           DELAY(l1,q2) /\
           AND(q2 t,in2 t,l3 t) /\
           DELAY(l3,q4) /\
           AND(in1 t,in3 t,l5 t) /\
           DELAY(l5,q6) /\
           DELAY(q6,q7) /\
           DELAY(q7,q8) /\
           OR(q4 t,q8 t,out t))

Theorems --
  HAZARD
    |- HAZARD_IMP(in1,in2,in3,out) ==>
       (in3 0 /\ in2 0 /\ ~in1 0) /\
       (!t. in3(SUC t)) /\
       (!t. in2(SUC t)) /\
       (!t. in1(SUC t)) ==>
       (?t. out t /\ ~out(SUC t) /\ out(SUC(SUC t)))

******************** hazard ********************
```

List of Figures

List of Tables