Linking higher order logic to binary decision diagrams

Mike Gordon

Theorem proving and model checking are complementary. Theorem proving can be applied to expressive logics (e.g. set theory and higher order logic) capable of modelling complex systems like complete processors. However, theorem proving systems require skilled manual guidance to verify most properties of practical interest. Model checking is automatic, but can only be applied to relatively inexpressive (e.g. propositional) logics and to small problems (e.g. fragments of processors). It can also provide counter-examples of great use in debugging.

The ideal would be to be able to automatically verify properties of complete systems (and find counter-examples when the verification of properties fails). This is not likely to be practical in the foreseeable future, so various compromises are being explored, for example

1. adding a layer of theorem proving on top of existing model checkers to enable large problems to be deductively decomposed into smaller pieces that can be checked automatically [8, 1];
2. adding checking algorithms to theorem provers so that subgoals can be verified automatically [13] and counter-examples found.

This paper describes some ideas behind an experimental system for exploring combinations of deduction and symbolic calculation based on adding support for binary decision diagrams (BDDs) to the HOL proof assistant. It is thus an example of the second approach.

The HOL theorem prover is based on Milner's LCF proof assistant [5] which introduced the idea of encapsulating a logic with an abstract type of theorems whose primitive operations are the axioms and rules of inference of the logic.[1] Theorem proving tools such as decision procedures, proof search strategies and simplifiers are implemented by composing together the primitive inference rules using programs in the ML programming language.

Theorem provers that expand all proof steps down to sequences of primitive inferences are called *fully-expansive*. There is a long-standing controversy concerning

[1] Historical note: initially the encapsulation of theorems was implemented with an ad hoc mechanism. Subsequently Hoare pointed out to Milner that this mechanism could be seen as just an abstract type, and this was one of the motivations for adding abstract types to ML.

whether such systems can achieve good enough efficiency. In a surprising number of cases adequate efficiency can be achieved [2]. However, programming decision procedures and theorem provers in a fully-expansive style is technically demanding and thus when a high assurance of soundness is not required it might be more cost-effective to use a non fully-expansive approach. For this (and other) reasons, some modern LCF-style provers (e.g. HOL and Isabelle [10]) allow 'oracles' to create theorems. Such oracles can either be ML programs that operate directly on the datastructures representing terms, or they might be external tools implemented in other languages (e.g. C). As part of research on architectures for tool integration platforms, a 'plug-in' mechanism for HOL has been implemented [11]. This enables the easy implementation of the kind of linking of theorem proving and model checking done in pioneering studies with PVS [13].

Many verification algorithms, including symbolic model checking [9], represent Boolean formulae as reduced ordered binary decision diagrams (ROBDDs or BDDs for short) [4]. The work described here differs from the plug-in approach in that it aims to provide general infrastructure enabling users to implement their own bespoke BDD-based verification algorithms and then to tightly integrate them with existing HOL tools like the simplifier. Currently, a plug-in is loosely coupled: it supports a 'black box' style of integration in which HOL passes a problem to an external tool down the plug-in interface and the tool then passes back a result via the interface. Boyer and Moore [3] have argued that decision procedures need to be tightly integrated with other tools (e.g. simplifiers and normalisers). The same point has been reiterated by the designers of PVS [12], which is built around a powerful suite of cooperating decision procedures.

It is not claimed that tight-integration is always the best way to connect external tools: for some applications the plug-in approach is appropriate. However, it seems that there may be interesting synergy between deduction and BDD-based calculation that are hard to realise with a loose connection (see Section 3).

Perhaps the experience gained from the approach described here and from the use of plug-ins will inform the design of a second generation tool integration platform supporting a spectrum from loose to tight integration of external tools.

1 Review of HOL

The HOL system provides an ML datatype `term` representing terms of higher order logic. Terms can be entered as quotations of the form ``` ``...`` ``` via a quotation parser. For example ``` ``∀v. (v = T) ∨ (v = F)`` ``` parses to the value of type `term` that expresses the Law of Excluded Middle. The type `term` comes equipped with various constructors and destructors, for example `dest_imp` splits an implication $t_1 \Rightarrow t_2$ into the ML pair (t_1, t_2), `dest_comb` splits a combination (function application) $t_1\ t_2$ into the ML pair (t_1, t_2) and `dest_abs` splits a λ-abstraction $\lambda v.\ t$ into the ML pair (v, t). The ML function `subst` takes a list of term-variable pairs and a term t and substitutes each term for the corresponding variable in t (renaming bound variables to avoid capture, if necessary).

In HOL (following LCF) the axioms and rules of inference are encapsulated

in an ML abstract type `thm`. For example, consider the axiom (`BOOL_CASES_AX`), axiom scheme (`BETA_CONV`) and rule of inference (`MP`).

ML name	*Informal description*	*Formal description*
`BOOL_CASES_AX`	Excluded Middle:	$\dfrac{}{\vdash \forall v.\ (v = \mathtt{T}) \vee (v = \mathtt{F})}$
`BETA_CONV`	β-conversion:	$\dfrac{}{\vdash (\lambda v.\ t)\ t' \ =\ t[t' \rightarrow v]}$
`MP`	Modus Ponens:	$\dfrac{\vdash t \Rightarrow t' \qquad \vdash t}{\vdash t'}$

The definition of type `thm` is illustrated below using simplified ML code to implement these three axioms and rules.

```
exception error;

abstype  thm = mk_thm of term list * term  with

 fun dest_thm(mk_thm(al,t)) = (al,t)

 val BOOL_CASES_AX = mk_thm([],``∀v. (v = T) ∨ (v = F)``)

 fun BETA_CONV t =
  let val (t1,t2)  = dest_comb t
      val (bv,bdy) = dest_abs t1
  in mk_thm([],mk_eq(t,subst[(t2,bv)]bdy)) end

 fun MP (mk_thm(al,t)) (mk_thm(al',t')) =
  let val (t1,t2) = dest_imp t
  in if t1=t' then mk_thm(union al al',t2) else raise error end

   ⋮

 end
```

The key point is that although the theorem destructor `dest_thm` is exported, the theorem constructor `mk_thm` is not exported. The only way of constructing theorems is by applying ML functions corresponding to logical inference rules (e.g. `MP`) to axioms (e.g. `BOOL_CASES_AX`) or instances of axiom schems (e.g. results of `BETA_CONV`) or previously computed theorems.

2 Binary decision diagrams

Binary decision diagrams (BDDs) are a data-structure for representing formulae (i.e. Boolean terms in HOL) that has found many uses in formal verification [4]. The construction of the BDD representing a term t requires the variables in t to be ordered. Given such an ordering, the BDD of t is unique. BuDDy is a state-of-the-art BDD package implemented in C by Jørn Lind-Nielsen. BuDDy provides

C functions for efficiently building and manipulating BDDs. MuDDy, due to Ken Friis Larsen, is a package that provides access to BuDDy via ML functions in Moscow ML. MuDDy makes BuDDy BDDs appear in ML as values of type `bdd`.

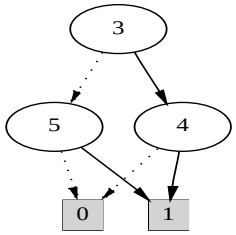
BuDDy uses non-negative integers to represent variables (ordered according to the usual arithmetical order). The MuDDy function `ithvar` of ML type `int->bdd`, maps a number to the corresponding BuDDy BDD variable. Besides the variables, there are two atomic BDDs `TRUE` and `FALSE` of ML type `bdd`. Compound BDDs are constructed using standard operations. For example, the function `NOT` of type `bdd->bdd` negates a BDD and the binary operations `AND`, `OR`, `IMP` and `BIIMP`, all of type `bdd*bdd->bdd`, combine two BDDs to yield the conjunction, disjunction, implication and equivalence (bi-implication) between BDDs.

BuDDy provides a wide repertoire of BDD operations, most of which are available in ML via MuDDy. There are standard algorithms for efficiently computing some combinations of operations. For example, the BDD representing the existential quantification of a conjunction by a number of variables can be efficiently computed in one step from the list of variables and the two conjuncts.

3 Formal link from HOL terms to BDDs

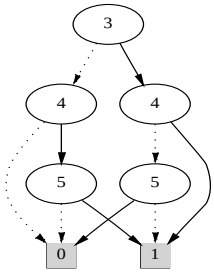
Theorems are one kind of judgement, represented in ML by values of type `thm`. Consider now another kind of judgement: $\rho\ t \mapsto b$ meaning that the HOL Boolean term t is represented by the BuDDy BDD b with respect to a mapping ρ from HOL variables to ML integers (representing BDD variables). Two examples of judgements $\rho\ t \mapsto b$ with the same t, but different ρs are

$\{a \mapsto 3, b \mapsto 4, c \mapsto 5\}$ $\quad (a \wedge b) \vee (\neg a \wedge c) \ \mapsto$



and

$\{a \mapsto 5, b \mapsto 4, c \mapsto 3\}$ $\quad (a \wedge b) \vee (\neg a \wedge c) \ \mapsto$



In the first example the variables are ordered $a < b < c$, but in the second example they are ordered $c < b < a$. Notice that the BDDs are different.

3.1 Linking rules

The valid judgements $\rho\ t \mapsto b$ can be specified by axioms, axiom schemes and inference rules similar to way theorems are defined.

The scheme that relates HOL logical variables to BuDDy variable nodes is

$$\frac{\rho(v) = n}{\rho\ v\ \mapsto\ \texttt{ithvar}\ n}$$

The HOL logical constants `T` and `F` denote truth and falsity, respectively, whereas the atomic MuDDy values `TRUE` and `FALSE` of ML type `bdd` are the corresponding BDDs. The axioms and rules that follow express the logical semantics of MuDDy (and hence BuDDy) in terms of HOL.

$$\frac{}{\rho\ \texttt{T} \mapsto \texttt{TRUE}} \qquad \frac{}{\rho\ \texttt{F} \mapsto \texttt{FALSE}} \qquad \frac{\rho\ t \mapsto b}{\rho\ \neg t\ \mapsto\ \texttt{NOT}\ b}$$

$$\frac{\rho\ t_1 \mapsto b_1 \qquad \rho\ t_2 \mapsto b_2}{\rho\ t_1 \wedge t_2\ \mapsto\ b_1\ \texttt{AND}\ b_2} \qquad \frac{\rho\ t_1 \mapsto b_1 \qquad \rho\ t_2 \mapsto b_2}{\rho\ t_1 \vee t_2\ \mapsto\ b_1\ \texttt{OR}\ b_2}$$

$$\frac{\rho\ t_1 \mapsto b_1 \qquad \rho\ t_2 \mapsto b_2}{\rho\ t_1 \Rightarrow t_2\ \mapsto\ b_1\ \texttt{IMP}\ b_2} \qquad \frac{\rho\ t_1 \mapsto b_1 \qquad \rho\ t_2 \mapsto b_2}{\rho\ t_1 = t_2\ \mapsto\ b_1\ \texttt{BIIMP}\ b_2}$$

The rules for the quantifiers relate the MuDDy BDD-building operations (`forall`, `exist`) to logical quantification. These operations can quantify several variables at once. MuDDy provides a special ML type `varSet` to represent sets of BDD variables and a constructor `makeset` of type `(int)list ->varSet`.

$$\frac{\rho\ t \mapsto b \qquad \rho(v_1) = n_1 \qquad \cdots \qquad \rho(v_p) = n_p}{\rho\ \forall v_1\ \cdots\ v_p.\ t\ \mapsto\ \texttt{forall}\ (\texttt{makeset}[n_1,\ldots,n_p])\ b}$$

$$\frac{\rho\ t \mapsto b \qquad \rho(v_1) = n_1 \qquad \cdots \qquad \rho(v_p) = n_p}{\rho\ \exists v_1\ \cdots\ v_p.\ t\ \mapsto\ \texttt{exist}\ (\texttt{makeset}[n_1,\ldots,n_p])\ b}$$

The BDDs of quantifications of conjunctions can be built by calling `AND` followed by `forall`, but it is more efficient to use the optimised algorithms `appall` and `appex` provided by BuDDy.

$$\frac{\rho\ t_1 \mapsto b_1 \qquad \rho\ t_2 \mapsto b_2 \qquad \rho(v_1) = n_1 \qquad \cdots \qquad \rho(v_p) = n_p}{\rho\ \forall v_1\ \cdots\ v_p.\ t_1 \wedge t_2\ \mapsto\ \texttt{appall}\ b_1\ b_2\ \texttt{And}\ (\texttt{makeset}[n_1,\ldots,n_p])}$$

$$\frac{\rho\ t_1 \mapsto b_1 \qquad \rho\ t_2 \mapsto b_2 \qquad \rho(v_1) = n_1 \qquad \cdots \qquad \rho(v_p) = n_p}{\rho\ \exists v_1\ \cdots\ v_p.\ t_1 \wedge t_2\ \mapsto\ \texttt{appex}\ b_1\ b_2\ \texttt{And}\ (\texttt{makeset}[n_1,\ldots,n_p])}$$

The *only* rule for deriving HOL theorems from BuDDy is

$$\texttt{bddOracle} \qquad \frac{\rho\ t \mapsto \texttt{TRUE}}{\vdash t}$$

A rule for transferring BDD representation links across HOL equalities is

$$\texttt{addEquation} \qquad \frac{\vdash t_1 = t_2 \qquad \rho\ t_2 \mapsto b}{\rho\ t_1 \mapsto b}$$

If t is a quantified Boolean formula[2] and ρ maps all the variables in t to distinct

[2] A quantified Boolean formula (QBF) is a formula built out of Boolean variables and constants using Boolean operations and quantification over Boolean variables.

numbers, then it is clear that the rules above enable a (necessarily unique) BDD b to be deduced such that $\rho\ t\ \mapsto\ b$.

An experimental system has been implemented that manages the linking of HOL terms to BDDs. Logically, this just provides LCF-like fully-expansive support for a formal system consisting of the union of higher order logic with a calculus of judgements $\rho\ t \mapsto b$. The only rule for converting BDD judgements into logic theorems is `addEquation`, so there is a clean interface from the BDD world to the HOL world. This interface is carefully implemented to minimise the danger of false theorems being generated.

The implementation of BDDs in HOL provides a function `termToBdd` that takes a term t and tries to compute a BDD b such that $\rho\ t \mapsto b$, using a ρ that is either explicitly supplied by the user or derived from the order in which variables are encountered. Logically `termToBdd` can be thought of as constructing a proof using linking rules like those above. The actual implementation tries to be quite efficient and to use optimised BDD algorithms (like `appall` and `appex`) whenever possible. BDD packages like BuDDy use a cache to remember the results of BDD operations. Similarly, HOL caches BDDs linked to terms so that if t subsequently occurs as part of a term whose BDD is being computed then b can be used. If the BDD of a term $f(t_1, \ldots, t_n)$ is being computed then the implementation of `termToBdd` looks to see if a BDD for $f(v_1, \ldots, v_n)$ has been cached (where $v_1, \ldots, v_n$ are variables) and if so tries to use the more general BDD to create a suitable instance with BuDDy's replacement and substitution algorithms.

The term t in any judgement $\rho\ t \mapsto b$ that can be derived will have Boolean type and will only contain free variables of Boolean type. However t need not be a quantified Boolean formula, because it might contain non-Boolean subterms (e.g. quantifications over non-Boolean variables). This is the case for model checking, where Boolean formulae involving quantifications over infinite traces arise.

3.2 Example: model checking using deduction and BDD rules

Model checking consists of algorithmically checking that all executions of a model of a system satisfy a property. Checkable properties are often expressed in temporal logic and models are specified in application-specific languages. The standard semantics of temporal logic defines the truth-value of temporal formulae with respect to sequences of states—called traces—that represent successive states of a system. Thus a temporal formula can be considered to be a predicate on sequences of states, that is a formula of the form $\Phi(\sigma)$, where Φ is the property and σ is a variable ranging over sequences of states (i.e. over traces). A model defines a set of sequences of states corresponding to possible executions of a system. Thus a model can also be represented as a predicate on traces, $\mathcal{M}(\sigma)$ say. To check that property Φ holds of all executions of model $\mathcal{M}$ it is sufficient to check the truth of the formula $\forall\sigma.\ \mathcal{M}(\sigma) \Rightarrow \Phi(\sigma)$.

Consider the special case in which the property Φ is `AG` P of CTL [9] (i.e. P true at all points in the trace) and $\mathcal{M}$ is given by a transition system $(\mathcal{R}, \mathcal{B})$, where $\mathcal{B}$ is a unary predicate defining an initial set of states and $\mathcal{R}$ is a binary transition relation.

Define

$$\begin{array}{lcl} \mathcal{M}_{(\mathcal{R},\mathcal{B})}(\sigma) & = & \mathcal{B}(\sigma(0)) \ \wedge \ \forall n.\ \mathcal{R}(\sigma(n),\sigma(n{+}1)) \\ (\mathtt{AG}\ P)(\sigma) & = & \forall n.\ P(\sigma(n)) \end{array}$$

Then proving $\forall\sigma.\ \mathcal{M}_{(\mathcal{R},\mathcal{B})}(\sigma) \Rightarrow (\mathtt{AG}\ P)(\sigma)$ is an example of a model checking problem: P holds globally (AG) of all executions of $\mathcal{M}_{(\mathcal{R},\mathcal{B})}$. We sketch how the problem can be solved in HOL.

First, inductively define $\mathcal{S}_n$ to be a predicate that is true of a state s if and only if s is reachable from a state satisfying $\mathcal{B}$ in n or fewer $\mathcal{R}$-steps.

$$\begin{array}{lcl} \mathcal{S}_0(s) & = & \mathcal{B}(s) \\ \mathcal{S}_{n+1}(s) & = & \mathcal{S}_n(s) \ \vee \ (\exists u.\ \mathcal{S}_n(u) \ \wedge \ \mathcal{R}(u,s)) \end{array}$$

The formula $\exists n.\ \mathcal{S}_n(s)$ true if s is reachable from $\mathcal{B}$ via some number of $\mathcal{R}$-steps. If $\mathcal{M}_{(\mathcal{R},\mathcal{B})}(\sigma)$, then by induction on n $\vdash \forall\sigma.\ \mathcal{M}_{(\mathcal{R},\mathcal{B})}(\sigma) \Rightarrow \forall n.\ \mathcal{S}_n(\sigma(n))$. Now suppose we can show for all n and s that $\mathcal{S}_n(s) \Rightarrow P(s)$ then, in particular, it would follow that $\forall\sigma\ n.\ \mathcal{S}_n(\sigma(n)) \Rightarrow P(\sigma(n))$ and hence $\forall\sigma.\ \mathcal{M}_{(\mathcal{R},\mathcal{B})}(\sigma) \Rightarrow \forall n.\ P(\sigma(n))$ which is $\forall\sigma.\ \mathcal{M}_{(\mathcal{R},\mathcal{B})}(\sigma) \Rightarrow (\mathtt{AG}\ P)(\sigma)$.

This shows that the model checking problem $\forall\sigma.\ \mathcal{M}_{(\mathcal{R},\mathcal{B})}(\sigma) \Rightarrow (\mathtt{AG}\ P)(\sigma)$ can be reduced to the state exploration problem $\forall s\ n.\ \mathcal{S}_n(s) \Rightarrow P(s)$. Such a reduction is easily done by automated deduction in HOL.

To solve the state exploration problem $\forall s\ n.\ \mathcal{S}_n(s) \Rightarrow P(s)$, note the fixed-point lemma $\vdash (\mathcal{S}_i(s) = \mathcal{S}_{i+1}(s)) \Rightarrow ((\exists n.\ \mathcal{S}_n(s)) = \mathcal{S}_i(s))$. This is true, because the sets corresponding to $\mathcal{S}_i$ increase as i increases.

The outline deduction below shows how a mixture of BuDDy BDD calculation and HOL proof can be used to deduce $\forall\sigma.\ \mathcal{M}_{(\mathcal{R},\mathcal{B})}(\sigma) \Rightarrow (\mathtt{AG}\ P)(\sigma)$.

Judgement	*Justification*
$\rho\ \mathcal{S}_0(s) \mapsto b_0$	`termToBdd`
$\vdots$	$\vdots$
$\rho\ \mathcal{S}_{20}(s) \mapsto b_{20}$	`termToBdd`
$\rho\ \mathcal{S}_{21}(s) \mapsto b_{21}$	`termToBdd`
$\rho\ (\mathcal{S}_{20}(s) = \mathcal{S}_{21}(s)) \mapsto b_{20}$ `BIIMP` b_{21}	rule for `BIIMP`
$\vdash \mathcal{S}_{20}(s) = \mathcal{S}_{21}(s)$	`bddOracle` (if b_{20} `BIIMP` b_{21} is `TRUE`)
$\vdash (\mathcal{S}_{20}(s) = \mathcal{S}_{21}(s)) \Rightarrow (\exists n.\ \mathcal{S}_n(s)) = \mathcal{S}_{20}(s)$	fixed-point lemma ($i = 20$)
$\vdash (\exists n.\ \mathcal{S}_n(s)) = \mathcal{S}_{20}(s)$	Modus Ponens
$\rho\ (\exists n.\ \mathcal{S}_n(s)) \mapsto b_{20}$	`addEquation`
$\rho\ P(s) \mapsto b_P$	`termToBdd`
$\rho\ (\exists n.\ \mathcal{S}_n(s)) \Rightarrow P(s) \mapsto b_{20}$ `IMP` b_P	rule for `IMP`
$\vdash (\exists n.\ \mathcal{S}_n(s)) \Rightarrow P(s)$	`bddOracle` (if b_{20} `IMP` b_P is `TRUE`)
$\vdash \forall n.\ \mathcal{S}_n(s) \Rightarrow P(s)$	from previous line (by HOL)
$\vdash \forall\sigma.\ \mathcal{M}_{(\mathcal{R},\mathcal{B})}(\sigma) \Rightarrow (\mathtt{AG}\ P)(\sigma)$	by argument given above

Such combinations of deduction and BDD calculation illustrate how symbolic model checking can be implemented in HOL.

4 Conclusions

The ideas described here are intended to provide a platform for experiments in combining deduction and symbolic calculation, such as model checking. The aim is to provide users with a clean logical view based on higher order logic that supports efficient BDD calculations, as provided by BuDDy via MuDDy. Experiments so far [6, 7] suggest that state-of-the-art performance via HOL is attainable and, furthermore, that there are interesting possibilities for combining deduction and symbolic calculation. For example, it is possible to use rewriting in HOL to modify Boolean formulae before converting them to BDDs. This provides a simple high level method of performing optimisations like early quantification [9, page 45] (or disjunctive partitioning) that avoids the need for tricky BDD programming as well as ensuring soundness. Another application is a 'reachable states viewer' that computes the BDD of the set of reachable states of a machine via a fixed-point calculation and then presents the set to the user by converting its BDD to a deductively simplified HOL term. Such a viewer appears to provide a powerful tool for state machines analysis [6].

5 Acknowledgements

The implementation of the interface between Moscow ML and the BuDDy BDD package was done by Ken Friis Larsen with the support of EPSRC grant GR/K57343 entitled *Checking equivalence between synthesized logic and non-synthesizable behavioural prototypes*. Ken Friis Larsen also implemented a prototype HOL oracle that was the starting point for the work reported here. Special thanks go to Jørn Lind-Nielsen for making his BuDDy code freely available. This work was also supported by ESPRIT Framework IV LTR 26241 project entitled *Proof and Specification Assisted Design Environments*.

John Herbert, Joe Hurd, Ken Friis Larsen and Michael Norrish provided valuable comments on a first draft of this paper.

References

[1] Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Lifted-FL: A Pragmatic Implementation of Combined Model Checking and Theorem Proving. In *Theorem Proving in Higher Order Logics (TPHOLs99)*, number 1690 in Lecture Notes in Computer Science, pages 323–340. Springer-Verlag, 1999.

[2] R. J. Boulton. *Efficiency in a Fully-Expansive Theorem Prover*. PhD thesis, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, U.K., May 1994. Technical Report 337.

[3] R. S. Boyer and J Moore. Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic. In *Machine Intelligence 11*, pages 83–124. Oxford University Press, 1988.

[4] Randall E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[5] M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[6] Mike Gordon. Programming combinations of deduction and BDD-based symbolic calculation. Technical Report 480, University of Cambridge Computer Laboratory, December 1999.

[7] Mike Gordon and Ken Friis Larsen. Combining the Hol98 proof assistant with the BuDDy BDD package. Technical Report 481, University of Cambridge Computer Laboratory, December 1999.

[8] K. L. McMillan. A methodology for hardware verification using compositional model checking. Technical report, Cadence Berkeley Labs, April 1999. Available at `http://www-cad.eecs.berkeley.edu/~kenmcmil/`.

[9] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[10] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[11] See web page `http://www.dcs.gla.ac.uk/prosper/`.

[12] See web page `http://www.csl.sri.com/pvs.html`.

[13] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, June 1995. Springer-Verlag.