Journal of Computer Science 3 (10): 810-817, 2007
ISSN 1549-3636

On The Integration of Decision Diagrams in High Order Logic Based Theorem Provers: a Survey

Sa'ed Abed, Otmane Ait Mohamed and Ghiath Al Sammane
Department of Electrical and Computer Engineering,
Concordia University, Montréal, Canada

Abstract: This survey discuss approaches that integrate Decision Diagrams inside High Order Logic based Theorem provers. The approaches can be divided in two kinds, one is based on building a translation between model checker and theorem prover, the second is based on embedding the model checker algorithms inside the theorem prover. A comparison between both is discussed in detail. The paper also tries to answer which is the best decision graphs formalization for theorem provers as what is the optimized set of operations to efficiently manipulate the decision graphs inside theorem provers. Then, we contrast between them according to their efficiency, complexity and feasibility.

Key words: Model checking, theorem proving, hybrid approach, deep embedding, logical representation, graphical representation

INTRODUCTION

State exploration method [1] (mainly model checking and equivalence checking) and Deductive verification (theorem proving) are two complementary approaches to formal verification of digital systems. In state exploration method, the design being verified is represented as a decision diagram and techniques such as reachability analysis are used to automatically verify that the properties of the design are satisfied against the model represented as a finite-state system. Thus, the verification of properties for finite-state systems is decidable. Much of this work is based on Binary Decision Diagrams [2]. Model checking is fully automatic and can also provide counter-examples when the verification of properties fails but suffers from the so-called state space explosion problem when dealing with complex systems.

In deductive theorem proving method, the correctness of a design is explored as a theorem in a mathematical logic and the proof of the theorem is checked using a general-purpose theorem-prover. Deductive theorem proving is a scalable technique that can handle complex designs and reason formally about unbounded data structures that make systems infinite-state. It provides relatively complete proof systems but requires skilled manual guidance for verification and human insight for debugging. Unfortunately, if the property fails to hold, deductive methods do not give counter-example.

There has been a great deal of work over the past decade to combine the two approaches to gain the strengths of both, and alleviate the weaknesses. A common approach is to use a state exploration method as an oracle which makes the proof and returns answer to the theorem prover. Successful combinations of this kind have been achieved in [3-9]. The strengths and weaknesses of model checking and deductive theorem proving, as discussed above, are summarized in (Table.1).

Table 1: Deductive theorem proving vs. state exploration method

	Deductive	State exploration	Hybrid
Automation	interactive	completely automatic	semi-automatic
Domain size	infinite system (complex)	finite system (large)	finite system (very large)
Debugging	expert based	generates counter-example	rarely generates counter-example

This survey discuss approaches that integrate Decision Diagrams inside High Order Logic based Theorem provers. The approaches can be divided in two kinds:

1. Hybrid approach: adding a layer of deduction theorems and rules on top of Decision

Corresponding Author: Sa'ed Abed, Department of Electrical and Computer Engineering, Concordia University, Montreal, Canada, Tel: +1(514)848-2424

Diagrams tool, combining theorem provers with other powerful model checking tool.
2. Deep embedding approach: adding Decision Diagrams algorithms to theorem provers.

A comparison between both is discussed in detail. The paper also tries to answer which is the best decision graphs formalization for theorem provers as what is the optimized set of operations to efficiently manipulate the decision graphs inside theorem provers. Then, we contrast between them according to their efficiency, complexity and feasibility.

FORMAL VERIFICATION TECHNIQUES

Formal verification problem consists of mathematically establishing that an implementation behaves according to a given set of requirements or specification. To classify the various approaches, we first look at three main aspects of the verification process: system under investigation (implementation) set of requirements to obey (specification) and formal verification tool to verify the process (relationship between implementation and specification).

Implementation refers to the description of the design that is to be verified. It can be described at different levels of abstraction which results in different verification methods. Another important issue with the implementation is the class of the system or circuit to be verified, i.e., whether it is combinational/sequential, synchronous/asynchronous, pipelined or parameterized hardware. These variations may require different approaches. Specification refers to the property with respect to which the correctness is to be determined. In practice, one needs to model both the implementation and the specification in the logic of the tool, and then uses tool to check the correctness of the system or in some cases give a trace of error (counter-example).

Formal verification approaches can be generally divided into two main categories: theorem proving methods and state exploration methods such as model checkers.

Theorem Proving is an approach where the specification and the implementation are usually expressed in first-order or higher-order logic. Their relationship is formed as a theorem to be proved within the logic system. The logic is formalized by a set of axioms and a set of inference rules. Steps in the proof appeal to the axioms and rules, and possibly derived definitions and intermediate lemmas. The axioms are usually "elementary" in the sense that they capture the basic properties of the logic's operators [10].

Theorem proving utilizes the proof inference technique. The problem itself is transformed into a sequent, a working representation for the theorem proving problem. Then a sequent holds if the formula f holds in any model:

$$\models f$$

A proof system is collection of *inference rules* of the form:

$$(name)\frac{P_1 \cdots P_n}{C}$$

Where C is a conclusion sequent, and P_i's are premises sequent. An inference rule means that if all the premises are derivable then the conclusion is guaranteed to hold.

Theorem proving methods have been in use in hardware and software verification for a number of years in various research projects. Some of the well-known theorem provers are HOL (Higher-Order Logic), ISABELLE, PVS (Prototype Verification System) and Coq [10-13]. These systems are distinguished by, among other aspects, the underlying mathematical logic, the way automatic decision procedures are integrated into the system, and the user interface. The advantage of the deductive verification approach is that it can handle very complex systems because the logics of theorem provers are more expressive. Even though they are powerful, they require expertise in using a theorem prover. User is expected to know the whole design leading to a white box verification approach. It is not fully automated and requires a large amount of time to verify the system. Another shortcoming is the inability to produce counter-examples in the event of a failed proof, because the user does not know whether the required property is not derivable or whether the person conducting the derivation is not experienced enough with the theorem prover logic.

State Exploration Methods use states space traversal algorithms on finite-state models to check if the implementation satisfies its specification. They are focused mostly on automatic decision procedures for solving the verification problem. Model checking is a state exploration based verification technique developed in the 1980s by Clarke and Emerson [14] and independently by Quielle and Sifakis [15]. In model checking, a state of the system under consideration is a snapshot of the system at certain time, given by the set of the variables values of that system at that time. The system is then modeled as a set of states together with a set of transitions between states that describe how the

system moves from one state to another in response to internal or external stimulus. Model checking tools are then used to verify that desired properties (expressed in some temporal logic) hold in the system. The state exploration approach has two important advantages. First, once the correct design of the system and the required properties has been fed in, the verification process is fully automatic. Second, in the event of a property not holding, the verification process is able to produce a counter-example (i.e. an instance of the behaviour of the system that violates the property) which is extremely useful in helping the human designers pinpoint and fix the flaw.

Model checkers such as SPIN [16], COSPAN[17], SMV[18], and Multiway Decision Graphs (MDGs)[19] take as input, essentially, a finite-state system and temporal property in some variety or subset of CTL*, and automatically check that the system satisfies the property. Moreover, the model is often restricted to a finite-state transition system, for which finite-state model checking is known to be decidable. The design or model is formalized in terms of a state machine (*Transition System*), or a Kripke structure:

$$M = (\mathrm{P}, \mathrm{S}, \mathrm{I}, \mathrm{R}, \mathrm{L})$$

Where M is a state machine (model) with a transition to describe the circuit behavior, P is a set of atomic propositions, S is a finite set of states, $I \subseteq S$ is a set of initial states, $R \subseteq SxS$ is a transition relation that must be total (i.e. for every $s \in S$ there exists $s' \in S$ such that ($s' \in R$)), and $L : S \to 2^P$ maps each state to the set of atomic propositions that are true in that state. The property ϕ is formalized as a logical formula that the machine should satisfy. The verification problem is stated as checking the formula ϕ in the model M :

$$M \models \phi$$

If the model M is represented explicitly as a transition relation, then the size of the model is limited to the number of states that can be stored in the computer memory. To increase the size of the model, more efficient state representations can be used:

1. Graphical representation: As Directed Acyclic Graphs (DAGs), trees and graphs, to save storage space and computation time by eliminating redundancies such as Binary Decision Diagrams (BDDs) and MDGs.
2. Logical representation: As terms and formulae using logic to simplify formulae and benefit of the power of automatic reasoning such as Conjunctive Normal Form (CNF).

Model checking is then done by manipulating these formulae using BDDs or SAT solving techniques.

Representing BBDs in High Order Logic: BDDs [2] are data structure used as a compact representation for Boolean functions which improves the capacity of model checkers. Different representations of ROBDDs (Reduced Order Binary Decision Diagrams) [20] are used to manipulate the state transition relations as diagrams and this allows model checkers to verify larger systems. Still, most model checkers face the state space explosion problems [14]. To be able to apply model checking to larger designs, state reduction techniques are used. Examples include partitioned transition relation, dynamic variable reordering, cone of influence reduction, abstraction, problem-specific techniques, e.g. when the original design is rewritten in a simpler way, omitting the irrelevant details, but preserving the important behavior for the property being verified.

An alternative for decision graphs is to represent the transition relation by a CNF (Conjunctive Normal Form), then to use SAT [21, 22] solvers to decide on the satisfiability of these formulae. SAT solvers can decide very large Boolean formulae in reasonable time, but they are not canonical and require additional efforts to check for the equivalence of formulas. As a result, various researchers have developed routines for performing Bounded Model Checking (BMC) [23, 24] using SAT. The common theme is to convert the problem of interest into a SAT problem, by devising the appropriate propositional Boolean formula. Then, they exploit the known ability of SAT solvers to find a single satisfying solution when it exists. Moreover, SAT solver technology has improved significantly in recent years with a number of sophisticated packages which are freely available. Well known state-of-the-art SAT solvers include CHAFF [25], GRASP [26] and SATO [27]. Since state sets can be represented as Boolean formulae, and since most model checking techniques manipulate state sets, SAT solvers have enormously boosted their speed and applicability.

In high order theorem provers, transition relations are formalized either as DAGs or as terms and formulae. The first is a graphical representation using trees and graphs, while the later is a formal logic representation using datatype.

First of all, the graph is represented as a data structure in the theorem prover. This representation should reflect the abstract properties of graphs and should be flexible to be suitable for different domains and for many applications to model complex designs.

Several examples can be cited: to model communication networks (railway track network [28]), also in transport industry, the problem of finding the most economical route of delivering goods and the problem of maximizing the network capacity can be solved using graphs.

Chou [29] gradually formalized a considerable part of graph theory in HOL theorem prover. The theory of undirected graphs is formalized as empty graphs, single-node graphs, finite graphs, subgraphs, paths, reachability, acyclicity, trees, subtrees, and merging disjoint subgraph of a graph. Based on this formalization, the correctness of distributed algorithms has been verified in HOL [30].

The authors in [31] modeled the graph structure of a BDD as heaps then they verified BDDs normalization algorithm in Isabelle/HOL theorem prover. The normalization follows the original algorithm presented by Bryant in 1986 which transforms an ordered BDD into a reduced, ordered and shared BDD. The verification is based on Schirmers research on the Verification Condition Generator (VCG) to generate the proof obligations for Hoare Logic.

The authors in [32] implemented and proved the correctness of BDDs operations completely formalized in Coq. Their objective was to extract certified algorithms for BDDs operations running in Caml (the implementation language of Coq). In their formalization, BDDs were represented as DAGs and the memory in which the BDDs nodes were stored has been formalized as well using pointers such that no new nodes will be created unless necessary. The model was more abstract and the normalization algorithm was not an issue and sharing was simply ignored, since only normalized BDDs will be constructed at all. The main difficulties of the graph formalization is related to data structure sharing and to the side-effects resulted in the computation. The algorithms usually mark the processed nodes or store the results calculated for a subtree or subgraph in a hash-table to avoid recalculation. Moreover, it would be very difficult to get a good performance that it is clearly never expected to compete (in terms of time and space usage) with BDDs libraries written in languages like C. The advantage of course is that there is very little work in this area so probably much scope for research.

On the other hand, modeling the transition relations as terms and formulae is smoother for proofs especially those based on induction. Also, in applications like model checking, one would deal with several terms, and any efficient implementation must define sharing. The work presented in [4, 33-36] is an example of this logical approach.

The choice between the two approaches depends on the objectives. If we want to reason about the implementation itself and its correctness, then its better to define transition relations as graphs and do sharing of common sub-trees. Clearly this makes the development and proofs complex. On the other hand, if we are only interested in a high-level view of algorithms, then a logical representation is preferred.

HYBRID APPROACH

We start by reviewing some work of linking proof systems to external automated verification tools. We concentrate on high-order logic proof systems since they are used much more often as a general property language and the logic is more expressive.

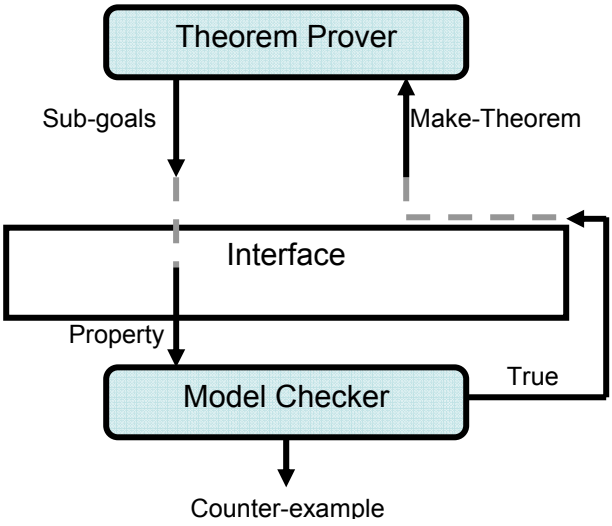

Fig.1: Theorem Proving and Model Checking Interface

The hybrid approach implement a tool linking model checking and theorem proving. During the verification procedure, the user deals mainly with the theorem proving tool. Verification using hybrid approach proceeds as shown in (Fig.1). The user starts by providing the theorem proving with the design (specification or implementation), the property and the goal to be proven. If the goal fits the required pattern, the theorem proving tool generates the required model checking files (sub-goals). The latter are sent to the model checking tool for verification. If the property holds, a theorem is created. Otherwise, the proof is performed interactively.

There are many examples of integration tools such as integrating the HOL system (HOL88) and Voss. Voss system [37], an implementation of Symbolic Trajectory Evaluation (STE), was implemented in a lazy Functional Language (FL). In [4] Voss was interfaced to HOL and the verification using a combination of deduction and STE was demonstrated. The HOL-Voss system integrates HOL88 deduction with BDDs computations.

The BDDs tools are programmed in FL as a built-in datatype. The assertion language of Voss was formalized in HOL and a tactic, which can make an external call to the Voss system, checks whether an assertion is true. Then the proved assertion was returned as a HOL theorem. A system based on this idea, called Voss-ThmTac, was later developed by Aagaard *et al.* [3], which combines the ThmTac theorem prover with the Voss system. Then the development of HOL-Voss evolved into a new system called Forte [38]. The idea comes from the very tight integration of the two provers, using a single language, FL, as both the theorem prover's meta-language and its object language. Thus lifted FL programs could evaluate FL expressions directly instead of having to translate back and forth, achieving the goal of efficiently unifying the model checker and theorem prover's specification languages. More recently, with industrial take-up at Intel, Forte [39] has become one of the most mature formal verification environments based on tool integration.

Rajan et al. [8, 40] described an approach where a BDD-based model checker for the propositional μ-calculus has been used as a decision procedure within the framework of PVS. An extension of the μ-calculus, consists of Quantified Boolean Formulae (QFB), is defined using PVS higher-order logic. The temporal operators are then defined using the μ-calculus. These temporal operators apply to arbitrary state spaces. In the case where the state type is constructed in a hereditarily finite manner, μ-calculus expressions are translated into input acceptable by a μ-calculus model checker. This model checker can then be used as a decision procedure to prove certain subgoals. The model checker accepts the translated input from μ-calculus expression. The generated subgoals are verified by the model checker and the results are used in the proof process of PVS.

Schneider et al. [5] used higher order hardware formulae to express the safety and liveness properties hierarchically. They proposed an approach of invoking model checking within HOL where properties are translated from HOL to temporal logic. A new class of higher-order formulae was presented, which allows a unified description of hardware structure and behavior at different levels of abstraction. Datapath oriented verification goals involving abstract data types can be expressed by these formula as well as control dominated verification goals with irregular structure. To ease the proof of the goals in HOL, a translation procedure was presented which converts the goals into several Computational Tree Logic (CTL) model checking problems, which are then solved outside HOL.

Schneider and Hoffmann [9] linked the SMV model checker to HOL using PROSPER. It provides an open proof architecture for the integration of different verification tools in a uniform higher-order logic environment. They embedded the linear time temporal logic (LTL) in HOL and translated LTL formulae into ω-Automata, a form that can be reasoned about within SMV. The translation is completely implemented by means of HOL rules. HOL terms are exported to SMV through the PROSPER plug-in interface. On successful model checking, the results are returned to HOL and turned to theorems. This integration tool allows SMV to be used as a HOL decision procedure. The deep embedding of the SMV specification language in HOL allows LTL specifications to be manipulated in HOL.

MDGs [41] are decision diagrams based verification tool, primarily designed for hardware verification. It is based on Multiway Decision Graphs which extend ROBDDs [2] with abstract sorts and uninterpreted function symbols. MDGs are canonical representations of a certain class of quantifier-free formulae of the logic called Directed Formulae (DFs) [19]. DFs can represent the transition and output relations of a state of machine, as well as the set of states.

The MDG-HOL system [6] is a hybrid system which links the HOL interactive proof system and the MDGs automated hardware verification system. It supports a hierarchical verification approach and fits the use of MDGs verification naturally within the HOL framework for a compositional hierarchical verification. The HOL system is used to manage the proof. The MDGs system is called to verify the submodules of a design. When the MDG-HOL system is used to verify a design, the design is modeled as a hierarchy structure with modules divided into submodules. An extension of the above work was presented in [7] to link HOL and the MDGs model checker. The interface between the two tools is implemented using ML.

Haiyan *et al.* [42] verified formally the linkage between a simplified version of MDG tool and the HOL

theorem prover. The verification is based on the importing of MDGs results to HOL theorems. Then, they combined translator correctness theorems with the linkage theorems in order to allow low level MDGs verification results to be imported into HOL in terms of the semantics of MDG-HDL. The work was concerned with ways of increasing trust in the linked systems.

DEEP EMBEDDING APPROACH

In this approach, a model checking is implemented inside a theorem proving tool. As shown in (Fig.2), the design and the property are fed to the model checking inside the theorem prover. If the property holds then a theorem is created, otherwise, the proof cannot be performed.

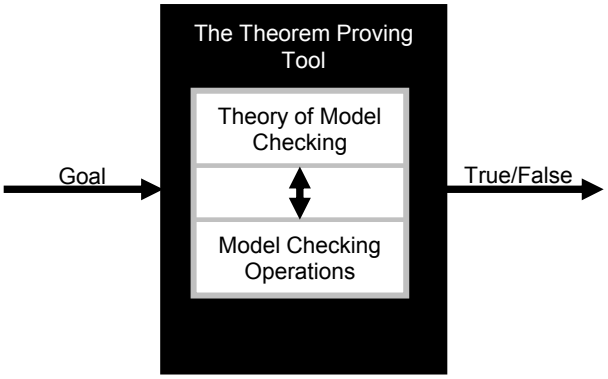

Fig.2: Embedding Model Checking inside Theorem Proving Tool

The result of the model checker is correct by construction, since both of the theory and the implementation are proved correct in the theorem prover. Thus soundness is guaranteed because more work is backed up by mechanized fully-expansive proof. The price for the extra proof and flexibility is in increased development effort.

The "deep embedding" approach[43] introduce the decision graph syntax as a new higher order logic type and then define the operations and algorithms based on this syntax within the theorem prover. This contrasts within a "shallow embedding" where the syntax is not formally represented in the logic, only in the meta-language. In general, a deep embedding allows one to reason about the language itself rather than just the semantics of programs in the language.

Gordon [44] integrated the BDDs based verification system BuDDY into HOL by implementing the primitive BDDs operations as inference rules added to the core of the theorem prover. The author used a small kernel of ML functions to convert between BDDs, terms and theorems. As long as those primitives are correct, it was possible to achieve the advantages of both theorem proving tools and decision diagram algorithms without the need to trust a complete external package, just a set of primitives. The aim of using BuDDy is to get near the performance of C-based model checking by using BDDs package implemented in C, whilst remaining fully expansive, though with a radically extended set of inference rules [34].

In [35], Harrison implemented BDDs inside the HOL system without making use of external oracle. The BDDs algorithms were used by a tautology-checker; however, the author found that the performance was about thousand times slower than with BDDs engine implemented in C. The author argued that by re-implementing some of HOL's primitive rules, performance could be improved by around ten times.

Amjad [33] demonstrated how BDDs based symbolic model checking algorithms for the propositional μ calculus L_μ can be embedded in the HOL theorem prover. His approach allows results returned from the model checker to be treated as theorems in HOL. By representing primitive BDDs operations as inference rules added to the core of the theorem prover, the execution of a model checker for a given property is modeled as a formal derivation tree rooted at the required property. These inference rules are hooked to a high performance BDDs engine [34] external to the theorem prover. The approach still leaves results reliant on the soundness of the underlying BDDs tools. Thus, the security of the theorem prover is compromised only to the extent that the BDDs engine or the BDDs inference rules may be unsound.

In [36], the authors followed a similar approach to the BuDDy work [34] but embedding MDGs rather than BDDs and inside HOL rather than using ML. They provided a complete formalization of the MDGs logic and its well-formedness conditions as DFs in HOL mechanically. Based on this infrastructure they formalized the basic MDGs operations in HOL following a deep embedding approach and proved their correctness. Their aim is to embed the MDGs model checker in the HOL theorem prover. Whereas [33] approach relies on the BDDs computations carried out by BDDs primitive inference rules, [36] work focuses more on how one can raise the level of assurance by embedding and proving formally the correctness of those operators in HOL.

In [45], the author showed a mechanism of how certifying model checker can be constructed. The idea

is that, a model checker can produce a deductive proof on either success or failure. The proof acts as a certificate of the result, since it can be checked independently. A certifying model checker thus provides a bridge from the model-theoretic to the proof-theoretic approach to verification. The author developed a deductive proof system for verifying branching time properties expressed in the μ-calculus, and showed it to be sound. Then, a proof generation in this system from a model checking run is presented.

In [46], the authors successfully carried out the verification of the model checker RAVEN. RAVEN is a real-time model checker which uses time-extended finite state machines (interval structure) to describe systems and a timed version of CTL (CCTL) to describe properties. Optimized algorithms based on an extended characteristic functions are used to compute the extension sets. The specification and the correctness proof were carried out using an interactive specification and verification system KIV.

CONCLUSION

Finally, in this paper we discussed a formalization of model checking approach in high order theorem provers and we presented an extended survey of relevant work.

BDDs based symbolic model checking has been proved to be a successful automatic verification technique that can be applied to real designs. However, the state space explosion problem caused by large datapaths is often the bottleneck in applying the symbolic model checking technique. Theorem provers are based on expressive formalism that are capable of modeling complex systems but requires expertise to verify most properties of practical interest. It has been shown through several research papers that model checking can be efficiently combined with theorem proving in a way that sacrifices neither efficiency of the former nor the expressiveness of the latter.

REFERENCES

1. Kropf T., 1999. Introduction to Formal Hardware Verification. Springer-Verlag.
2. Bryant, R., 1986. Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers, 35(8):677–691.
3. Aagaard, M., R. Jones and C. Seger, 1998. Combining Theorem Proving and Trajectory Evaluation in an Industrial Environment. In DAC, pp: 538-541.
4. Joyce, J. and C. Seger, editors, 1994. The HOL-Voss system: Model checking inside a general-purpose theorem prover, vol. 780, Springer-Verlag.
5. Schneider, K., and T. Kropf, 1995. Verifying Hardware Correctness by Combining Theorem Proving and Model Checking. Technical Report SFB358-C2-5/95, Institut f¨ur Rechnerentwurfund Fehlertoleranz.
6. Kort, S., S. Tahar, and P. Curzon, 2003. Hierarchal verification using an MDG-HOL hybrid tool. International Journal on STTT, 4(3):313–322.
7. Mizouni, R., S. Tahar, and P. Curzon, 2006. Hybrid verification incorporating HOL theorem proving and MDG model checking. Microelectronics Journal, 37(11):1200-1207.
8. Rajan, S., N. Shankar, and M. Srivas, 1995. An integration of model checking with automated proof checking. In CAV, vol. 939, pp: 84–97, Liege, Belgium. Springer-Verlag.
9. Schneider, K., and D. Hoffmann, 1999. A HOL conversion for translating linear time temporal logic to ω-automata. In TPHOLs, vol. 1690, pp: 255–272, Nice, France. Springer-Verlag.
10. Gordon M. and T. Melham, editors, 1993. Introduction to HOL (A theorem-proving environment for higher order logic). Cambridge University Press.
11. Paulson L., 1994. Isabelle: A Generic Theorem Prover. Springer-Verlag.
12. Crow, J., S. Owre, J. Rushby, N. Shankar, and M. Srivas. A Tutorial introduction to PVS. http://www.dcs.gla.ac.uk/proper/papers.html.
13. Huet, G., G. Kahn, and C. Paulin-Mohring. The Coq Proof Assistant: A Tutorial. http://coq.inria.fr/doc/tutorial.html.
14. Clarke, E., O. Grumberg, and D. Long, 1997. Model checking. LNCS, vol. 1346, pp: 54-200. Springer-Verlag.
15. Quille, J., and J. Sifakis, 1982. Specification and verification of concurrent systems in CESAR. In the 5th International Symposium on Programming, vol. 137, pp: 337–351. Springer-Verlag.
16. Holzmann G., 1990. Design and Validation of Computer Protocols. Prentice Hall.
17. Kurshan, R., and L. Lamport, 1993. Verification of a multiplier: 64 bits and beyond. In CAD, vol. 697, pp: 166–179, Elounda, Greece. Springer-Verlag.
18. McMillan K., 1993. Symbolic model checking. Kluwer Academic Publishers, Boston, Massachusetts.

19. Xu, Y., X. Song, E. Cerny, and O. Ait Mohamed, 2004. Model checking for a first-order temporal logic using multiway decision graphs (MDGs). The Computer Journal, 47(1):71–84.
20. Bryant, R., 1992. Symbolic Boolean manipulation with ordered binary decision diagrams. ACM Computing Surveys, 24(3):293–318.
21. Davis, M., G. Logemann, and D. Loveland, 1962. A machine program for theorem-proving. Commun. ACM, 5(7):394–397.
22. Sheeran, M., S. Singh, and G. Staalmarck, 2000. Checking safety properties using induction and a sat-solver. In FMCAD'00, pp: 108–125, London, UK. Springer-Verlag.
23. Bjesse, P., and K. Claessen, 2000. SAT-based verification without state space traversal. In FMCAD, pp: 372–389.
24. Ganai, M., and A. Aziz. Improved sat-based bounded reachability analysis, 2002. In ASPDAC, pp: 729-734, Washington, USA. IEEE Computer Society.
25. Moskewicz, M., C. Madigan, Y. Zhao, L. Zhang, and S. Malik, 2001. Chaff: Engineering an Efficient SAT Solver. In DAC, pp: 530-535.
26. Marques-Silva, J., and K. Sakallah, 1996. GRASP - A New Search Algorithm for Satisfiability. In CAD, pp: 220–227.
27. Zhang, H., 1997. SATO: an efficient propositional prover. In CADE, vol. 1249 of LNAI, pp: 272–275.
28. Archer, M., J. Joyce, K. Levitt, and P. Windley, 1992. A Simple Graph Theory and Its Application in Railway Signalling. HOL Theorem Proving System and Its Applications, pp: 395-409.
29. Chou, C., 1994. A formal theory of undirected graphs in higher-order logic. In TPHOLs, vol. 859, pp: 144–157, Malta. Springer-Verlag.
30. Chou, C., 1994. Mechanical verification of distributed algorithms in higher-order logic. In TPHOLs, vol. 859, pp: 158–176, Malta. Springer-Verlag.
31. Ortner, V., and N. Schirmer, 2005. Verification of BDD normalization. In TPHOLs, pp: 261–277.
32. Verma, K., J. Goubault-Larrecq, S. Prasad, and S. Arun-Kumar, 2000. Reflecting BDDs in Coq. In ASIAN, Penang, Malaysia, vol. 1961, pp: 162–181. Springer-Verlag.
33. Amjad, H., 2003. Programming a symbolic model checker in a fully expansive theorem prover. In TPHOLs, vol. 2758, pp: 171–187. Springer-Verlag.
34. Gordon, M., 2002. Programming combinations of deduction and BDD-based symbolic calculation. LMS Journal of Computation and Mathematics, 5:56–76.
35. Harrison, J., 1995. Binary decision diagrams as a HOL derived rule. The Computer Journal, 38:162–170.
36. Abed, S., and O. Ait Mohamed, 2006. Embedding of MDG directed formulae in HOL theorem prover. In MCSEAI, pp: 659–664.
37. Seger, C., 1993. VOSS – a formal hardware verification system, user's guide. Technical report TR-93-45, Nortel Networks, Ottawa, Canada, The University of British Columbia.
38. Aagaard, M., R. Jones, R. Kaivola, K. Kohatsu and C. Seger, 2000. Formal verification of iterative algorithms in microprocessors. In DAC, pp: 201-206.
39. Melham, T., 2004. Integrating model checking and theorem proving in a reflective functional language. In IFM, Canterbury, UK, vol. 2999, pp: 36–39. Springer-Verlag.
40. Owre, S., S. Rajan, J. Rushby, N. Shankar, and M. Srivas, 1996. PVS: Combining specification, proof checking, and model checking. In CAV, vol. 1102, pp: 411–414, USA. Springer Verlag.
41. Corella, F., Z. Zhou, X. Song, M. Langevin, and E. Cerny, 1997. Multiway decision graphs for automated hardware verification. In Formal Methods in System Design, vol. 10, pp: 7–46.
42. Xiong, H., P. Curzon, S. Tahar, and A. Blandford, 2006. Providing a formal linkage between MDG and HOL. Formal Methods in Systems Design, 29(3):1–36.
43. Boulton, R., A. Gordon, M. Gordon, J. Herbert, and J. van Tassel, 1992. Experience with embedding hardware description languages in HOL. In International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience, pp: 129–156, Nijmegen, North-Holland.
44. Gordon, M., 2000. Reachability programming in HOL98 using BDDs. In TPHOLs, pp: 179–196.
45. Namjoshi, K., 2001. Certifying model checkers. LNCS, 2102:2–13.
46. Reif, W., J. Ruf, G. Schellhorn, and T. Vollmer, 2000. Do you trust your model checker? In FMCAD, vol. 1954, pp: 385-393, Springer-Verlag.