Providing Automated Verification in HOL Using MDGs

Tarek Mhamdi and Sofiène Tahar

Department of Electrical and Computer Engineering
Concordia University, Montreal, Canada
{mhamdi,tahar}@ece.concordia.ca

Abstract. While model checking suffers from the state space explosion problem, theorem proving is quite tedious and impractical for verifying complex designs. In this work, we present a verification framework in which we attempt to strike the balance between the expressiveness of theorem proving and the efficiency and automation of state exploration techniques. To this end, we propose to integrate a layer of checking algorithms based on Multiway Decision Graphs (MDG) in the HOL theorem prover. We deeply embedded the MDG underlying logic in HOL and implemented a platform that provides a set of algorithms allowing the user to develop his/her own state-exploration based application inside HOL. While the verification problem is specified in HOL, the proof is derived by tightly combining the MDG based computations and the theorem prover facilities. We have been able to implement and experiment with different state exploration techniques within HOL such as MDG reachability analysis, equivalence and model checking.

1 Introduction

Whenever an error creeps into a design, time and money must be spent to locate the problem and correct it, and the longer a bug evades a detection, the harder and more expensive it is to fix. As design complexity increases, simulation times become prohibitive and coverage becomes poor, allowing numerous bugs to slip through to later stages of the design cycle. What is needed, therefore, is a complement to simulation for determining the correctness of a design. For this reason, there has been a surge of research interest in *formal verification* techniques. In general, a formal verification problem consists of mathematically establishing that an implementation satisfies a specification. The implementation refers to the system design that is to be verified and the specification refers to the property with respect to which the correctness is to be determined.

Formal verification methods fall into two categories: *proof-based* methods, mainly theorem proving and *state-exploration* methods, mainly model checking and equivalence checking. While theorem proving is a scalable technique that can handle large designs, model checking suffers from the so-called state-explosion problem which prevents its application to industrial systems. On the other hand,

F. Wang (Ed.): ATVA 2004, LNCS 3299, pp. 278–293, 2004.

while model checking is fully automatic, deriving proofs is a user guided technique that requires a lot of expertise and hence can be tedious and difficult. Both techniques do not allow the automatic verification of large systems. So, various compromises are being explored to combine the strengths of both. They can be summarized as : (i) tools integration, (ii) adding deduction rules to a state-of-the-art checking tool, or (iii) deeply embedding checking algorithms inside a theorem prover. For the first approach, we start with two stand-alone tools, a theorem prover and a checking tool, where we link the latter to the theorem prover using scripting languages to be able to automatically verify small subgoals generated by the theorem prover from a large system. The starting point of the second approach is an automatic (model) checker to which we add proving rules to hopefully extend the verification to complete systems. Finally, the third approach, which is the one we adopt in our work, consists of embedding algorithmic infrastructures inside a theorem prover resulting in a hybrid system tightly combining checking algorithms and proving facilities. This approach differs from the first one in the way the verification is performed. In fact, we do not use an external checking tool, instead we develop state-exploration algorithms inside the theorem prover.

In this work, we developed a platform of state-exploration algorithms inside the HOL proof system [9]. Our decision diagram data structure is the *Multiway Decision Graphs* (MDGs) [4], which we integrate in HOL as a built-in datatype. The logic underlying MDGs is embedded as a theory that provides the tools to specify the verification problem in the logic supported by the MDGs. The specification consists of a set of HOL formulae that can be represented by their correspondent MDGs. Operations over these formulae are viewed as MDG operations over their respective graphs. An MDG package is then used to build the graph representation of HOL formulae allowing the manipulation of graphs rather than HOL terms. Once available inside the theorem prover, the MDG data structure and operators can be used to automate parts of the verification problem or even to write state enumeration algorithms like reachability analysis or model checking.

The organization of this paper is as follows: Section 2 reviews some related work. Section 3 describes the embedding of the logic underlying the MDGs in HOL. Section 4 shows how HOL is linked to the MDG package. In Section 5, we describe the embedding of the reachability analysis procedure. Sections 6 and 7 illustrate the use of the embedding in the implementation of state-exploration algorithms and decision procedures, respectively. Section 8, finally, concludes the paper and gives some future research directions.

2 Related Work

The quest for an efficient combination of theorem proving and model checking has long been one of the major challenges in the field of formal verification. The work described here has been strongly influenced by the HolBdd [7,8] system developed by Gordon. HolBdd consists of a platform allowing the programming

of Binary Decision Diagrams (BDD) based symbolic algorithms in the Hol98 proof assistant. It provides intimate combinations of deduction and algorithmic verification. They use a small kernel of ML functions to convert between BDDs, terms and theorems. Their work was applied to perform reachability programming in Hol98.

A pioneering work in the area of linking theorem proving with automated verification tools is the one of Joyce and Seger [11] combining HOL and the symbolic trajectory evaluation (STE) tool VOSS. A HOL tactic, VOSS_TAC, calls the VOSS system as a child process of the HOL system to check whether an assertion, expressed as a term of higher-order logic, is true. Early experiments with HOL-VOSS suggested that a lighter theorem prover component was sufficient, since all that was needed was a way of combining results obtained from STE. A system based on this idea, called Voss-ThmTac, was later developed by Aagaard *et al.* [1], which combines the ThmTac theorem prover with the VOSS system. Its power 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.

Rajan *et al.* [18] 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 the PVS proof checker [16]. They used μ-calculus as a medium for communicating between PVS and the model checker. Temporal operators are given the customary fixpoint definitions using the μ-calculus. These expressions were translated to the form required by the model checker.

Hurd [10] used PROSPER [5] to combine the Gandalf first-order theorem prover with HOL. A HOL tactic, GANDALF_TAC, is used to enable first-order HOL goals to be proven by Gandalf and mirror the resulting proofs in HOL. It takes the original goal, converts it to the appropriate format, and sends it to Gandalf. Gandalf then parses the proof, translates it to a HOL proof and proves the original goal in HOL.

Schneider and Hoffmann [19] linked the SMV model checker [13] to HOL using PROSPER. They embedded the linear time temporal logic (LTL) in HOL and translated LTL formulae into equivalent ω-Automata, a form that can be reasoned about within SMV. The translation is completely implemented by means of HOL rules. The deep embedding of the SMV specification language in HOL allows LTL specifications to be manipulated in HOL.

In [12], [17] and later [15] a hybrid tool and a methodology tailored to perform hierarchical hardware verification have been developed by the *Hardware Verification Group* of *Concordia University*. The hybrid tool, called HOL-MDG, integrates the HOL theorem prover with the MDG tool by performing equivalence and model checking using two HOL tactics, MDG_EQ_TAC and MDG_MC_TAC, respectively. In case the design is large enough to cause state explosion, and assuming a hierarchical model, a tactic HIER_VERIF_TAC is called to break the design into sub-blocks. The same procedure is recursively applied if necessary. At any point, the goal proof can be done in HOL.

While [12,15,17] describe systems integrating two stand-alone tools, namely, HOL and an external MDG tool, the work described in this paper is not intended to use an external tool to verify subgoals. Instead, MDGs are defined as a built-in datatype of HOL and operators over MDGs are available in the proof system, which allows us to tightly combine HOL deduction and MDG computations. Besides, state-exploration algorithms will be written inside HOL. Thereafter, the main difference between our approach and the HOL-MDG tool is that our embedding provides a secure and general programming infrastructure to allow the users to implement their own MDG-based verification algorithms inside the HOL system.

The work in [1,10,11,19] use the same approach as the HOL-MDG hybrid tool in the way they integrate the model checker to the theorem prover. The work in [18] uses the μ-calculus as a medium for communicating between the theorem prover and the model checker. It is a shallow embedding of a stand-alone tool's language while ours is a deep embedding of the decision diagram data structure and its operators are embedded inside the theorem prover.

Obviously, the most related work to ours is that of Gordon [7,8]. Our work, however, deals with embedding MDGs rather than BDDs. In fact, while BDDs are widely used in state-exploration methods, they can only represent Boolean formulae. MDGs, however, represent a subset of first-order terms allowing the abstract representation of data and hence raising the level of abstraction.

3 Embedding the MDG Logic in HOL

3.1 Multiway Decision Graphs

A *Multiway Decision Graph* (MDG) is a finite directed acyclic graph G where the leaf nodes are labeled by formulae, the internal nodes are labeled by terms, and the edges issuing from an internal node N are labeled by terms of the same sort as the label of N. Such a graph represents a formula defined inductively as follows: (i) if G consists of a single node labeled by a formula P, then G represents P; (ii) if G has a root node labeled A with edges labeled $B_1, ..., B_n$ leading to subgraphs $G'_1, ..., G'_n$ and if each G'_i represents a formula P_i then G represents the formula $\vee_{1 \leq i \leq n}((A = B_i) \wedge P_i)$.

The above is of course too general, a set of *well-formedness conditions* [4] turns MDGs into *canonical representations* that can be manipulated by efficient algorithms. Multiway Decision Graphs are intended to represent Abstract State Machines (ASM) [4], an abstract description of state machines based on a many-sorted first order logic with a distinction between abstract and concrete sorts. It is then possible to let nodes range over abstract sorts for which there is no enumerable set of edges, and to use non-mutually-exclusive first-order terms as edge labels. More details on MDG are described in the sections to follow.

3.2 MDG Sorts

An enumeration is a finite set of constants. While concrete sorts have enumerations, abstract sorts do not. This is embedded in HOL using two constructors

called *Concrete_Sort* and *Abstract_Sort*. The former takes as arguments a sort name and its enumeration to define a concrete sort. For example, if *state* is a concrete sort with [*stop*; *run*] as enumeration, then this is declared in HOL by:

$\vdash_{def}$ `state = Concrete_Sort "state" [stop;run]`

To define an abstract sort of type *alpha* (which means that the sort is actually abstract and hence can represent any HOL type) we use the *Abstract_Sort* constructor as follows:

$\vdash_{def}$ `alpha = Abstract_Sort "alpha"`

To determine whether a sort is concrete or abstract, we use predicates over the sorts constructors called *IsConcreteSort* and *IsAbstractSort*. These predicates will be used for instance to determine the sort of a variable or a function symbol.

The vocabulary of the MDG based logic consists of concrete and generic constants, variables and function symbols (also called operators). The distinction between abstract and concrete sorts leads to a distinction between three kinds of function symbols. Let f be a function symbol of type $\alpha_1 \times ... \times \alpha_n \rightarrow \alpha_{n+1}$. If α_{n+1} is an abstract sort then f is an *abstract function symbol*. Abstract function symbols are used to denote data operations and are uninterpreted. If all $\alpha_1 ... \alpha_{n+1}$ are concrete, f is a *concrete function symbol*. Concrete function symbols, and concrete constants as a special case, can always be entirely interpreted and thus be eliminated; for simplicity, we assume that they are not used. Finally, if α_{n+1} is concrete while at least one of $\alpha_1 ... \alpha_n$ is abstract, then we refer to f as a *cross-operator*.

3.3 MDG Variables

An abstract variable can be either primary or a secondary variable. A primary variable labels a node in the graph while a secondary variable is an abstract variable occurring in the argument list of a function symbol. It can also be an abstract variable labeling an edge in the graph. In our embedding, a primary abstract variable is declared using the *Abstract_Var* constructor. The *Secondary_Var* constructor is used to declare a secondary variable.

A variable is identified by its name and sort. For example, If x is a concrete variable of sort *state*, declared above, then this is written in HOL as follows:

$\vdash_{def}$ `x = Concrete_Var "x" state`

Similarly, we use some predicates to determine whether a variable is concrete, abstract or secondary. They are called, respectively, *IsConcreteVar, IsAbstractVar* and *IsSecondaryVar*.

3.4 MDG Constants

A constant can be either an individual (concrete) constant or an abstract generic constant. The latter is identified by its name and its abstract sort. The individual

constants can have multiple sorts depending on the enumeration of the sort in which they are. We use the *Individual_Const* and *Generic_Const* constructors to declare constants in HOL. For example, the enumeration of the concrete sort *state* is [*stop* ; *run*]. *stop* and *run* are two individual constants that have *state* as their sort. They must be already defined in order to be able to declare the sort *state*. To check whether a constant is an individual constant or an abstract generic constant, we define two predicates, *IsIndividualConstant* and *IsGenericConstant*.

3.5 MDG Functions

MDG functions can be either concrete, abstract or cross-operators. As mentioned before, concrete functions are not used since they can be eliminated by case splitting. Cross-functions are those that have at least one abstract argument. But when we focus on terms that are concretely reduced, all the sub-terms of a compound term (abstract/cross function) have to be abstract. In addition they are secondary variables.

In general, a function is identified by its name, the sorts of its arguments and its sort. In this case, we specify the variables rather than sorts because we focus on cross-terms or abstract terms instead of the correspondent symbols. If *equal* is a function that checks if two abstract variables are equal, then, *equal* is a cross-function.

```
⊢def  bool = Concrete_Sort "bool" ["0";"1"]
⊢def  y1 = Secondary_Var "y1" alpha
⊢def  y2 = Secondary_Var "y2" alpha
⊢def  equal = Cross_Function "equal" [y1;y2] bool
```

If *max* is a function that takes two abstract variables as arguments and returns the greater one, then *max* is an abstract function.

```
⊢def  max = Abstract_Function "max" [y1;y2] alpha
```

The predicates *IsAbstractFunction* and *IsCrossFunction* are used to determine the nature of a compound term.

3.6 MDG Terms

MDG terms are either individual constants, generic constants, concrete or abstract variables, cross-operators or abstract function symbols. We provide a constructor called *MDG_Term* that is used every time a new term is declared. The single constructor is used so that terms will have the same type and hence can be used in equalities. In fact if x is declared using the *Concrete_Var* constructor and *stop* using the *Individual_Const* constructor, we will not be able to write an equation of the form $x = stop$ due to type mismatching. However, such an equation is possible if both are declared using the same constructor.

3.7 Well-Formed MDG Terms

For BDDs to be canonical, they have to be reduced and ordered. Similarly, MDGs require certain well-formedness conditions to canonically represent the MDG terms. Such terms are called *Directed Formulae* (DF). Given two disjoint sets of variables U and V, a DF of type $U \rightarrow V$ is a formula in disjunctive normal form (DNF) such that

1. Each disjunct is a conjunction of equations of the form:
   - $A = a$, where A is a cross-term of concrete sort α containing no variables other than elements of U, and a is an individual constant in the enumeration of α, or
   - $u = a$, where $u \in U$ is a variable of concrete sort α and a is an individual constant in the enumeration of α, or
   - $v = a$, where $v \in V$ is a variable of concrete sort α and a is an individual constant in the enumeration of α, or
   - $v = A$, where $v \in V$ is a variable of abstract sort α and A is a term of type α containing no variables other than elements of U;
2. In each disjunct, the left hand sides of the equations are pairwise distinct; and
3. In each disjunct, every variable $v \in V$ should appear as the left hand side of an equation $v = A$.

Conditions 2 and 3 must be respected by the user when specifying the verification problem. Condition 3 is less stringent than it seems. In practice, one can introduce an additional dependent variable u and add an equation $v = u$ to a disjunct where an abstract v is missing.

For example, we embedded condition 1 in HOL and check it using the function *Well_formedTerm* that, recursively, calls *Well_formedEQ* to check the well-formedness of an equation. In the definition below, *eq* is an equation of the form *lhs* = *rhs*.

```
⊢def  Well_formedEQ eq =
      ((IsConcreteVar lhs)   ∧ (IsConcreteConstant rhs)) ∨
      ((IsCrossFunction lhs) ∧ (IsConcreteConstant rhs)) ∨
      ((IsAbstractVar lhs )  ∧ (IsAbstractFunction rhs)) ∨
      ((IsAbstractVar lhs )  ∧ (IsAbstractVar rhs))      ∨
      ((IsAbstractVar lhs )  ∧ (IsGenericC rhs))         ∨
      (IsBool lhs)
```

4 Linking HOL to the MDG Package

The MDG logic is embedded in HOL to make it possible to specify a verification problem in HOL in terms of formulae that can be represented by canonical MDGs. The next step would be to provide the necessary tools to build and manipulate the graph representations of these formulae. This platform will consist of ML functions that call an MDG package as an external process. The package

is invoked using a script file, in which, the different manipulations to be done in MDG are specified. For example, to perform the conjunction of a list of well-formed terms, we use the ML function *Conj*. This function calls an intermediate function to write the script file corresponding to a conjunction, then calls the specific MDG functions to perform the operation and eventually return the result to HOL. The ML functions pass the script file to the MDG package using the *system* function. The latter computes the result (MDG graph) and then writes it in a file "*mdghol.ch*". Using the function *ReadMdgOutput*, the result is retrieved.

4.1 Constructing MDGs in HOL

To construct the graph representation of a HOL term, we use the function *termToMdg*. Well-formedness conditions are first checked using the predicate *Well_formedTerm*, which either raises an exception when this is not the case or begins gathering the information to call the package.

The first step is to determine the sorts of all the sub-terms using the function *ToMdgSorts*. If a sub-term is of concrete sort *Sort*, it is declared as *concrete_sort(Sort,Enum)*, where *Enum* is the enumeration of *Sort*. When an abstract sort, say *alpha*, is encountered, then it is declared by *abs_sort(alpha)*. For example, if a term A includes a concrete variable of sort *bool* and an abstract variable of sort *alpha*, then *ToMdgSorts* returns the following list:

```
[conc_sort(bool,[0,1]),abs_sort(alpha)].
```

The second step is to declare all the variables, functions and generic constants used in the term. A variable is declared by *signal(label,sort)*. A generic constant is declared by *gen_const(label,sort)*. When a function is encountered, both the secondary variables and the function symbol must be declared. The function symbol is declared as *function(f,[sorts],sort)*. *sorts* are the sorts of the secondary variables, arguments to the function symbol f. *sort* is its target sort.

Thereafter, *termToMdg* writes the variables order list in the script file and then calls the function *header* responsible for retrieving the list of the LHSs and RHSs of the equations in the term which will be the parameters of the *mdg* function. The latter is then called and the result is retrieved using the *readMDGOutput* function. Instead of returning the whole graph structure, we return only its ID, which will be used to map the term to its MDG representation.

4.2 Embedding MDG Basic Operators

The MDG operators are embedded, as well, to allow the manipulation of graphs rather than terms. We show below the basic MDG operators.

- *Conj*: performs the conjunction of a set of graphs;
- *Disj*: performs the disjunction of a set of graphs;
- *Relp* (Relational Product): used for image computation. It takes the conjunction of a collection of MDGs, having pairwise disjoint sets of abstract

primary variables, and, existentially quantifies with respect to a set of variables, either abstract or concrete, that have primary occurrences in at least one of the graphs. In addition, it can rename some of the remaining primary variables according to a renaming substitution;
- *PbyS* (Pruning By Subsumption): used to approximate the set difference operation. Informally, it removes all the paths of a graph P from another graph Q.

5 Reachability Analysis in HOL

The reachability analysis is embedded using the MDG operators interfaced to HOL. We show here the different steps to compute the set of the reachable states of an abstract state machine.

5.1 Computing Next States

Let I, B and R be, respectively, a set of inputs, a set of initial states of a machine and its transition relation. The set of next states reached from B with respect to the transition relation R is computed using the ML function *ComputeNext*. This is done by, first, computing the graphs of I, B and R. The *RelP* operator is then used after identifying the renaming substitution function and the set of inputs and state variables over which the MDG is quantified. The resulting graph represents the set of next states.

5.2 Computing Outputs

The set of outputs corresponding to a set of initial states and inputs, with respect to an output relation O is computed in the same way as the next states. But instead of using the transition relation R of the machine, the output relation O is used. For every state of the machine, and a set of data inputs, corresponds a set of output values. These will be used to check if an invariant holds in the current state.

5.3 Computing Frontier Set

The frontier set is the set of newly visited states. If V represents the set of states already visited, V_n = *ComputeNext*(I V R) is the set of next states reached from V. In this case the frontier set is $V_n \setminus V$ which is represented by the ML function *ComputeFrontier*. The frontier set is used to check if all the states reachable by the machine are already reached. If this is the case (the frontier set is empty), then the reachability analysis terminates and the set of reachable states is returned. If the frontier set is not empty, then new states were visited during the last iteration. In this case, the analysis continues until reaching the fixpoint (set).

5.4 Computing Reachable States

The set of reachable states is the set of all the states of a machine, starting from an initial state, for a certain set of inputs. For abstract state machines, the state space can be infinite. Hence, the set of reachable states may not exist[1]. We implemented in HOL the solutions proposed in [2] to compute the set of reachable states which we represented by the function, *ComputeReachable*[2], defined in Figure 1.

```
ComputeReachable G_I G_B G_R =
        K = 0, S = G_B
        loop
            K = K + 1
            N = ComputeNext G_IK G_B G_R
            if ComputeFrontier N S = F then return success
            G_B = ComputeFrontier N S
            S = Disj N S
        end loop
end;
```

Fig. 1. Reachability Analysis Algorithm

ComputeReachable computes the set of reachable states S of a state machine described by its transition relation, starting from an initial state and for a certain data input. S is initialized to B (the initial state), and the sets of next-states are computed until reaching a fixpoint characterized by an empty frontier set.

6 Invariant and Model Checking in HOL

6.1 Invariant Checking

Invariant checking is a direct application of the reachability analysis algorithm. It consists of checking that a property or an invariant holds on the outputs of a state machine in every reachable state. First, the invariant is checked in the initial state. This is done by computing the outputs corresponding to that state and then using the MDG operators to check that these outputs satisfy the invariant. After that, next-states are computed and for every state reached, the invariant is checked on the outputs. In a given iteration, if the outputs of the machine satisfy the invariant, then the procedure continues for the next-state. If, on the other hand, the invariant does not hold, the analysis terminates and a failure is reported. A counterexample can be generated to trace the error.

[1] This is the well-known non-termination problem in MDG, which is discussed in [2] providing various heuristics to solve it.

[2] For the sake of clarity, this is just a simplified version of the algorithm

We implemented the invariant checking algorithm in HOL as an ML function *InvariantChecking* which takes as arguments:

- T_R: the transition relation specified as a list of directed formulae;
- O_R: the output relation specified by a directed formula;
- I_N: the initial state specified by a directed formula;
- *Inputs*: the input variables list;
- *States*: the state variables list;
- *NxStates*: the next-state variables list corresponding to *States*.
- *Inv*: the invariant to be checked, specified as a directed formula.

We implemented in HOL the function *InvariantChecking* as defined in Figure 2. It first, builds the graphs of the transition relation, output relation, the initial state and the invariant using the function *termToMdg*. Then, generates the input graph. After that, the outputs are computed using *NewOutputs* and then the invariant is checked. If the invariant holds, the next-state variables are computed using *ComputeNext*. Checking the frontier set will cause the termination of the analysis or another iteration.

```
InvariantChecking T_R O_R I_N Inputs States NxStates Inv =
    // builds the MDG representations
    K = 0, S = G_IN, R = G_IN
    loop
        K = K + 1
        // generates the input graph G_IK
        O_S = ComputeOutputs G_OR R G_IK
        if (PbyS O_S G_Inv) ≠ F return failure
        N = ComputeNext G_IK R G_TR
        if ComputeFrontier N S = F then return success
        R = ComputeFrontier N S
        S = Disj N S
    end loop
end InvariantChecking;
```

Fig. 2. Invariant Checking Algorithm

6.2 Model Checking

MDG temporal operators can be implemented in HOL for model checking. In Figure 3 we present, for illustration purposes, how the operator **AF** on a first-order property formula P [20] is embedded. All the MDG model checking algorithms are embedded in HOL in a similar fashion. More details can be found in [14].

```
Check_AF T_R I_N Inputs States NxStates P =
    // builds the MDG representations G_TR, G_IN, G_P
    K = 0, Σ = F, C = G_IN
    // Σ contains sets of states not satisfying P
    loop
        = ComputeFrontier C G_P
        // removes states satisfying P
        if Q = F then return success
        if ComputeFrontier Σ Q ≠ Σ then return failure
        Σ = Disj Σ Q
        K = K + 1
        C = ComputeNext G_IN Q G_TR
    end loop
end Check_AF;
```

Fig. 3. Model Checking Algorithm for AF

6.3 Application

We have experimented our embedding on some benchmark examples and case studies, including the Island Tunnel Controller (ITC), which was originally introduced by Fisler and Johnson [6]. The ITC controls vehicles traffic in a one-lane tunnel connecting the mainland to a small island. The ITC is specified using three communicating controllers and two abstract counters. We used the invariant checking procedure discussed above to verify a number of properties on the ITC. For each property, we derived those transition relations and variables involved in the property (specified manually) and let the property checking run automatically from within HOL. This reduces the verification problem and promotes hierarchical verification. In fact, every module of the design can be treated separately. Thus, enhancing a lot the performance of the verification task in terms of memory usage compared to verifying the whole system in MDG. It is needless to mention that the memory usage is one of the most challenging factors in formal verification as it is the cause of the state-space explosion problem.

Table 1 summarizes the verification results of checking a set of properties on ITC using: (1) pure MDG, (2) the MDG-HOL shallow embedding approach [15], and (3) our MDG-HOL deep embedding. A "*" beside a property means that this latter failed in the invariant checking, where a counterexample is generated. Experiments are run on an Ultra2 Sun workstation with 296Mhz CPU and 768MB memory. The CPU times represent the system times to perform the reachability analysis. They also include the time to translate the HOL specification to MDG files in the case of our HOL-MDG deep embedding. The memory usage statistics represent the total memory used by the MDG tool to build the different graphs. As expected, when the verification can be handled using pure

Table 1. Performance comparison for the ITC benchmark

Property	CPU_{sec}			Memory_{MByte}		
	MDG	HOL-MDG (Shallow)	HOL-MDG (Deep)	MDG	HOL-MDG (Shallow)	HOL-MDG (Deep)
Property1	0.87	1.15	101.9	0.66	0.47	0.220
*Property2	0.55	0.87	52.8	0.27	0.23	0.013
Property3	0.57	0.91	54.9	0.31	0.26	0.077
Property4	0.53	0.81	44.3	0.23	0.22	0.02
Property5	0.71	1.04	72.0	0.37	0.32	0.058
Property6	0.53	0.80	44.8	0.24	0.17	0.035
Property7	0.69	0.96	63.9	0.33	0.27	0.039
Property8	0.70	0.98	64.2	0.32	0.27	0.039
Property9	0.54	0.85	45.4	0.21	0.15	0.035

MDG or shallow embedding, the CPU time is lower compared to the time needed for verifying using our deep embedding of MDG in HOL. However, the latter remains reasonable especially when the timing is not the major performance measure. On the other hand, we notice the drastic memory usage reduction provided by the deep combination of HOL and MDG compared to using pure MDG or the shallow embedding of MDG in HOL. This reduction can be explained by the fact that we are considering the graphs of each directed formula separately instead of working with the whole system to verify, leading to a better garbage collection. This advantage is crucial because it would enable the handling of larger designs. More details about the ITC models and properties specification and verification can be found in [14].

7 MDG as a Decision Procedure

The multiway decision graphs are a canonical representation of the directed formulae. Two directed formulae are equivalent if and only if they are represented by the same graph for a fixed order. This property can be used to prove automatically the equivalence of HOL terms or to check that a formula is a tautology in case it is represented by the MDG *true*.

7.1 Combinational Equivalence Checking

We provide here a decision procedure that enables us to verify automatically the equivalence of a certain subset of first-order HOL terms. This is performed using the ML function *EquivCheck*.

$\vdash_{def}$
```
EquivCheck order t1 t2 =
 let s1 = termToMdg order t1
     s2 = termToMdg order t2
 in (s1=s2)
```

Using *EquivCheck* we write an oracle that builds a theorem stating the equivalence between terms. The theorem is not derived from axioms and inference rules, which will endanger the security provided by the HOL reasoning style. Theorems created using the oracle are tagged so that an error can be traced whenever it occurs. This kind of decision procedures are widely used to introduce some automation to the theorem provers.

7.2 Tautology Checking

A formula is a tautology if it is represented by the MDG T. This makes the check very easy for the subset we consider, which are the directed formulae. We use the ML function *Tautology* which we implemented in HOL.

$\vdash_{def}$ `Tautology order t =`
`let s = termToMdg order t`
`in (isTrue s)`

8 Conclusions and Future Work

Expertise and user guidance is a major problem for applying theorem proving on even the most trivial systems. On the other hand, state exploration techniques suffer from the state space explosion problem, which limits their applications to industrial designs. An alternative to these techniques would be to combine the advantages of both in a hybrid approach that will lead to a hopefully, automatic or semi-automatic technique, which can handle large designs. In this paper, we proposed an approach that allows certain verification problems, specified in the HOL theorem prover, to be verified totally or in part using state-exploration algorithms. Our approach consists of an infrastructure of decision diagrams data structure and operators made available in HOL, which will allow the user to develop his/her own state-exploration algorithms in the HOL proof system. The data structure we considered in our work is the multiway decision graphs (MDG). MDG is an extension to the well-known binary decision diagrams (BDD) in that it eliminates the state explosion problem introduced by the datapath.

The platform we provide allowed us to develop state-exploration algorithms inside HOL like reachability analysis, model checking and invariant checking procedures. We also developed decision procedures based on the MDGs allowing the equivalence and tautology checking of a certain subset of HOL terms automatically. Finally, we demonstrated the feasibility of our approach by considering some case study examples, which we have been able to verify using a seamless interaction between HOL and MDG.

The embedding of the MDGs in HOL opens the way to the development of a wide range of new verification applications combining the advantages of state-exploration techniques and theorem proving. There are many opportunities for further work on using this embedding for formal verification. For instance, MDG canonicity can be used in HOL for term simplification. In fact, when built, MDGs

are reduced by construction. Retrieving the term represented by this graph gives a simplification of the original one. The Embedding can be used for the formal proof of the soundness of the MDG algorithms extending the work in [21], where the correctness of the MDG system translators was proved, ensuring the correctness of the whole MDG system. A similar work was done in [3] to verify a SPIN model checking algorithm in HOL. Finally, the embedding can be enhanced by using the *LCF style* [9]. In this case, an MDG representation for a HOL term can only be derived using inference rules and trivial MDGs. The graph of a directed formula is derived from the graphs of its equations (trivial MDGs) and the MDG operators (inference rules). This restricts the scope of soundness to single operators which, are easier to get right [8].

References

1. M.D. Aagaard, R.B. Jones, and C.-J. Seger. Lifted-FL: A Pragmatic Implementation of Combined Model Checking and Theorem Proving. In *Theorem Proving in Higher-Order Logics*, volume 1690 of LNCS, pages 323–340, Springer-Verlag, 1999.
2. O. Ait Mohamed, X. Song, and E. Cerny. On the Non-termination of MDG-Based Abstract State Enumeration. *Theoretical Computer Science*, 300:161–179, August 2003.
3. C.-T. Chou and D. Peled. Verifying a Model-checking Algorithm. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of LNCS, pages 241–257, Springer-Verlag, 1996.
4. F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny. Multiway Decision Graphs for Automated Hardware Verification. *Formal Methods in System Design*, 10(1):7–46, 1997.
5. L.A. Dennis, G. Collins, M. Norrish, and R. Boulton. The PROSPER Toolkit. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of LNCS, pages 78–92, Springer-Verlag, 2000.
6. K. Fisler and S. Johnson. Integrating Design and Verification Environments Through A Logic Supporting Hardware Diagrams. In *Proc. of IFIP Conference on Hardware Description and Their Applications*, Chiba, Japan, August 1995.
7. M. Gordon. Combining Deductive Theorem Proving with Symbolic State Enumeration. *21 Years of Hardware Formal Verification*, December 1998. Royal Society Workshop to mark 21 years of BCS FACS.
8. M. Gordon. Reachability Programming in HOL98 Using BDDs. In *Theorem Proving and Higher Order Logics*, volume 1869 of LNCS, pages 179–196, Springer-Verlag, 2000.
9. M. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
10. J. Hurd. Integrating Gandalf and HOL. In *Theorem Proving in Higher Order Logics*, volume 1690 of LNCS, pages 311–321, Springer-Verlag, 1999.
11. J. Joyce and C. Seger. The HOL-Voss System: Model-Checking inside a General Purpose Theorem-Prover. In *Higher Order Logic Theorem Proving and Its Applications*, volume 780 of LNCS, pages 185–198, Springer-Verlag, 1994.
12. S. Kort, S. Tahar, and P. Curzon. Hierarchical Formal Verification Using a Hybrid Tool. *Software Tools for Technology Transfer*, 4(3):313–322, May 2003.
13. M. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

14. T. Mhamdi. On the embedding of Multiway Decision Graphs in HOL. Master's thesis, Electrical and Computer Engineering Department, Concordia University, Canada, 2003.
15. R. Mizouni. Linking HOL Theorem Proving and MDG Model Checking. Master's thesis, Electrical and Computer Engineering Department, Concordia University, Canada, 2002.
16. S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In *Automated Deduction*, volume 607 of LNCS, pages 748–752, Springer-Verlag, 1992.
17. V. Pisini. Integration of HOL and MDG for Hardware Verification. Master's thesis, Electrical and Computer Engineering Department, Concordia University, Canada, 2000.
18. S. Rajan, N. Shankar, and M. K. Srivas. An Integration of Model Checking with Automated Proof Checking. In *Computer Aided Verification*, volume 939 of LNCS, pages 84–97, Springer-Verlag, 1995.
19. K. Schneider and D. Hoffmann. A HOL Conversion for Translating Linear Time Temporal Logic to ω-automata. In *Theorem Proving in Higher Order Logics*, volume 1690 of LNCS, pages 255–272, Springer-Verlag, 1999.
20. Y. Xu. Model Checking for a First-Order Temporal Logic Using Multiway Decision Graphs. PhD thesis, Computer Science Department, University of Montreal, Canada, 1999.
21. H. Xiong, P. Curzon, S. Tahar, and A. Blandford. Formally Linking MDG and HOL Based on a Verified MDG System. In *Integrated Formal Methods*, volume 2335 of LNCS, pages 205–224, Springer Verlag, 2002.