# Embedding Multiway Decision Graphs in HOL

Tarek Mhamdi and Sofiène Tahar

Electrical and Computer Engineering Department

Concordia University, Montreal, Canada

Email: {mhamdi, tahar}@ece.concordia.ca

**Technical Report**

February 2004

**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 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 different state exploration techniques within HOL such as MDG reachability analysis, equivalence and model checking.

**Keywords**: Theorem Proving, Model Checking, Decision Procedure, HOL, MDG

# Contents

## 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 [15]. In general, 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 [13]: *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 [16]. On the other hand, 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 sub-goals generated by the theorem prover from a large system. The starting point of the second approach is a state-of-the-art checker to which we add proving rules to hopefully extend the verification to complete systems. Finally, the third approach, which is the one we adopted 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 [10]. Our decision diagram data structure is the *Multiway Decision Graphs* (MDGs) [5], which we integrate in HOL as a built-in datatype. The logic underlying MDGs will be embedded as a theory that provides the tools to specify the verification problem in the logic supported by the MDGs. The specification will consist of a set of HOL formulae that can be represented by their correspondent MDGs. Operations over these formulae will be viewed as MDG operations over their respective graphs. An MDG package will, then, be 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 gives some preliminaries on HOL and MDG. Section 3 reviews some related work. Section 4 describes the embedding of the logic underlying the MDGs in HOL. Section 5 shows how HOL is linked to the MDG package. In Section 6, we describe the embedding of the reachability analysis procedure. Section 7 and 8 illustrate the use of the embedding in the implementation of state-exploration algorithms. In Section 9 we present an application to illustrate our approach. Section 10 concludes the paper and Section 11 gives some future research directions.

## 2 Preliminaries

### 2.1 The HOL Theorem Prover

The HOL system [10] is a general purpose theorem prover based on high-order logic. It supports both forward and goal-directed backward proofs in a natural-deduction-style calculus. The user interacts with HOL through the functional metalanguage ML. The system is guided by applying *tactics* to proof obligations. A tactic corresponds to a high-level proof step and automatically generates the sequence of elementary inferences necessary to justify the step.

A notable aspect of the system is that user-defined tactics cannot compromise the soundness of a proof because the basic inferences operate on proof states. The results are safe and the user can have great confidence since the most primitive rules are used to prove a theorem. HOL system also has automatic recursive type definitions, structural induction tools and rewriting tools.

The set of types, type operators, constants, and axioms available in HOL are organized in the form of theories. There are two built-in primitive theories, *bool* and *ind*, for Booleans and individuals, respectively. Other important theories, which are arranged in a hierarchy, have been added to axiomatize lists, products, sums, numbers, primitive recursion, and arithmetic. On top of these, users are allowed to introduce application-dependent theories by adding relevant types, constants, axioms, and definitions.

Verification tasks in the HOL system can be set in a number of different ways. The most common one is to prove that an implementation, described structurally, implies or is equivalent to, a behavioral specification. The application of the HOL system can be found in hardware verification, reasoning about security, verification of fault-tolerant systems, reasoning about real-time systems, etc. It is also used in compiler verification, program refinement calculus, software verification, modelling concurrency and automata theory. HOL allows the use of hierarchical verification methodology wherein the modules are divided in sub-modules and even the sub-modules are divided until the lowest implementation level is reached. Each sub-module is verified, and its result is used to verify the other sub-modules as needed. To complete a verification, however, a very deep understanding of the internal structure of the design is required, as it is a white-box approach.

### 2.2 Multiway Decision Graphs

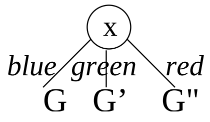
Binary Decision Diagrams are used to represent, canonically, Boolean functions. Consider a BDD $G$ with a root node labelled $x$ and subgraphs $G'$ and $G''$. If $G'$ and $G''$ represent the formulae $P'$ and $P''$, respectively, then $G$ is viewed as representing the formula $P$:

$$((\neg x) \wedge P') \vee (x \wedge P'') \tag{1}$$

However, it can also be viewed as representing the formula

$$((x = 0) \wedge P') \vee ((x = 1) \wedge P'') \tag{2}$$

This suggests a generalization of the notion of decision graph: there is no need for $x$ to only range over the set $\{0,1\}$. Furthermore, there is no need for the labels of the edges to exhaustively denote all the possible values of $x$. For example, $x$ could range over $\{$*blue, green, yellow, red*$\}$, and there could be, say, only three edges issuing from the root, as in the following graph:

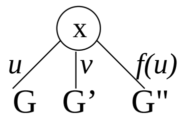


If $G$, $G'$, and $G''$ represent the formulae $P$, $P'$, and $P''$, respectively, then this graph could represent the formula

$$((x = \mathit{blue}) \wedge P) \vee ((x = \mathit{green}) \wedge P') \vee ((x = \mathit{red}) \wedge P''). \tag{3}$$

When $x$ denotes *yellow*, this formula is simply a false sentence. Finally there is no need for the edges to be mutually exclusive.

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. For example, if $x$, $u$, and $v$ are variables of abstract sort $\alpha$, $f$ is a function symbol of type $\alpha \to \alpha$, and $G$, $G'$, and $G''$ represent $P$, $P'$, and $P''$, respectively, then the graph



represents the formula

$$((x = u) \wedge P) \vee ((x = v) \wedge P') \vee ((x = f(u)) \wedge P''). \tag{4}$$

The above observations lead to the following preliminary definition:

A *Multiway Decision Graph* (MDG) is a finite directed acyclic graph $G$ where the leaf nodes are labelled by formulae, the internal nodes are labelled by terms, and the edges issuing from an internal node $N$ are labelled 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 labelled by a formula $P$, then $G$ represents $P$; (ii) if $G$ has a root node labelled $A$ with edges labelled $B_1, ..., B_n$ 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* [5] turns MDGs into *canonical representations* that can be manipulated by efficient algorithms. More details on MDG are described in the sections to follow.

## 3 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 Diagram, (BDD) [3] based symbolic algorithms in the Hol98 proof assistant. It provides intimate combinations of deduction and algorithmic verification. They use a small kernel of ML [11] functions to convert between BDDs, terms and theorems. Their work was applied to perform reachability programming in Hol98.

A pioneering work in the area is the one of Joyce and Seger [12] combining HOL and the symbolic trajectory evaluation (STE) tool VOSS. HOL-VOSS presents a mathematical link between the specification language of the VOSS system and the specification language of HOL. A tactic, VOSS_TAC, was implemented as a remote function. It 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. If this is the case, the assertion will be turned to a HOL theorem. The early experiment 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 VossProver was developed. As a continuation of HOL-VOSS, Aagaard *et al.* [1] developed the Voss-ThmTac system combining 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.* [21] described an approach where a BDD based model checker for the propositional $\mu$-calculus has been used as a decision procedure within the framework of the PVS [19] proof checker. They used $\mu$-calculus as a medium for communicating between PVS and the model checker. It was formalized by using the higher-order logic of PVS. The temporal operators are given the customary fixpoint definitions using the $\mu$-calculus. These expressions were translated to the form required by the model checker. The latter was then used to verify the subgoals generated within PVS.

Schneider and Hoffmann [22] linked the SMV model checker [17] to HOL using PROSPER. They embedded the linear time temporal logic (LTL) in HOL and translated LTL formulae into equivalent $\omega$-Automata, a form that can be reasoned about within SMV. The translation is completely implemented by means of HOL rules. On successful model checking, the results are returned to HOL and turned to theorems. The deep embedding of the SMV specification language in HOL allows LTL specifications to be manipulated in HOL.

In [14], [20] and later [18] a hybrid tool and a methodology tailored to perform hierarchical hardware verification have been developed by the *Hardware Verification Group* of *Concordia University*. They integrate the HOL theorem prover to the MDG equivalence checker and later to the MDG model checker. The work is done within the proof system but using the specification style of the automated verification tool. The HOL-MDG tool is used to verify that a structural specification of hardware implementation implies its behavioral specification. They perform the equivalence checking within the MDG tool by applying a HOL tactic MDG_EQ_TAC. This latter mainly generates the MDG required files and ensures the interaction with the MDG equivalence checker. If the design is large enough to cause state explosion, and since the description model is written in a hierarchical way, 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. Similarly, they provide a way to express temporal properties inside the theorem prover and support the full properties specification language of MDG by introducing abstract datatypes and uninterpreted functions. A HOL tactic, called MDG_MC_TAC is used to perform model checking. It supports hierarchical verification and model reduction.

While [14, 18, 20] describe systems integrating two stand-alone tools, namely, HOL and an external MDG tool, the work described here is not intended to use an external tool to verify subgoals. Instead MDGs are 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, 12, 22] 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 [21] uses the $\mu$-calculus as a medium for communicating between the theorem prover and the model checker. It is a shallow embedding of stand-alone tools 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. On the other hand, MDGs represent a subset of first-order terms allowing the abstract representation of data and hence raising the level of abstraction.

## 4 Embedding The MDG Logic in HOL

Multiway Decision Graphs are intended to represent Abstract State Machines (ASM) [5], an abstract description of state machines based on a many-sorted first order logic with a distinction between abstract and concrete sorts.

### 4.1 MDG Sorts

Concrete sorts have enumerations, while abstract sorts do not. An enumeration is a finite set of constants. This is embedded in HOL as follows:

- `Concrete_Sort = Concrete_Sort of string ⇒ string list;`

It declares a constructor called *Concrete_Sort* that 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:

```
val state = Define ‘state = Concrete_Sort ‘‘state’’ [ stop; run ]‘;
```

- `Abstract_Sort = Abstract_Sort of ’a;`

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:
val *alpha* = Define '*alpha* = Abstract_Sort "*alpha*"';
To determine whether a sort is concrete or abstract, we use predicates over the sorts constructors called *IsConcreteSort* and *IsAbstractSort*, where "_" means "don't care".

```
(IsConcreteSort (Concrete_Sort _ _)  = T) /\ (IsConcreteSort _ = F);
(IsAbstractSort (Abstract_Sort _)    = T) /\ (IsAbstractSort _ = F);
```

These predicates will be used for instance to determine the sort of a variable or a function symbol.

The vocabulary 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 $\alpha_1 \times ... \times \alpha_n \to \alpha_{n+1}$. If $\alpha_{n+1}$ 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 $\alpha_{n+1}$ is concrete while at least one of $\alpha_1 ... \alpha_n$ is abstract, then we refer to $f$ as a *cross-operator.*

## 4.2 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 labelling an edge in the graph. In our embedding, a primary abstract variable will be declared using the *Abstract_Var* constructor while a secondary variable will be declared using the *Secondary_Var* constructor.

- `Concrete_Var = Concrete_Var of string` ⇒ `Concrete_Sort;`

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

```
val x = Define `x = Concrete_Var ``x'' state`;
```

- `Abstract_Var = Abstract_Var of string` ⇒ `Abstract_Sort;`

An abstract variable $y$ with name "$y$" and sort *alpha* is declared using:

```
val y = Define `y = Abstract_Var ``y'' alpha`;
```

- `Secondary_Var = Secondary_Var of string` ⇒ `Abstract_Sort;`

The *Secondary_Var* constructor is similar to the *Abstract_Var* constructor. For example:

```
val y1 = Define `y1 = Secondary_Var ``y1'' alpha`.
```

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

## 4.3 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. In HOL they are declared as follows:

- `Individual_Const = Individual_Const of string;`

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 defined in order to be able to declare the sort state.

```
val stop = Define `stop = Individual_Const ``stop''`;
val run = Define `run = Individual_Const ``run''`;
```

- `Generic_Const = Generic_Const of string ⇒ Abstract_Sort;`

Having declared "*alpha*" as abstract sort, we can declare generic constants of that sort. Say *a* is a generic constant of sort *alpha*.

```
val a = Define `a = Generic_Const ``a'' alpha`;
```

To check whether a constant is an individual constant or an abstract generic constant, we define two predicates, *IsIndividualConstant* and *IsGenericConstant*.

## 4.4 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.

- `Cross_Function = Cross_Function of string ⇒ Secondary_Var list ⇒ Concrete_Sort;`

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.

```
val bool  = Define `bool = Concrete_Sort "bool" ["0";"1"]`;
val y1    = Define `y1 = Secondary_Var ``y1'' alpha`;
val y2    = Define `y2 = Secondary_Var ``y2'' alpha`;
val equal = Define `equal = Cross_Function "equal" [y1;y2] bool`;
```

- `Abstract_Function=Abstract_Function of string ⇒ Secondary_Var list ⇒ Abstract_Sort;`

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

```
val max = Define ‘max = Abstract_Function ‘‘max’’ [y1;y2] alpha‘;
```

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

## 4.5 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.

```
Hol_datatype ‘MDG_Term =
  Individual_Const  of string => Concrete_Sort
| Generic_Const     of string => ’a Abstract_Sort
| Concrete_Var      of string =>    Concrete_Sort
| Abstract_Var      of string => ’a Abstract_Sort
| Cross_Function    of string=>(’a Secondary_Var)list=> Concrete_Sort
| Abstract_Function of string=>(’a Secondary_Var)list=>’a Abstract_Sort‘
```

## 4.6 Well-formed MDG Terms

For BDDs to be canonical, they have to be reduced and ordered. Similarly, MDG require certain well-formedness conditions to represent canonically the MDG terms. The set of well-formed terms that can be represented canonically by the MDGs is called the set of *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 $\alpha$ containing no variables other than elements of $U$, and $a$ is an individual constant in the enumeration of $\alpha$, or
   - $u = a$, where $u \in U$ is a variable of concrete sort $\alpha$ and $a$ is an individual constant in the enumeration of $\alpha$, or
   - $v = a$, where $v \in V$ is a variable of concrete sort $\alpha$ and $a$ is an individual constant in the enumeration of $\alpha$, or
   - $v = A$, where $v \in V$ is a variable of abstract sort $\alpha$ and $A$ is a term of type $\alpha$ 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. The condition 3 is less stringent than it seems. In practice, one can introduce an additional dependant variable $u$ and add an equation $v = u$ to a disjunct where an abstract $v$ is missing.

For example, condition 1 is embedded in HOL and checked using the function *Well_formedTerm* that, recursively, calls *Well_formedEQ* to check the well-formedness of an equation.

```
fun 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);
```

## 5 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 tools to build and manipulate the graph representations of these formulae. This platform will consist of ML functions that call an MDG package[1] 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 [11]. The latter computes the result (MDG graph) and then writes it in a file "*mdghol.ch*". Using the function *ReadMdgOutput*, the result is retrieved.

### 5.1 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*. It 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

[1]We provide a lifted version of the MDG package with which we are able to call internal MDG functions.

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.

## 5.2 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 $Q$.

# 6 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.

## 6.1 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 ML function *ComputeNext* representing the set of next states, computed from $B$ with respect to $R$, is defined by:

$$\mathit{ComputeNext}(G_I\ G_B\ G_R) = \mathit{RelP}(G_I\ G_B\ G_R\ Q\ \eta).$$

where, $G_I, G_B$ and $G_R$ are the MDG representations for $I, B$ and $R$, respectively. $Q$ is the set of input and state variables over which the MDG is quantified. $\eta$ is the renaming substitution. $B$ can be the set of initial states as well as the set of states already reached by the machine.

## 6.2 Computing Outputs

The set of outputs corresponding to a set of initial states and inputs, with respect to an output relation $O$, is represented by the ML function *ComputeOutputs* below, where $G_O$ is the MDG representation of $O$.

$$ComputeOutputs(G_I\ G_B\ G_O) = RelP(G_I\ G_B\ G_O\ Q)\ \text{“\_”}.$$

For every state of the machine, and a set of data inputs, corresponds a set of output values. These will be used to check an invariant.

## 6.3 Computing Frontier Set

The frontier set is the set of newly visited states. If $V$ represents the set of states already visited, $V_n = ComputeNext(G_I\ V\ G_R)$ 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*.

$$ComputeFrontier(V_n\ V) = PbyS(V_n\ V).$$

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).

## 6.4 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[2]. Using the solutions proposed in [2], the set of reachable states is computed and represented by the function, *ComputeReachable*, defined by[3]:

```
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;
```

*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

[2]This is called the non-termination problem which was tackled in [2] using various heuristics.

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

to $B$ (the initial state), and the sets of next-states are computed until reaching a fixpoint characterized by an empty frontier set.

## 7 Invariant and Model Checking in HOL

### 7.1 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. The invariant checking algorithm is implemented 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.

The function *InvariantChecking*, 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
    // generates the renaming substitution function
    $K = 0$, $S = G_{IN}$, $R = G_{IN}$
    loop
      $K = K + 1$
      // generates the input graph $G_{IK}$
      $O_S$ = **ComputeOutputs** $G_{OR}$ $R$ $G_{IK}$
      if (**PbyS** $O_S$ $G_{Inv}$) $\neq F$ 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;
```

## 7.2 Model Checking

Similarly, MDG temporal operators can be implemented in HOL for model checking. In the following we present how the operator **AF** on a first-order property formula P [23] is embedded.

```
Check_AF $T_R$ $I_N$ $Inputs$ $States$ $NxStates$ $P$ =
    // builds the MDG representations $G_{TR}$, $G_{IN}$, $G_P$
    // generates the renaming substitution function
    $K = 0$, $\Sigma = F$, $C = G_{IN}$
    // $\Sigma$ contains sets of states not satisfying $P$
    loop
        $Q$ = **ComputeFrontier** $C$ $G_P$
        // removes states satisfying $P$
        if $Q = F$ then return success
        if **ComputeFrontier** $\Sigma$ $Q \neq \Sigma$ then return failure
        $\Sigma$ = **Disj** $\Sigma$ $Q$
        $K = K + 1$
        $C$ = **ComputeNext** $G_{IN}$ $Q$ $G_{TR}$
    end loop
end Check_AF;
```

# 8 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*.

## 8.1 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*.

```
fun EquivCheck order t1 t2 =
    let   val s1 = termToMdg order t1
          val s2 = termToMdg order t2
    in
```

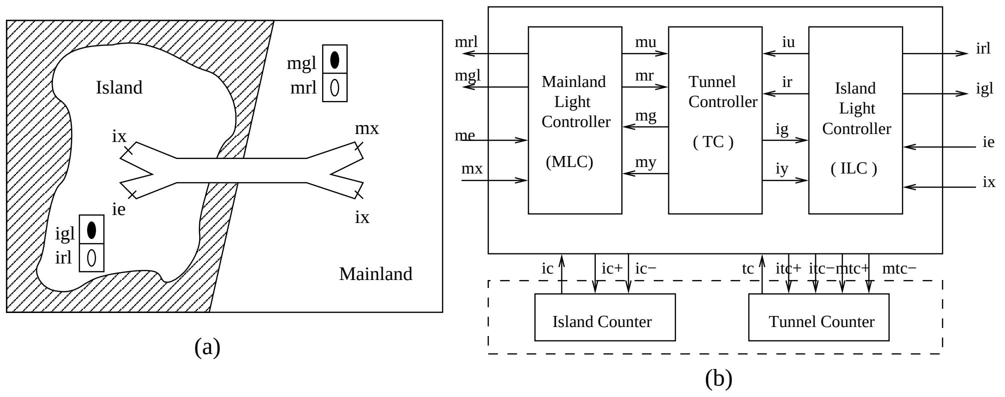


Figure 1: The Island Tunnel Controller

```
        (s1=s2)
    end;
```

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.

### 8.2 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*.

```
fun Tautology order t =
    let val s = termToMdg order t
    in
        isTrue s
    end;
```

## 9 Application

In this section we illustrate our embedding with the case study of the Island Tunnel Controller. The ITC was originally introduced by Fisler and Johnson [6]. It controls the vehicles traffic in a one-lane tunnel connecting the mainland to a small island, as shown in Figure 1(a).

The ITC is specified using three communicating controllers and two counters as shown in Figure 1(b). We list below some examples properties that we verified.

## 9.1 Properties

### 9.1.1 Property 1

Our ITC model must respect the safety property stating that the lights on the island side and the mainland side cannot be green at the same time. This is specified by the following invariant.

```
val P1 = ( (igl = true)  /\ (mgl=false) )  \/
         ( (igl = false) /\ (mgl=true)  )  \/
         ( (igl = false) /\ (mgl=false) );
```

### 9.1.2 Property 2

If the light in the mainland side is green, it stays green as long as the tunnel is not requested from the island. This is a faulty behavior since the number of allowed cars on the island side is limited. Checking this property returns *false* and a counter-example can be generated.

```
val P2      = ( (mgl=true) );
val initial = ( (ms=green) /\ (my=false) );
```

### 9.1.3 Property 3

This property corrects property 2 by adding the island capacity constraint. If the light on the mainland side is green, it stays green as long as the tunnel is not requested from the island and the number of allowed cars (ic), which is an abstract variable, is not exceeded.

```
val P3      = ( (mgl=true) );
val initial = ( (ms=green) /\ (my=false) /\ (lessn(ic)=true) );
```

## 9.2 Experimental Results

To verify the mentioned properties, we used the invariant checking procedure. For each property we used only the transition relations and the variables involved in the property (specified manually). 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 by reducing the CPU time and the memory usage compared to verifying the whole system. The CPU time represents the system time to perform the reachability analysis and also includes the time to translate the HOL specification to MDG files. The verification results, run on an Ultra2 Sun workstation with 296Mhz CPU and 768MB memory, are reported in Table 1. A "*" beside a property means that this latter failed in the invariant checking.

## 10 Conclusions and Future Work

Expertise and user guidance need is a major problem for applying theorem proving on, even, the most trivial systems. On the other hand, state-exploration techniques suffer from the

| **Property** | **CPU**$_s$ | **Memory**$_{MByte}$ |
|---|---|---|
| Property1 | 101.9 | 0.220 |
| *Property2 | 52.8 | 0.013 |
| Property3 | 54.9 | 0.077 |
| Property4 | 44.3 | 0.020 |
| Property5 | 72.0 | 0.058 |
| Property6 | 44.8 | 0.035 |
| Property7 | 63.9 | 0.039 |
| Property8 | 64.2 | 0.039 |
| Property9 | 45.4 | 0.035 |

Table 1: Property Checking Results using *InvariantChecking*

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 that 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 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 in that it eliminates the state explosion problem introduced by the datapath.

The MDGs are embedded in HOL as a built-in datatype. Operations over the MDGs are interfaced to HOL functions allowing the manipulation of graphs rather then their correspondent HOL terms. Using the embedding of the logic underlying the multiway decision graphs in HOL, the verification problem is specified as a set of well-formed directed formulae that can be represented canonically by well-formed MDGs. This is made possible thanks to the lifted MDG package that we provided and interfaced to HOL resulting in a platform of functions to represent terms by their correspondent MDGs and manipulate them.

The platform, we provide, allowed us to develop state-exploration algorithms inside HOL like the reachability analysis, model checking and the invariant checking procedures. The transition and output relations are written as HOL terms. They are translated to their corresponding MDGs and then reachability analysis is performed. The state machines we consider are the abstract state machines which raises the level of abstraction of the problem specification. We also developed decision procedures based on the multiway decision graphs allowing the equivalence checking and tautology checking of a certain subset of HOL terms automatically. Finally we illustrated our approach by considering the Island Tunnel Controller example for which we verified a number of safety properties.

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 this embedding and its use for formal verification. For instance, MDG canonicity can also 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 term. The Embedding can be used for the formal proof of the soundness of the MDG algorithms. A similar work was done in [4] to verify a SPIN model checking algorithm. Finally, the embedding can be enhanced by using the LCF style. In this case, an MDG representation for a HOL term cannot be constructed by using the *termToMdg* function, instead, it is derived from inference rules, corresponding to MDG operators, and the trivial MDGs representing simple equations. This restricts the scope of soundness to single operators which are easy to get right [9].

## 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*, 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] R. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[4] 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.

[5] 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.

[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, August 2000.

[9] M. Gordon. Holbddlib Version 2, Documentation. Technical report, Computer Laboratory, Cambridge University, U.K., March 2002.

[10] M. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[11] R. Harper. *Introduction to Standard ML.* School of Computer Science, Carnegie Mellon University, Pittsburgh, USA, 1993.

[12] 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.

[13] C. Kern and M. Greenstreet. Formal Verification in Hardware Design: A Survey. *ACM Transactions on Design Automation of Electronic Systems*, 4:123–193, 1999.

[14] 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.

[15] T. Kropf. *Introduction to Formal Hardware Verification.* Springer Verlag, 1999.

[16] R. P. Kurshan. Formal Verification in a Commercial Setting. In *Proc. of Design Automation Conference*, pages 258–262, Anaheim, California, USA, June 1997.

[17] M. McMillan. *Symbolic Model Checking.* Kluwer, 1993.

[18] R. Mizouni. Linking HOL Theorem Proving and MDG Model Checking. Master's thesis, Electrical and Computer Engineering Department, Concordia University, Canada, 2002.

[19] 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.

[20] V. Pisini. Integration of HOL and MDG for Hardware Verification. Master's thesis, Electrical and Computer Engineering Department, Concordia University, Canada, 2000.

[21] 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.

[22] K. Schneider and D. Hoffmann. A HOL Conversion for Translating Linear Time Temporal Logic to $\omega$-automata. In *Theorem Proving in Higher Order Logics*, volume 1690 of LNCS, pages 255–272. Springer-Verlag, 1999.

[23] Y. Xu. *Model Checking for a First-Order Temporal Logic Using Multiway Decision Graphs.* PhD thesis, Computer Science Department, University of Montreal, Canada, 1999.