On the Embedding of Multiway Decision Graphs in HOL

Tarek Mhamdi

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science at

Concordia University

Montreal, Quebec, Canada

August 2003

Your file Votre référence
ISBN: 0-612-83872-2
Our file Notre référence
ISBN: 0-612-83872-2

ABSTRACT

On the Embedding of Multiway Decision Graphs in HOL

Tarek Mhamdi

The increasing complexity of hardware systems requires more and more sophisticated methods of verification. While model checking suffers from the state space explosion problem, theorem proving is quite tedious and impractical for verifying complex designs. In this thesis, we propose 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.

To My Family

ACKNOWLEDGEMENTS

I have been very fortunate to have Dr. Sofiène Tahar as my supervisor. I am deeply grateful for his strong support and encouragement through out my Master's studies. His expertise and competent advice have shaped the character of my research.

I would like to thank the examination committee members for reviewing my thesis and giving me invaluable feedback. I am particularity grateful to Dr. Othmane Ait-Mohamed for his fruitful discussions.

My colleagues from the Hardware Verification Group (HVG) provided a nice atmosphere for discussions and research, I thank them for all their support and valuable hints.

I also wish to thank the University Mission of Tunisia in North America for financing my studies facilitating me to actively concentrate on research.

I would like to reserve my deepest thanks for my family for their perpetual love.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

ASM	Abstract State Machines
ATM	Asynchronous Transfer Mode
CTL	Computational Tree Logic
FSM	Finite State Machine
HDL	Hardware Description Language
HOL	Higher-Order Logic
ILC	Island Light Controller
ITC	Island Tunnel Controller
LTL	Linear Temporal Logic
ML	Meta Language
MLC	Main Land Controller
PVS	Prototype Verification System
ROBDD	Reduced Ordered Binary Decision Diagram
RTL	Register Transfer Lever
SMV	Symbolic Model Verifier
TC	Tunnel Controller
VIS	Verification Interacting with Synthesis
VLSI	Very Large Scale Integration

Chapter 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 [22]. In general, the 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 [20]: *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 [23]. 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.

Ideally, one would like to combine the strengths of both techniques resulting in a hopefully automatic theorem prover. This is not likely to be practical in the foreseeable future, so various compromises are being explored. They can be summarized in either adding a layer of theorem proving on top of existing model checkers, to enable large problems to be deductively decomposed into smaller pieces that can be checked automatically, or adding checking algorithms to theorem provers so that subgoals can be verified automatically and counter-examples found.

Motivated by a desire to combine the expressiveness and scalability of theorem proving and the automation and efficiency of state-exploration based techniques, we developed a platform of state-exploration algorithms inside the HOL proof system [16]. Our decision diagram data structure is the *Multiway Decision Graphs* (MDGs) [9] which we integrate in HOL as a built-in datatype.

1.1 Formal Verification Techniques

Formal verification [20] consists of formally establishing that an implementation satisfies a specification. To classify the various approaches, we first look at the three main aspects of the problem [22]: the implementation, the specification and the relationship between them.

An implementation is a description of the actual hardware design that is to be verified. It usually can be described at different levels of abstraction: circuit level, switch level, gate level or register-transfer level. Different abstraction levels often result in different verification methods. A method that is good at one level may become cumbersome at another one. Another important issue with the implementation is the class of circuits we wish to verify, i.e., whether it is combinational/sequential, synchronous/asynchronous, pipelined or parametrized hardware. These variations may require different approaches (though not mutually exclusive).

There are two verification paradigms depending on the two different kinds of specifications:

1. Verification of behavioral equivalence: It intends to prove that an implementation is behaviorally equivalent to the specification which is a description of the intended/required behavior of a hardware design. This can be applied for the proof of implication.

2. Property verification: It intends to prove that the implementation is a model

of the specification which consists of the set of properties to be satisfied.

The above two styles of verification are not mutually exclusive, in fact, they are somehow complementary. It can be useful to verify important properties as well as to verify the behavioral equivalence or logical implication of an implementation against a specification.

1.1.1 Theorem Proving

Theorem proving is an approach where both the system and its desired properties are expressed as formulae in some mathematical logic. This logic is defined by a formal system, called *proof system* or *calculus*, which defines a set of *axioms* and a set of inference rules. Theorem proving is the process of deriving a proof from the basic axioms of the system. 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.

Proof styles are often characterized as "forward" or "backward". A forward proof starts with the axioms and assumptions; inferences are then applied until the desired theorem is proven. A backward proof starts with the theorem as a goal and applies the inverses of inferences rules to reduce the theorem to simpler intermediate goals. Sufficiently simple goals are discharged by matching axioms or assumptions or by applying built-in decision procedures.

Many theorem-proving systems have been implemented, and many have been used for hardware verification, including HOL [16], ISABELLE [27], and PVS [26]. 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. In the next chapter, we will overview the HOL theorem proving system, which we intend to use in this thesis.

1.1.2 Model Checking

Model checking is a technique that relies on building a model of a system and checking that a desired property holds in that model by exploring a state space search in that model. Model checking is mainly used in hardware and protocol verification. Temporal model checking, is a technique developed in the 1980s by Clarke and Emerson [8] and by Queille and Sifakis [30]. In this approach, specifications are expressed in a temporal logic [29] and systems are modelled as finite state systems. An efficient search procedure is used to check if a given finite state transition system is a model for the specification.

The model checking technique described by Clarke [8] requires that the entire state transition graph be constructed. Thus, the space requirements are at least linear in the size of the model's reachable state space. However, the latter is often exponential in the number of state holding elements (e.g., latches) of a design. For instance, a device with only two 32-bit registers would already have 10^{20} states [6]. An alternative to explicit enumeration is to use a symbolic representation.

Binary Decision Diagrams

Binary decision diagrams (BDDs) are data structures for representing Boolean functions. Bryant [5] introduced the BDD in its current popular representation, although the general idea have been floating around for quite some time (e.g., as branching programs in the theoretical computer science literature).

BDDs have several useful properties. First, many common functions have small BDDs. In addition BDDs are easy to manipulate. We can evaluate a function in linear time in the number of variables. We can existentially or universally quantify (Boolean) variables of a function in time quadratic in the size of the BDD. Finally, once we fix the order in which the variables appear, the BDD is a canonical representation for the Boolean function. Thus function comparison, including special cases tautology and satisfiability, become trivially easy.

BDDs are a practically efficient representation of Boolean functions. Many variations of BDDs were proposed to avoid the state-explosion problem. Multiway Decision Graphs (MDG), [9] are a special kind of decision diagrams that subsumes BDDs and extends them by canonically and compactly representing a subset of first-order functions.

Symbolic Model Checking

Symbolic model checking was initially explored by Coudert, Madre and Berthet [10], and independently by McMillan [24] and by Bose and Fisher [3]. The underlying idea common to these approaches is the use of symbolic Boolean representations for the sets of states and transition functions (or relations) of a sequential system, in order to avoid building its global state-transition graph explicitly. Efficient symbolic Boolean manipulation techniques are then used to evaluate the truth of temporal logic formulae with respect to those models. Symbolic representations (like BDDs) allow the regularity in state-space of some circuits (e.g., datapaths) to be captured succinctly, thus facilitating verification of much larger circuits compared to the explicit state enumeration techniques, as shown by Burch *et al.* [6].

1.2 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 [13, 14] system developed by Gordon. HolBdd consists of a platform allowing the programming of Binary Decision Diagram (BDD) [5] based symbolic algorithms in the Hol98 proof assistant. It provides intimate combinations of deduction and algorithmic verification. They use a small kernel of ML [17] functions to convert between BDDs, terms and theorems. Their work was applied to perform reachability programming in Hol98.

A similar work was the pioneering work of Joyce and Seger [19] 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.* [31] 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 [26] proof checker. They used μ-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 fix-point definitions using the μ-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.

Hurd [18] used PROSPER [11] 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 [32] linked the SMV model checker [24] 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. 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 [28], and later [21] 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. 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 try to do 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.

An extention of the work above was done within the same group to link HOL and the MDG model checker [25]. The approach adopted is similar to [21], however, instead of considering the full behavior of the system, only properties are checked, hence reducing the verification complexity. To do so, they provide a way to express temporal properties inside the theorem prover. Besides they support the full input language of MDG by introducing abstract datatypes and uninterpreted functions. The verification is done using a HOL tactic called MDG_MC_TAC and also supports hierarchical verification and model reduction.

While [21, 25, 28] 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, 18, 19, 32] 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 [31] uses the μ-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 [13, 14]. Our work, however, deals with embedding MDGs rather than BDDs. In fact, BDDs are widely used in state-exploration methods. However 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.

1.3 Scope of the Thesis

In this thesis, we propose a platform of state-exploration algorithms, based on the multiway decision graphs, inside a proof assistant, namely HOL. We propose to embed the logic underlying MDGs into HOL. The normal operations over HOL terms are interpreted as MDG operations. Compared to related research [13, 14] we raise the level of abstraction at which the problem is stated and explore state-exploration techniques at a higher abstraction level. Our embedding is based on abstract description of state machines (ASM) [9] where a data value is represented by a single variable of abstract type, rather than by a vector of boolean variables and a data operation is represented by an uninterpreted or partially interpreted function symbol. The state explosion problem caused by descriptions of large datapaths at the Boolean logic level is then avoided.

To tightly integrate a platform of MDG based algorithms inside HOL we propose to embed the MDG data structure as a built-in datatype of HOL. The logic underlying MDGs will be available in the *HolMdgTheory*. This theory 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. Hence, the lifted MDG package will be used to build the graph representation of a HOL formula and to allow the manipulation of graphs rather than HOL terms.

The MDG data structure and operators, once available inside the theorem prover, can be used to automate parts of the verification problem or even to write state enumeration algorithms like reachability analysis or model checking.

1.4 Contributions of the Thesis

The purpose of our work is to intimately combine the HOL proof system and implicit state explorations using MDGs. The main contributions we report in this thesis are:

1. The embedding of the formal logic underlying the abstract state machines inside HOL. This will be used to specify the verification problem as first-order formulae that can be represented by MDGs.

2. The introduction of the notion of well formed terms in HOL. This is the subset of the terms that can be represented canonically by MDGs.

3. The development of a lifted version of the MDG package using an available version allowing to communicate interactively with HOL.

4. The development of an ML interface that is responsible of calling the lifted MDG package functions corresponding to the operations over HOL terms.

5. The implementation of some state-exploration applications inside HOL based on MDGs. We implemented the reachability analysis and used it for model checking and the invariant checking procedure.

1.5 Outline of the Thesis

The rest of this thesis is organized as follows:

In Chapter 2, we overview the basics of the HOL theorem prover. We also describe in more details the class of Abstract State Machines (ASM) and decision diagrams (MDGs) that we are using.

In Chapter 3, we present the formal logic underlying MDGs and its embedding inside HOL. Well-formedness conditions will also be discussed.

In Chapter 4, the lifted version of the MDG package will be presented and the linking between HOL and MDG discussed. We will discuss how the MDG data structure and its basic operators are made available in HOL.

In Chapter 5, we show how our embedding can be used to implement state-exploration algorithms inside HOL. We illustrate this by implementing the reachability analysis inside HOL. This is then used for applications like model checking and invariant checking.

In Chapter 6, we consider the Island Tunnel Controller example as a case study for which we specified and verified a number of safety properties using the invariant checking procedure.

In Chapter 7, we conclude the thesis and outline future research directions.

Chapter 2

Preliminaries

In this chapter we will overview the HOL theorem prover as well as the Abstract State Machines (ASM) and the Multiway Decision Graphs (MDG). The description of ASM and MDG are based on material in [9] and [35].

2.1 The HOL Theorem Prover

The HOL system [16] 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.

2.1.1 Types

A HOL type can be a variable, a constant, or a compound type, which is a constant of arity n applied to a list of n types.

```
hol_type ::=  'ident                                    (type variable)
           |  bool                           (type of truth values)
           |  ind                             (type of individuals)
           |  hol_type -> hol_type                 (function arrow)
           |  ident                         (nullary type constant)
           |  hol_type ident                 (unary compound type)
           |  (hol_type, ..., hol_type) ident    (compound type)
```

Type constants are also known as type operators. They must be alphanumeric. Type variables are alphanumerics written with a leading prime ('). *bool* is the two element type of truth values. The binary operator *fun* is used to denote function types; it can be written with an infix arrow. The nullary type constant *ind* denotes an infinite set of individuals. Thus `'a -> 'b` and `(bool -> 'a) -> ind` are both well-formed types. The function arrow is right associative.

Many formalizations require the definition of new types. In HOL, such types may be specified using the invocation:

```
Hol_datatype `<spec>`
```

where `<spec>` should conform to the following grammar:

```
spec ::= [ <binding> ; ]* <binding>
binding ::= <ident> = [ <clause> | ]* <clause>
          | <ident> = <| [ <ident> : <type> ; ]* <ident> : <type> |>
clause ::= <ident>  | <ident> of [<type> => ]* <type>
```

For example, we can define a type of binary trees where the leaves are numbers as :

```
Hol_datatype ‘tree = Leaf of num | Node of tree => tree‘
```

2.1.2 Terms

Ultimately, a HOL term can only be a variable, a constant, an application, or a lambda term (to denote a function).

```
term  ::=   ident              (variable or  constant)
         |  term term                   (combination)
         |  \ident. term         (lambda abstraction)
```

In the HOL system, the usual logical operators have already been defined, including truth (T), falsity (F), negation (~), equality (=), conjunction ($\wedge$), disjunction ($\vee$), implication (==>), universal (!) and existential (?) quantification, and an indefinite description (choice) operator (@). Besides, the basis includes conditional, lambda, and "let" expressions. Thus the set of terms available is, in general, an extension of the following grammar:

```
term  ::=  term : hol_type                 (type constraint)
        |  term term                           (application)
        |  ~term                                  (negation)
        |  term = term                            (equality)
        |  term ==> term                       (implication)
        |  term \/ term                        (disjunction)
        |  term /\ term                        (conjunction)
        |  term => term | term                 (conditional)
        |  \ident ... ident.  term      (lambda abstraction)
```

```
    |   !ident ... ident.  term                       (forall)
    |   ?ident ... ident.  term                       (exists)
    |   @ident ... ident.  term                       (choose)
    |   ?!ident ... ident. term                (exists-unique)
    |   let ident = term
         [and ident=term]* in term           (let expression)
    |   T                                              (truth)
    |   F                                            (falsity)
    |   ident                           (constant or variable)
    |   (term)                           (parenthesized term)
```

Some HOL syntax examples may be found in Table 2.1. The lexical structure of term identifiers is much like that for ML: identifiers can be alphanumeric or symbolic. Variables must be alphanumeric. A symbolic identifier is any concatenation of the characters in the following list: "`#?+*/\\=<>&%@!,:;_~-`" with the exception of the keywords "`\\`", "`;`", "`=>`" and "`:`". Any alphanumeric can be a constant except the keywords "`let`", "`in`" and "`of`".

`x = T`	x is equal to true.
`!x. Person x ==> Mortal x`	All persons are mortal
`!x y z. (x==>y) /\ (y==>z) ==> x ==> z`	implication is transitive.
`!x. P x ==> Q x`	P is a subset of Q.

Table 2.1: HOL Syntax Examples

2.2 Abstract State Machines

In this section, we present a theory of abstract description of state machines in a many-sorted first order logic with a distinction of abstract and concrete sorts.

The theory provides a foundation for automated state enumeration methods, which complexity is independent of the width of the datapath.

2.2.1 Formal Logic

Syntax

The formal logic that we use is many-sorted first-order logic, with a distinction between *abstract sorts* and *concrete sorts.*

Concrete sorts have *enumerations*, while abstract sorts do not. An enumeration is a finite set of constants. A constant that appears in the enumeration is called an *individual constant.* Besides individual constants, the vocabulary consists of *generic constants*, *variables* and *function symbols* (also called *operators*). Generic constants and variables each have one sort. An individual constant, on the other hand, is treated as having multiple sorts, one for each enumeration of which it is a member. An n-ary function symbol ($n > 0$) has a type $\alpha_1 \times ... \times \alpha_n \rightarrow \alpha_{n+1}$, where $\alpha_1 ... \alpha_{n+1}$ are sorts. Generic constants can be viewed as 0-ary function symbols.

If X is a set (or "vector") of variables, we write X_{conc} and X_{abs} to denote the sets of elements of X that are variables of concrete and abstract sort, respectively. 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 generic 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.*

Semantics

An *interpretation* is a mapping ψ that assigns a denotation to each sort, constant and function symbol and satisfies the following conditions:

1. The denotation of $\psi(\alpha)$ of an abstract sort α is a non-empty set.

2. If α is a concrete sort with enumeration $\{a_1, ..., a_n\}$ then $\psi(\alpha) = \{\psi(a_1), ..., \psi(a_n)\}$ and $\psi(a_i) \neq \psi(a_j)$ for $1 \leq i \leq j \leq n$.

3. If f is a function symbol of type $\alpha_1 \times ... \times \alpha_n \rightarrow \alpha_{n+1}$, then $\psi(f)$ is a function from the Cartesian product $\psi(\alpha_1) \times ... \times \psi(\alpha_n)$ into the set $\psi(\alpha_{n+1})$. In particular, if $n = 0$ (i.e., f is a generic constant of sort α_1), $\psi(f) \in \psi(\alpha_1)$.

X being the set of variables, a *variable assignment* with domain X compatible with an interpretation ψ is a function ϕ that maps every variable $x \in X$ of sort α to an element $\phi(x)$ of $\psi(x)$. We write Φ_X^ψ for the set of ψ-compatible assignments to the variables in X. The denotation of a term and the truth or falsity of a formula under an interpretation and a compatible variable assignment are defined as usual.

We write $\psi, \phi \models P$ if a formula P denotes truth under an interpretation ψ and a ψ-compatible variables assignment ϕ to the variables that occur free in P, and $\models P$ for all such ψ, ϕ. Two formulae P and Q are *logically equivalent* iff $\models P \iff Q$.

2.2.2 Directed Formulae

Given two disjoint sets of variables U and V, a *directed formula* (DF) of type $U \to V$ 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. Every variable $v \in V$ appears as the left hand side of an equation $v = A$ in each of the disjuncts

Intuitively, in a DF of type $U \to V$, the U variables play the role of independent variables, the V variables play the role of dependent variables, and the disjuncts enumerate possible cases. In each disjunct, the equations of the form $u = A$ and $A = a$ specify a case in terms of the U variables, while the other equations specify the values of (some of the) V variables in that case. The cases need not be mutually exclusive, nor exhaustive. The condition that every abstract variable $v \in V$ must appear in every disjunct 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 v is missing.

A DF is said to be *concretely reduced* iff every A in an equation $A = a$ is a cross-term, and every A in an equation $v = A$ is a concretely reduced term. It is easy to see that every DF is logically equivalent to a concretely reduced DF, given complete or partial specifications of the concrete function symbols and concrete generic constants; the reduction can be accomplished by case splitting.

We use DFs for two distinct purposes: to represent relations (transition an output relations) and to represent sets (sets of states as well as sets of input vectors and output vectors).

2.2.3 Abstract Description of State Machines

A state machines is described using a finite set X of input variables, a finite set Y of state variables, a finite set Y' of next state variables, and a finite set Z of output variables, which are pairwise disjoint. An abstract description of the state machine, or an abstract state machine (ASM), is obtained by letting some data input, state or output variables be of an abstract sort.

The behavior of a state machine is defined by its transition and output relations, together with its set of initial states. Thus an *abstract description* of a state machine is a tuple $D = (X, Y, Z, Y', \eta, F_I, F_T, F_O)$ where:

1. X, Y, and Z are, pairwise disjoint vectors of input, state and output variables. Note that Y and Z must be disjoint. To allow for *observable state variables*, i.e., state variables that are also output variables, we let X and Y be the sets of *all* input and state variables respectively, while Z comprises only the output variables other than the observable state variables; Z can be empty.

2. Y' is the set of next-state variables, disjoint from $X \cup Y \cup Z$, and η is the function that maps each state variable to the corresponding next-state variable. We usually obtain each next-state variable by priming the corresponding state variable.

3. F_I is a DF of type $U_0 \rightarrow Y$, where U_0 is a set of abstract variables disjoint from $X \cup Y \cup Y' \cup Z$. F_I is the abstract description of the set of initial states.

4. F_T is a DF of type $(X \cup Y) \rightarrow Y'$. F_T is the abstract description of the transition relation.

5. F_O is a DF of $(X \cup Y) \rightarrow Z$. F_O is the abstract description of the output

relation.

2.2.4 Example

We use the traditional version of the Greatest Common Divisor (GCD) benchmark. This version computes the greatest common divisor of two positives numbers p_1 and p_2 by repeated subtraction.

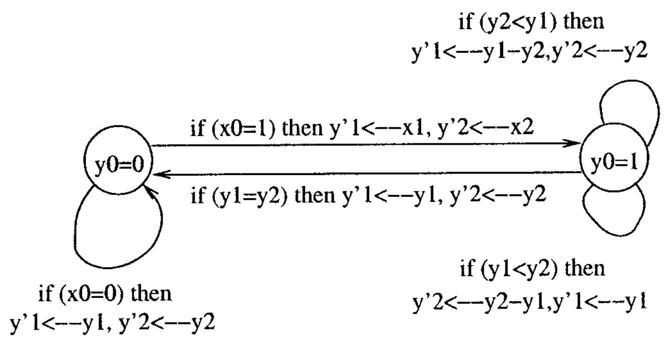


Figure 2.1: The GCD State Machine

The state machine initializes two variables y_1 and y_2 with values p_1 and p_2, then repeatedly assigns to the variable with the highest value the difference of the two values, until the two values are the same. When done, the value stored in the two variables is the greatest common divisor. Besides the two data state variables y_1 and y_2, there is a control state variable y_0 which determines two control states $y_0 = 0$ and $y_0 = 1$. When $y_0 = 0$, the machine waits for the two values p_1 and p_2 to be presented at two data inputs x_1 and x_2, an event which is indicated by the control input x_0 taking the value 1. Then p_1 and p_2 are loaded into y_1 and y_2, and the machine goes to the control state $y_0 = 1$ where it loops until the result has been computed. There is only one output: a data output z_0 that takes the value 0 while the result is not ready, and produces the greatest common divisor when it has been computed. The graphical representation of the GCD state machine is depicted in Figure 2.2, where the circles correspond to the values of the control state variable y_0 and the arrows correspond to the control transitions of the machine. The transition labels specify the conditions under which each transition is taken and an assignment of values to the abstract next state variables y'_1 and y'_2.

To obtain an abstract description of this state machine, we use a concrete sort *bool* with enumeration $\{0, 1\}$ and an abstract sort *num* intended to denote the set on n-bit numbers. The input variable x_0 and the state variable y_0, which are control variables, are of sort *bool*. On the other hand, the input variables x_1 and x_2, the state variables y_1 and y_2, and the output variable z_0, which are data variables, are of abstract sort *num*. We also use three next-state variables, y'_0 of sort *bool*, and y'_1 and y'_2 of sort *num*.

To denote the subtraction, a datapath operation, we define an abstract function symbol *sub* of type $num \times num \rightarrow num$. The function symbol *sub* is uninterpreted, which means that we do not have to describe the details of the subtraction operation. However, two pieces of information are needed: whether $y_1 = y_2$, to terminate the loop, and whether $y_1 < y_2$, to decide which substraction to make and which value to replace. This feedback from the datapath is modelled using two function symbols *eq* and *lt* of type $num \times num \rightarrow bool$. Thus the transition relation of the state machine can be described by the following formula:

$$\begin{aligned}
&((y_0 = 0) \wedge (x_0 = 0) \wedge (y'_0 = 0) \wedge (y'_1 = y1) \wedge (y'_2 = y2))\vee \\
&((y_0 = 0) \wedge (x_0 = 1) \wedge (y'_0 = 1) \wedge (y'_1 = x1) \wedge (y'_2 = x2))\vee \\
&((y_0 = 1) \wedge (eq(y_1, y_2) = 0) \wedge (lt(y_1, y_2) = 0) \wedge (y'_0 = 1) \\
&\wedge(y'_1 = sub(y_1, y_2)) \wedge (y'_2 = y2))\vee \\
&((y_0 = 1) \wedge (eq(y_1, y_2) = 0) \wedge (lt(y_1, y_2) = 1) \wedge (y'_0 = 1) \\
&\wedge(y'_1 = y_1) \wedge (y'_2 = sub(y_1, y_2)))\vee \\
&((y_0 = 1) \wedge (eq(y_1, y_2) = 1) \wedge (y'_0 = 0) \wedge (y'_1 = y1) \wedge (y'_2 = y2))
\end{aligned} \tag{2.1}$$

2.3 Multiway Decision Graphs

2.3.1 From BDDs to MDGs

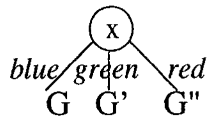
Binary Decision Diagrams are used to represent, canonically, Boolean functions. Consider a BDD G with a root node labelled x and subgraphs G' and G''. If G' and G'' represent the formulae P' and P'', respectively, then G is viewed as representing the formula P:

$$((\neg x) \wedge P') \vee (x \wedge P'') \tag{2.2}$$

However, it can also be viewed as representing the formula

$$((x = 0) \wedge P') \vee ((x = 1) \wedge P'') \tag{2.3}$$

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:

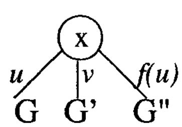


If G, G', and G'' represent the formulae P, P', and P'', respectively, then this graph could represent the formula

$$((x = \mathit{blue}) \wedge P) \vee ((x = \mathit{green}) \wedge P') \vee ((x = \mathit{red}) \wedge P''). \tag{2.4}$$

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 α, 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

$$((x = u) \wedge P) \vee ((x = v) \wedge P') \vee ((x = f(u)) \wedge P''). \tag{2.5}$$

The above observations lead to the following preliminary definition:

Definition 1 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 $\vee_{1 \leq i \leq n}((A = B_i) \wedge P_i)$.

Definition 1 is of course too general, a set of *well-formedness conditions* turns MDGs into *canonical representations* that can be manipulated by efficient algorithms.

2.3.2 Well-formedness Conditions

We first define the class of *concretely reduced* terms inductively as comprising: the individual constants, the abstract generic constants, the abstract variables and the terms of the form "$f(A_1, ..., A_n)$" where f is an abstract function symbol and $A_1 ... A_n$ are concretely reduced terms. Thus the concretely reduced terms are those that have no concrete sub-terms other than individual constants, and the only *concrete* terms that are concretely reduced are the individual constants. A term of the form "$f((A_1, ..., A_n)$" where f is a cross-operator and $A_1 ... A_n$ are concretely-reduced terms, is a *cross-term*. Note that no concrete variables can occur in a concretely-reduced term or in a cross-term.

For BDDs to be canonical, certain conditions must hold. They have to be reduced and ordered. Similarly, MDGs require certain well-formedness conditions.

Definition 2 An MDG G is said to be *well-formed* iff it satisfies the following six conditions.

1. *Kinds of nodes.* An internal node must be labelled by a variable of abstract sort, with edges issuing from the node labelled by concretely-reduced terms of that same sort; or by a variable of concrete sort, with edges labelled by individual constants in the enumeration of that sort; or by a cross-term, with edges labelled by individual constants in the enumeration of the sort of the cross-term. A leaf node must be labelled by T (true), except in the case where the graph has only one node labelled F (false).

Note that the conditions about concretely reduced terms and cross-terms are only syntactical restrictions, since it is possible to meet these restrictions using case splitting.

We refer to an occurrence of a variable in a term that labels an edge or in a cross-term that labels a node as a *secondary occurrence*, while an occurrence of a variable as the label of a node is a *primary occurrence*. Neither the edge labels, which are concretely reduced-terms, nor the cross-terms, contain concrete variables. Hence only abstract variables can have secondary occurrences. The *primary variables* (resp. *secondary variables*) of a graph G are those that have primary (resp. secondary) occurrences in G.

2. *Ordering.* The labels of the edges issuing from a given node must appear in a *standard term order*, without repetitions. Along each path, the variables and the cross-operators of the cross-terms that label the nodes must appear in a *custom symbol order*, and cross-terms with same cross-operator must appear in the standard term order; there must be no repeated labels.

The custom symbol order is a generalization (to include cross-operators) of the variable ordering used for BDDs, and plays the same role. It involves the cross-operators and those variables that may appear as node labels. It is chosen carefully for each particular application so as to keep the MDGs of manageable size if possible. The standard term ordering, on the other hand, is chosen arbitrarily once and for all, it needs not to be compatible with the custom symbol order. From these two orderings we define node-label ordering among the variables and cross-terms as follows: A comes before B iff the the top symbol of A comes before the top symbol of B, or A and B are cross-terms with the same cross-operator and A comes before B in the standard term order. Condition 2 states that node labels must appear in node-label order along each path.

3. *Minimality.* There must be no distinct isomorphic subgraphs, and no *redundant nodes.*

In an MDG, a redundant node is a node labelled by a concrete variable or cross-term of sort α, with edges labelled by all the individual constants in the enumeration of α, all leading to the same subgraph.

4. No variable should have both primary and secondary occurrences in the same graph.

5. The set of abstract variables having primary occurrences along a path is the same for all paths in a given graph.

6. If a node N is labelled by an abstract variable x, and an abstract variable y participating in the custom symbol order occurs in a term A that labels one of the edges that issue from N, then y must come before x in the custom symbol order. Similarly, if N is labelled by a cross-term A with cross-operator f, and y is an abstract variable that occurs in A, then y must come before f in the custom symbol order.

Figure 2.2 shows the MDG representations for the DFs describing the transition relations of the GCD state machine. From now on, unless otherwise stated, we shall not make the distinction between an MDG and the DF that it represents.

2.3.3 MDG Basic Operators

BDD operations can be manipulated using a single generic algorithm *Apply* [4]. This is because the two edges that issue from a BDD node span the range of values $\{0, 1\}$ and this makes it possible to reason by cases. For MDGs, a single algorithm must be provided for each operation.

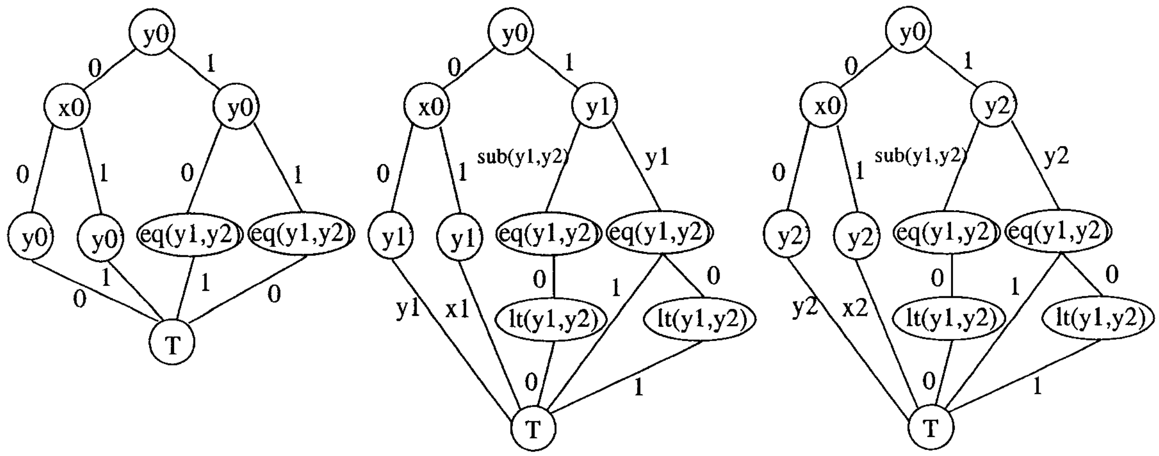


Figure 2.2: Transition Relations (MDGs) of the GCD State Machine

Disjunction

$R = Disj(\{P_i\}_{1\leq i\leq n})$

Argument: A set $S = \{P_i\}_{1\leq i\leq n}$ of MDGs P_i. Each P_i other than T or F is an MDG of type $U_i \to Y$. (Note that all the P_i have the same set of primary abstract variables.)

Result: An MDG R that can be F, T, or an MDG of type $V \to Y$, where V is the union of the sets of variables U_i such that:

$$\models R \iff (\bigvee_{1\leq i\leq n} P_i)$$

Relational Product

The relational product operation is used for image computation. It takes the conjunction of a collection of MDGs P_i, having pairwise disjoint sets of abstract primary variables, and, existentially quantifies with respect to the variables in a set E, 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 the renaming substitution η.

$R = RelP(\{P_i\}_{1\leq i\leq n}, V, \eta)$

Arguments: A set $S = \{P_i\}_{1\leq i\leq n}$, $n \geq 0$, of MDGs P_i, each being either T, or F, or of type $X_i \rightarrow Y_i$, a set of variables V and a renaming substitution η.

Result: An MDG R that can be F, T, or an MDG of type $(X \setminus Y) \rightarrow ((Y \setminus V) \cdot \eta)$ such that

$$\models R \cdot \eta^{-1} \iff (\exists V)(\bigwedge_{1\leq i\leq n} P_i).$$

Pruning By Subsumption

The pruning by subsumption operations is used to approximate the set difference operation. Informally, it removes all the paths of a graph P from another graph Q.

$$P' = PbyS(P, Q)$$

Arguments: Two MDGs P and Q of type $V \rightarrow Y$, where V contains only abstract variables that do not participate in the custom symbol ordering; P and Q can both be T or F.

Result: An MDG P', derivable from P by pruning, such that:

$$\models P \vee (\exists V)Q \iff P' \vee (\exists V)Q$$

Since P' is derivable from P by pruning, it is, like P, of type $V \rightarrow Y$. Moreover, if P is of type $V \rightarrow Y_1$, $Y_1 \subseteq Y$, then P' is also of type $V \rightarrow Y$ when it is not F.

2.3.4 MDG Reachability Analysis

We show here how the analysis of the reachable states of a state machine can be performed using MDGs. The main application is the invariant checking, which consists of verifying that the outputs of the machine satisfy a condition C in all the reachable states. Let $D = (X, Y, Z, Y', \eta, F_I, F_T, F_O)$ be an abstract description of a state machine using MDGs. The following pseudo-code provides an overview of the reachability analysis algorithm **ReAn**.

ReAn(D,C)

    $R := F_I$; $Q := F_I$; K :=0;
    loop
        $K := K$ +1;
        I := **NewInputs**(K);
        O := **Outputs**(I,Q,F_O);
        if not **Subset**(O,C) then return failure;
        N := **NextStates**(I,Q,F_T);
        Q := **FrontierSet**(N,R);
        if **Empty**(Q) then return success;
        R := **Union**(R,Q);
    end loop;
end **ReAn**;

The procedure **NewInputs** produces an MDG representing the set of input vectors which depends on the iteration number. The procedure **Outputs** computes an MDG representing the set of output vectors that is used to check whether the outputs satisfy the invariant. If this is the case, the verification algorithm continues. Otherwise it stops and reports failure. At the same time, a counterexample facility is initiated. The procedure **NextState** computes an MDG representing the set of states reachable in one transition of the state machine from the previously reached states. The procedure **FrontierSet** computes the set of newly reached stated. If this set is empty this means that all reachable states are already tested for the invariant and then the verification succeeds. Otherwise, the algorithm continues.

2.4 The MDG Package

In this section, we briefly present the MDG package [34] providing functions to assemble graphs and manipulate them. It was used for various MDG applications like the MDG model checker [33].

2.4.1 Graph Structure

The nodes of a graph are either internal nodes or leaves.

- *Leaves*: for well-formed MDGs, leaves are represented by T (True) or F (False). An MDG contains only one leaf labelled T except when the MDG is equal to the leaf node F.
- *Internal nodes*: an internal node is represented by the following structure:

graph(TopSymbolOrder, NodeKind, NodeLabel, Id, Edges, SubGraphs, SecVars).

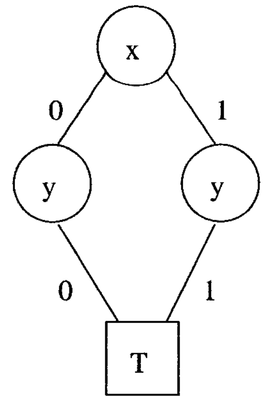


Figure 2.3: Example of the MDG Representation

where, *TopSymbolOrder* is the custom order number for the top symbol of the node label. *NodeKind* specifies whether the node is a concrete variable, a cross-term or an abstract variable, respectively. *Sort* being the sort of the node. *NodeLabel* is the term which is the label of the node. *Id* is the unique identifier of the graph. *Edges* and *SubGraphs* are two lists, representing the root edges and the immediate sub-graphs, respectively. Finally, *SecVars* is a sorted list that contains all the secondary variables in the graph. The internal representation of the example MDG given in Figure 2.3 is implemented as follows:

```
graph(1,concvar(bool),x,1,[0,1],
      [ graph(2,concvar(bool),y,2,[0],[t],[ ]),
        graph(2,concvar(bool),y,3,[1],[t],[ ]) ],
      [ ] ).
```

2.4.2 Assembling Graphs

Given the root information, the root edges and immediate subgraphs, the function *assemble* is used to build a graph R. It first checks if the result graph already exists by looking up the reduction table. If so, R points to that graph (thus achieving graph sharing). Otherwise, a new graph structure is created and entered in the graph array. The function *assemble* is invoked as follows:

assemble(RootInfo, Edges, SubGs, Method, R, G1, E1, G, E).

where, *RootInfo* contains the order, kind and label of the root. *Edges* are the root edges that should be of the same sort as the root, and *SubGs* are the immediate subgraphs. *Method* specifies the way secondary variables are computed and R is the result graph. While *G1* and *E1* are the current graph and term arrays, G, E are the updated arrays.

The secondary variables are computed, either by searching the graph or simply by giving the ordered union of the sets of secondary variables of the immediate subgraphs. To construct the example graph in Figure 2.3, assuming that the orders of x and y are 1 and 2, respectively, we use the function *assemble* as follows:

```
assemble(rootinfo(1,concvar(bool),x), [0,1],
         [ graph(2,concvar(bool),y,4,[0],[t],[ ]),
           graph(2,concvar(bool),y,5,[1],[t],[ ]) ],
           noop, G1, E1, G, E).
```

2.4.3 Manipulating Graphs

In this section we will overview the basic MDG operators provided by the package to manipulate the MDGs. In the following, G_1 and E_1 are the graph and term arrays before applying the operators, G and E are the graph and term arrays after.

Disjunction

Mode : *disj*(P_s,Q,G_1,E_1,G,E).

Arguments : P_s is a list of MDGs. Q is the result MDG.

Function : Q is the disjunction of P_s.

Relational Product

Mode : *relp*(P_s, U_s, *Ren*, Q, G_1,E_1,G,E).

Arguments : P_s is a list of MDGs, U_s is an ordered list of symbol orders, *Ren* is the renaming substitution.

Function : Q is the conjunction of P_s with existential quantification of variables in U_s, and if a node label is an argument of the renaming substitution then replace it with the new label.

Pruning by Subsumption

Mode : *pbys*(P,Q,R,G_1,E_1,G,E).

Arguments : P is the graph to be pruned by Q. R is the result graph.

Function : R is obtained by removing the paths in P which are subsumed by Q.

More operators are presented in the Developer's Manual of the MDG package [34].

Chapter 3

Embedding the MDG Logic

As in ordinary multi-sorted first order logic, the vocabulary of the MDG underlying logic consists of sorts, constants, variables, and function symbols or operators. In this chapter, we show how this underlying logic of MDGs is embedded in HOL. We will also show how the well-formedness conditions are specified in HOL resulting in what we will call: the *Well-formed MDG Terms.* Finally we will present some utilities to manipulate the well-formed terms.

3.1 MDG Sorts

The logic underlying the MDGs deviate from the standard many-sorted first-order logic by introducing a distinction between concrete or enumerated sorts, and abstract sorts (cf. Section 2.2.1). This is embedded in HOL as follows:

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

This 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".

$(IsConcreteSort\ (Concrete\_Sort\ \_\ \_\ ) = T) \wedge (IsConcreteSort\ \_ = F)$;

$(IsAbstractSort\ (\ Abstract\_Sort\ \_) = T) \wedge (IsAbstractSort\ \_ = F)$;

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

3.2 MDG Variables

As mentioned before, the distinction between sorts leads to the distinction between concrete and abstract variables. An abstract variable can be either primary or a secondary variable. 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` $\Rightarrow$ `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` $\Rightarrow$ `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` $\Rightarrow$ `Abstract_Sort;`

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

val y_1 = *Define* 'y_1 = *Secondary_Var* "y_1" *alpha*'.

We make the difference, however, to avoid mixing the variables in further manipulations and to allow us to declare the *MDG_Term* constructor as we will see in Section 3.5. 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*.

(IsConcreteVar(Concrete_Var _ _) = T) $\wedge$ *(IsConcreteVar _ = F)*;

$(IsAbstractVar(Abstract\_Var\ \_\ \_) = T) \wedge (IsAbstractVar\ \_ = F)$;

$(IsSecondaryVar(Secondary\_Var\ \_\ \_) = T) \wedge (IsSecondaryVar\ \_ = F)$;

3.3 MDG Constants

A constant can be either an individual 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` $\Rightarrow$ `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 use the predicates, *IsIndividualConstant* and *IsGenericConstant*.

(IsIndividualConstant(Individual_Const _) = T) $\wedge$ *(IsIndividualConstant _ = F)*;

(IsGenericConstant(Generic_Const _ _) = T) $\wedge$ *(IsGenericConstant _ = F)*;

3.4 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` $\Rightarrow$ `Secondary_Var list` $\Rightarrow$ `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` $\Rightarrow$ `Secondary_Var list` $\Rightarrow$ `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.

(IsAbstractFunction(Abstract_Function _ _ _) = T) $\wedge$ *(IsAbstractFunction _ = F);*

(IsCrossFunction(Cross_Function _ _ _) = T) $\wedge$ *(IsCrossFunction _ = F);*

3.5 MDG Terms

MDG terms are the individual constants, generic constants, concrete and abstract variables, cross and 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`
```

3.6 MDG Well-formed Terms

Well-formed terms are those that can be represented by well-formed MDGs, which are the directed formulae. Having embedded the notion of an MDG term in HOL, we should now specify the set of formulae that can be used to specify the MDG verification applications. To do so, we have to check, first, if a term or formula is well-formed before constructing its correspondent MDG. A well-formed term is actually a DF (more precisely, a concretely-reduced DF).

For a term to be a DF, conditions 1 to 3 of Section 2.2.2 must be satisfied. Condition 1 states that the term must be a formula in disjunctive normal form, in which, every disjunct is a conjunction of equations. The equations must respect the rules of Section 2.2.2. Condition 2 requires that the left hand sides of the equations are pairwise distinct and finally Condition 3 states that every abstract variable must appear in every disjunct.

Condition 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.

Condition 1 is embedded in HOL using an ML function called *Well_formedTerm* that uses the previously mentioned predicates to determine the nature of each equation in the term and returns *true* if the term is a directed formula. *Well_formedTerm* is a recursive function that splits the term in disjuncts and checks that every disjunct is well-formed. It uses an intermediate predicate *Well_formedEQ* that checks the well-formedness of an equation.

For every equation, we check if the left hand side and the right hand side respect one of the four allowed forms. For example, if an equation $eq : l = r$, then the main check is the following:

```
fun Well_formedEQ eq =
            (^(eval_IsConcreteVar l) /\ ^(eval_IsConcreteC r))   \/
            (^(eval_IsCrossF l)      /\ ^(eval_IsConcreteC r))   \/
            (^(eval_IsAbstractVar l )/\ ^(eval_IsAbstractF r))   \/
            (^(eval_IsAbstractVar l )/\ ^(eval_IsAbstractVar r)) \/
            (^(eval_IsAbstractVar l )/\ ^(eval_IsGenericC r))    \/
             ^(eval_IsBool l);
```

This means that *eq* is well-formed, if for example, l is a concrete variable and r is a concrete constant.

3.7 Utility Functions

In order to make use of the MDG embedding mentioned above, we provide various utility functions to facilitate the further manipulation of the MDG terms. For instance, to retrieve the label of a term, we use the function *name*. For example, if x is an individual constant defined by : *val* x = *Individual_Constant "stop"*, then the label *"stop"* of the term is given by *"name(x)"*.

The function *name* is defined as follows:

```
Define `( name ( Concrete_Var n _ )    = n)  /\
        ( name ( Abstract_Var n _ )    = n)  /\
        ( name ( Individual_Const n _ ) = n)  /\
        ( name ( Generic_Const n _ )= n)     /\
        ( name ( Cross_Function n _ _ )  = n) /\
        ( name ( Abstract_Function n _ _ ) = n)`;
fun name t =
        let val th = EVAL (--`name ^t`--)
            val res = rhs(concl(th))
        in
            stringSyntax.fromHOLstring res
        end;
```

Similarly, we define the following main utility functions:

- *sort*: determines the sort of a term. If y_1 is a secondary variable as defined in Section 3.4, *sort* y_1 returns *alpha*;

- *enum*: determines the enumeration of a concrete variable, e.g., *enum bool* returns $\{0, 1\}$;

- *cross_term*: determines the arguments of a compound term (a cross-term or abstract function). For example, if *equal* is a cross-function as defined in Section 3.4, *cross_term equal* returns $[y_1, y_2]$.

3.8 Summary

So far, we have embedded the logic underlying the multiway decision graphs into HOL. We made the distinction between concrete and abstract sorts. We defined the

MDG terms inside HOL then we defined the subset of first-order terms that can be represented by well-formed graphs. This subset is the so called Directed Formulae. We also introduced the constraints over HOL formulae to be well-formed. This can be checked before future manipulation with their correspondent MDGs. Finally, we provided a number of utility functions to further manipulate the MDG terms[1]. In the next chapter, we will present the new version of the MDG package that we have implemented to provide various utilities to construct MDGs and manipulate them.

[1]A complete description of all MDG embedding and utility functions can be found in http://hvg.ece.concordia.ca/Research/MDGHOL/Embedding.html.

Chapter 4

Linking MDG and HOL

Based on the embedding of the logic underlying the MDGs in HOL, in this chapter we discuss an interface that links the theorem prover to a lifted version of the MDG package.

4.1 Lifted MDG Package

The MDG package [34], provides tools for assembling graphs and manipulating them. However, these functionalities are not suitable to work interactively with HOL. In fact, as it is implemented, the MDG package allows the developer to write MDG based applications that take input files, process them, and return the result of the verification. In our case, we need functions that, for example, build the graph of a directed formula and return the resulting graph to be used afterwards. Besides, when leaving the MDG environment back to HOL, we need to save the graph and term arrays. For this purpose, we developed a lifted version of the MDG package that inherits the functionalities of the former package and provides new ones needed in our embedding.

4.1.1 Modified Functionalities

To allow the interaction between HOL and MDG we modified the MDG operators. Besides, we modified the function *assemble*, responsible for building a graph representation, given the root information, the edges and the immediate sub-graphs, to remove redundant nodes.

Building Graphs

We modified the main function *assemble* to remove redundant nodes when assembling a graph. This is done by, first, checking that the immediate subgraphs are not equal and are issuing from all the constants appearing in the enumeration of the root node sort. If this is the case, *assemble* continues the construction. Otherwise, the result graph is the immediate subgraph itself (they are all equal).

MDG Operators

We have modified the MDG operators so that we do not have to pass the graph and term arrays as arguments to the operators. This is very practical because when switching from HOL to MDG and vice-versa it is very inconvenient to carry these arrays as arguments especially when they get big.

To do so, The different operators use the arrays stored in the MDG environment, perform the operation over the graphs or terms and then update the arrays. All this work is done inside MDG and the result graph is returned back to HOL. For every operator, we provide two versions, the first takes graphs or terms as arguments and the second takes their IDs, instead.

4.1.2 New Functionalities

The lifted MDG package provides functions to build the graph of a well-formed HOL term and other facilities needed for the reachability analysis procedure.

Assembling the Graph of an Equation

Using the function *assemble*, we will define now the functions that are used to build the graph of an equation of the form $x = c$. The graph of such an equation will have a root node labelled by x with an edge labelled by c leading to T (true). If, for example, x is a concrete variable with order number 1, sort *state*, c is an individual constant, then the graph of $x = c$ is

$$graph(1,concvar(state),x,Id,[c],[T],[\ ]).$$

To build the graph of an equation, many cases are to be considered depending on the kind of the left hand side (LHS) and the right hand side (RHS) of the equation. If the LHS is a concrete variable x and the RHS is an individual constant c, then the graph of the equation is built using the function *mdgc*. It takes as input x and c and returns the result R.

```
mdgc(x,c,R):-
   g(Id1,G1),
   t(Id2,E1),
   signal(x,Sort),
   conc_sort(Sort,_),
   find_order(concvar(_),x,Order),
   RootInfo = rootinfo(Order,concvar(Sort),x),
   assemble(RootInfo,[c],[T],noop,R,(Id1,G1),(Id2,E1),G,E).
```

The first two lines of this Prolog code retrieve the graph and term arrays and their correspondent sizes. The third line determines the sort of x which is checked in the fourth line if it is concrete. Then in the fifth line the order of c is determined. In line 6, we set the information of the node labelled by x in the *RootInfo* structure. Finally, the information is gathered and the function *assemble* is called to build the graph.

Similarly, we use the following functions, depending on the kind of the sides of the equation:

- Cross-term - Concrete Constant : we use the function *mdgx*. In this case, the cross-term has to be built in advance;
- Abstract Variable - Generic Constant : we use the function *mdgac*;
- Abstract Variable - Abstract Variable: we use the function *mdgav*;
- Abstract Variable - Abstract function: the function used is *mdga*. The abstract function must be assembled and and added to the term array in advance.

Assembling the Graph of a Directed Formula

A directed formula is a disjunction of conjunctions of equations. To build its graph we need to build the MDGs of every equation and then perform the correspondent operations (conjunction, disjunction). The main function to call is *mdg* which will use the previous functions and the disjunction (*disj*) and conjunction *conj* operators.

To build the graph of a directed formula, we represent the formula as a list of lists, where every internal list contains the different equations of a disjunct. For example if:

$$f = [(x_1{=}c_1) \wedge (x_2{=}c_2)] \vee [(\mathrm{eq}(y_1,y_2){=}c_3) \wedge (x_4{=}c_4)]$$

then the mentioned list would be $[[x_1{=}c_1,x_2{=}c_2],[\mathrm{eq}{=}c_3,x_4{=}c_4]]$. This is in turn split into two lists containing the LHSs and the RHSs of the equations:

$$[[x_1,x_2],[\mathrm{eq},x_4]],\ [[c_1,c_2],[c_3,c_4]]$$

If a term is compound then the arguments are specified in another list otherwise the correspondent element will be "_". This list for the previous example is the following:

$$[[\_,\_]\ ,\ [[y_1,y_2],\_]]$$

These three lists of lists are passed to the function *mdg* which returns the graph of the directed formula.

$$mdg([[x_1,x_2],[\mathrm{eq},x_4]],[\ [\_,\_]\ ,\ [[y_1,y_2],\_]\ ],\ [[c_1,c_2],[c_3,c_4]],Result).$$

Generating Input Graphs

The function *inputs* allows to build a one-path graph that will be used as the input graph during the reachability analysis.

Mode: *inputs(Xs,J,G)*.

Arguments: *Xs* is a list of abstract variables, J is an integer, G is the result graph

Function: For every abstract variable x in the list *Xs*, builds an MDG of the equation $x = x\#j$ (input value associated to x in the j^{th} iteration) and then returns the conjunction of all the MDGs obtained. The result graph will serve as the input graph during the reachability analysis.

Filtering Abstract Variables

When generating the input graph, only abstract inputs are considered. To extract them we use the function *filter_abs*.

Mode: *filter_abs*(L_1, L_2).

Arguments: L_1 is a list of variables. L_2 is the result list

Function: picks the abstract variables of L_1 and inserts them in L_2.

Renaming Substitution

The renaming substitution function is used by the relational product operator.

Mode: *modify_ren(S,NS,Ren)*.

Arguments: S is a list of state variables and *NS* a list of their corresponding next-states. *Ren* is the result renaming substitution.

Function: Generates the renaming substitution for a list of state variables. First, the orders of the variables are retrieved. Then the maximum order is determined (used to optimize the relational product algorithm). Finally the renaming substitution is generated in the form *ren(Name, LO, Substitution)*. *Name* being the name of the substitution, *LO*, the maximum order of the list and *Substitution* is a list of 3-tuples *(Label, NewLabel, NewOrder)*.

Retrieving Graphs

Mode: *get_mdg_from_logarr(Id, G)*.

Arguments: *Id* is an ID, *G* is the graph corresponding to ID.

Function: Retrieve a graph from the graph array according to its ID. This function is used every time we call the MDG package to assemble a graph or to manipulate graphs. It returns the resulting graph. It is also used internally used by the package to perform operations over graphs.

4.2 Linking MDG to HOL

In Section 4.1 we presented a lifted version of the MDG package to manipulate the MDGs. In this section we will discuss the way we link HOL and the MDG package to solve the verification problem.

4.2.1 HOL-MDG Interaction

To let HOL communicate with the lifted MDG package, we use the SML library *Process* [17]. This library provides a function called *system* allowing HOL to call external processes. In our case, the external process will be the lifted MDG package compiled as a stand-alone program.

The lifted MDG package is invoked by HOL functions using a script file, in which, we specify the different manipulations to be done in MDG. For every HOL function, that needs to call the lifted MDG package, we provide a function that generates automatically the corresponding script file. If, for example, we want to perform the conjunction of a list of terms using the ML function *Conj*, an intermediate function called *MakeConjScript* is invoked to generate a script file. In this file, we will find a call to the MDG function *conj* to perform the conjunction of the corresponding graphs, and a call to the MDG function *get_graphId* to retrieve the ID of the resulting graph.

The HOL function passes the script file to the MDG package using the *system* function mentioned above. The MDG package computes the result and then writes it in a file "*mdghol.ch*". Using the function *ReadMdgOutput*, the result is returned to HOL.

4.2.2 Constructing MDGs in HOL

To construct the graph representing a HOL term we use the function *termToMdg*. This function uses the MDG function *mdg* (cf. Section 4.1.2) by passing a script file in which all the necessary data are specified.

Before calling the MDG package, *termToMdg* invokes *Well_formedTerm* to check if the term if well-formed. 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:

$$[\text{"}conc\_sort(bool,[0,1]).\text{"},\text{"}abs\_sort(alpha).\text{"}].$$

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.

4.2.3 Interfacing MDG Basic Operators

As mentioned before, the MDG operators are interfaced to HOL using script files. The same names will be given to MDG functions and their corresponding HOL functions. The manipulation of HOL terms, resolves to the manipulation of their MDG representations. This means that the operators call *termToMdg* to build the MDG representations and then call the corresponding MDG functions to compute the result. *termToMdg* returns the ID of a graph. This explains the introduction of two versions of the same operator.

- *Conj* : conjunction of HOL terms using their graph representations;
- *ConjId* : conjunction of HOL terms using their graph representations IDs;
- *Disj* : disjunction of HOL terms using their graph representations;
- *DisjId* : disjunction of HOL terms using their graph representations;
- *Relp* : relational product using the graph representations;
- *RelpId* : relational product using the graph representations IDs;
- *PbyS* : pruning by subsumption using the graph representations;
- *PbysID* : pruning by subsumption using the graph representations IDs.

4.3 Summary

In this chapter we presented a lifted MDG package providing functions to build and manipulate graphs and allowing to perform the operations interactively. We also showed the way HOL and MDG communicate[1]. HOL calls the lifted MDG package via script files that are generated by the calling functions. To build the graph representing a well-formed term, we use the ML function *termToMdg* which returns the ID of the graph as a result. Finally, the MDG operators are linked to ML functions allowing the manipulation of HOL terms by manipulating their graph representations, instead.

[1]A detailed description of the lifted MDG package and MDG-HOL linking functions can be found in http://hvg.ece.concordia.ca/Research/MDGHOL/Embedding.html.

Chapter 5

Embedding MDG Applications

In this chapter we will show how to use our embedding to implement MDG applications inside HOL. We will illustrate this by different applications like the reachability analysis to perform invariant checking and sequential equivalence checking.

5.1 Reachability Analysis

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.

5.1.1 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:

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

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

5.1.2 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.

5.1.3 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 fix-point (set).

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

```
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

[1]This is called the non-termination problem which was tackled in [2] using various heuristics.
[2]For the sake of clarity, this is just a simplified version of the algorithm

data input. S is initialized to B (the initial state), and the sets of next-states are computed until reaching a fix-point characterized by an empty frontier set.

5.2 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.

5.2.1 Examining the Outputs

For a certain state of the machine and a certain set of inputs, the set of outputs is computed and represented by an MDG O_s. Similarly, the invariant is given by its MDG representation C. To verify that the invariant holds on these outputs, we use the MDG operator *PbyS* with O_s and C as arguments. The pruning by subsumption operation returns the graph resulting from removing the paths of O_s assumed by C

(i.e, the paths of C). If the resulting graph is equal to the MDG F (false), then all the graphs of O_s are assumed by C. This ensures that the outputs of the machine satisfy the invariant and then the new outputs are computed and checked until reaching all the states of the machine or finding outputs that do not satisfy the invariant. However, if the graph resulting from pruning the invariant from the outputs is not the false MDG, then there exists some outputs where the invariant does not hold. In this case, the procedure terminates and an error is reported.

5.2.2 Generating the Inputs

The reachability procedure requires a supply of fresh input variables. Whenever an abstract variable x is used as input, a fresh variable u_k^x is generated in every iteration k of the procedure to serve as the symbolic value of x. In practice, we construct u_k^x by concatenating the identifier x, the symbol # and the decimal representation of the number k, i.e., u_k^x is "x#k". The function *NewInputs* constructs a linear (one-path) MDG representing the formula $\bigwedge_x (x = u_k^x)$.

The function *NewInputs*, first, retrieves the abstract variables from the set of the inputs using the function *FilterAbs* and then constructs the MDG representing the inputs using the function *GenerateInputs*. The latter takes the list of the abstract input variables and the iteration number as arguments and calls the MDG function *inputs* (cf. Section 4.1.2) to construct the graph.

5.2.3 Renaming Substitution

During the reachability analysis, we have to use the renaming substitution operation which renames the next-state variables to their corresponding current-state variables. For example, if s is a state variable, computing the next-state of the machine will introduce the next-state variable s' of the variable s. Before proceeding to the next iteration we need to rename s' to s. To generate the renaming substitution function η we use the ML function *GenerateRenaming* which in turn calls the MDG function *modify_ren* described in Section 4.1.2.

5.2.4 Checking an Invariant

Using the previously mentioned functions, 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 Tr Or In 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) != 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;
```

5.3 Model Checking in HOL

Checking that a property, described as a temporal logic formula, holds on a model of a system is the essence of model checking. Using the reachability analysis embedding, we implemented a certain number of MDG temporal operators [33] inside HOL. The property templates that we considered are the following:

- **AG** P: P holds on all the states of every path;
- **AF** P: In all paths, P eventually holds;

- **A** P : In all paths, P holds in all the states reached in at least n transitions.

In the following we present how the property **AF** P is embedded in HOL.

```
Check_AF Tr In 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 != Sigma then return failure
         Sigma = Disj Sigma Q
         K = K+1
         C = ComputeNext G_In Q G_Tr
       end loop
end Check_AF;
```

5.4 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 T.

5.4.1 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
         (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.

5.4.2 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;
```

5.5 Summary

In this chapter, we have embedded the reachability analysis inside HOL using the MDG embedding described in Chapter 3 and the lifted MDG package displayed in Section 4.1. We used the reachability analysis to implement the model and invariant checking procedures allowing to perform property checking for abstract state machines specified in HOL. We have also implemented functions to check, automatically, the equivalence of HOL terms and perform tautology checking[3]. This shows the importance of our embedding to provide some automation to the HOL theorem prover. Another application of the reachability analysis would be the sequential equivalence checking of abstract state machines. This is, somehow, similar to the invariant checking procedure as we will consider the product of the state machines, the invariant stating the equivalence of their correspondent outputs.

[3]A full description of the MDG applications embedding can be found in http://hvg.ece.concordia.ca/Research/MDGHOL/Embedding.html.

Chapter 6

Case Study : Island Tunnel Controller

In this chapter we will show how the invariant checking procedure described in Section 5.2 is used to verify a range of properties on the Island Tunnel Controller (ITC) as a case study example. The ITC was originally introduced by Fisler and Johnson [12]. It controls the vehicles traffic in a one-lane tunnel connecting the mainland to a small island, as shown in Figure 6.1(a). At each end of the tunnel, there is

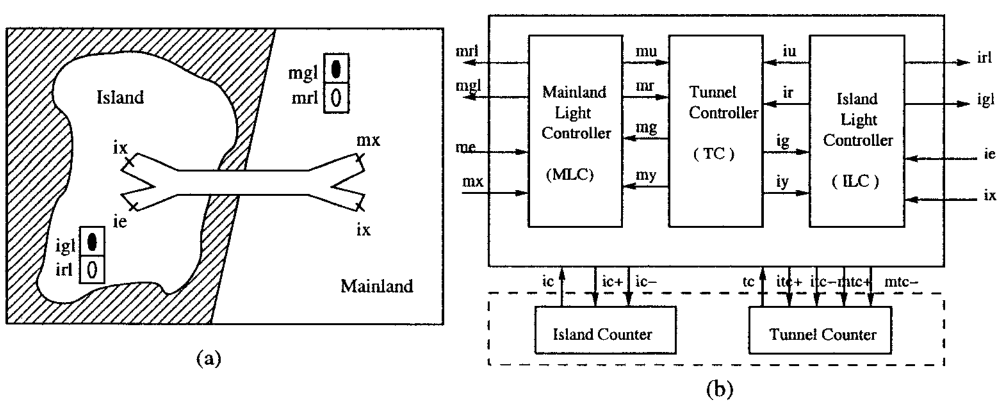


Figure 6.1: The Island Tunnel Controller

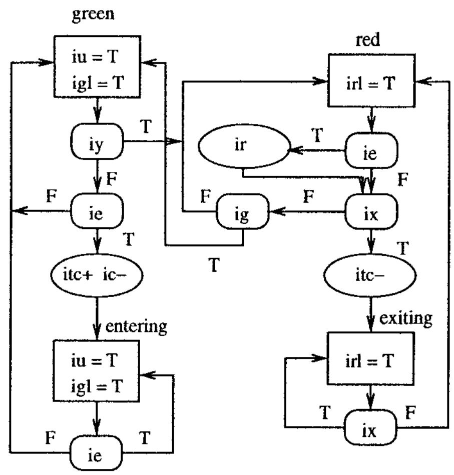


Figure 6.2: State Transitions Diagram of the ILC

a traffic light. There are four sensors for detecting the vehicles: one at the tunnel entrance (*ie*) and one at the tunnel exit on the island side (*ix*), and one at the tunnel entrance (*me*) and one at the tunnel exit on the mainland side (*mx*). It is assumed that all cars are finite in length, that cars cannot enter the tunnel on red light, that no car gets stuck in the tunnel, that cars do not exit the tunnel before entering the tunnel, and that there is sufficient distance between two cars such that the sensors can distinguish the cars.

The ITC is specified using three communicating controllers and two counters as shown in Figure 6.1(b). The state transition diagram of the Island Light Controller (ILC) is shown in Figure 6.2. The ILC has four states: *green, entering, red* and *exiting*. The output *igl* and *irl* control the green and red lights on the island side, respectively; *iu* indicates that the cars from the island side are currently using the tunnel, and *ir* indicates that the island is requesting the tunnel. The input *iy* requests the island to yield control of the tunnel, and *ig* grants control of the tunnel.

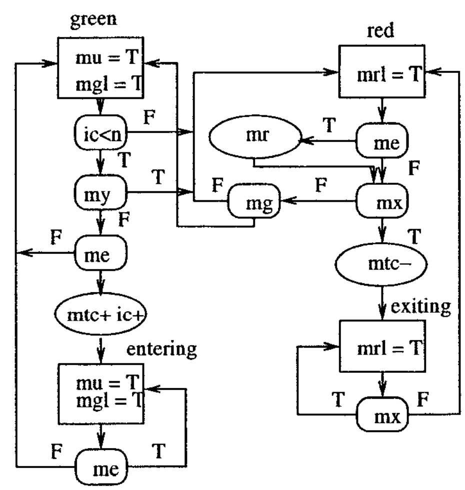


Figure 6.3: State Transitions Diagram of the MLC

A similar set of signals is defined for the Mainland Light Controller (MLC) as shown in Figure 6.3.

The state transition diagram of the the Tunnel Controller (TC) is depicted in Figure 6.4. The TC processes the requests for access issued by the ILC and MLC. The Island Counter and the Tunnel Counter keep track of the numbers of cars currently on the island and in the tunnel, respectively. At each clock cycle, the count tc of the tunnel counter is increased by 1 depending on signals $itc+$ and $mtc+$, or decremented by 1 depending in $itc-$ and $mtc-$, unless it is already 0. The island counter operates in a similar way, except that the increment and decrement signals are $ic+$ and $ic-$, respectively.

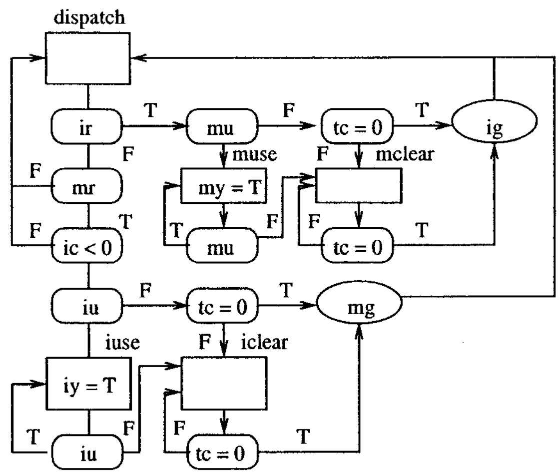


Figure 6.4: State Transitions Diagram of the TC

6.1 ITC Specification using Directed Formulae

Let *is*, *ms* and *ts* be the control state variables of the three controllers ILC, MLC and TC, respectively, and let *n_is*, *n_ms* and *n_ts* be the corresponding next state variables. We define a concrete sort *mi_sort* having the finite enumeration {*green, red, entering, exiting*} .

```
val mi_sort = Define`mi_sort=Concrete_Sort "mi_sort"
                                  ["green";"red";"exiting";"entering"]`;
val green    = Define`green    = CONCRETE_CONST "green"    mi_sort`;
val red      = Define`red      = CONCRETE_CONST "red"      mi_sort`;
val exiting  = Define`exiting  = CONCRETE_CONST "exiting"  mi_sort`;
val entering = Define`entering = CONCRETE_CONST "entering" mi_sort`;
```

The variables *is* and *ms* and their next state variables are assigned to be of this sort.

```
val is   = Define` is   = Concrete_Var "is"   mi_sort`;
val n_is = Define` n_is = Concrete_Var "n_is" mi_sort`;
val ms   = Define` ms   = Concrete_Var "ms"   mi_sort`;
val n_ms = Define` n_ms = Concrete_Var "n_ms" mi_sort`;
```

Similarly, we let *ts* and *n_ts* to be of sort *ts_sort* which has the enumeration {*dispatch, iuse, muse, iclear, mclear*}.

```
val ts_sort = Define‘ ts_sort = Concrete_Sort "ts_sort"
      ["dispatch";"iuse";"muse";"iclear";"mclear"]‘;
```

All other control signals (ie, ix, me, mx, etc) are of sort *bool* with the enumeration $\{0, 1\}$. The condition "$ic < n$" is represented by the cross-term $lessn(ic)$, where the uninterpreted cross-function *lessn* of type *wordn* $\rightarrow$ *bool* represents the operation "$< n$". *wordn* is a default abstract sort for n-bit words.

```
val bool  = Define‘ bool  = Concrete_Sort "bool" ["0";"1"]‘;
val wordn = Define‘ wordn = Abstract_Sort "wordn"‘;
val ie    = Define‘ ie    = Concrete_Var  "ie" bool‘;
val ix    = Define‘ ix    = Concrete_Var  "ix" bool‘;
val me    = Define‘ me    = Concrete_Var  "me" bool‘;
val mx    = Define‘ mx    = Concrete_Var  "mx" bool‘;
val lessn = Define‘ lessn = Cross_Fun "lessn" [ic] bool‘;
```

Both the island and the tunnel counters have each only one control state, *ready*, hence no control state variable is needed. An abstract state variable ic (tc) represents the current count number. At each clock cycle, the count is updated according to the control signals. In this abstract description, the count ic (tc) is of sort *wordn*.

```
val wordn = Define‘ wordn = Abstract_Sort "wordn"‘;
val ic    = Define‘ ic    = Abstract_Var  "ic"   wordn‘;
val n_ic  = Define‘ n_ic  = Abstract_Var  "n_ic" wordn‘;
val tc    = Define‘ tc    = Abstract_Var  "tc"   wordn‘;
val n_tc  = Define‘ n_tc  = Abstract_Var  "n_tc" wordn‘;
```

The control signals (ic+,ic-,etc.) are of sort *bool*. The uninterpreted function *inc* of type *wordn* $\rightarrow$ *wordn* denotes the operation of increment by 1, and *dec* of the same type denotes decrement by 1. The cross-term $equz(tc)$ represents the condition "tc=0" and models the feedback from the counter to the control circuitry; *equz* is a cross-function symbol of type *wordn* $\rightarrow$ *bool*.

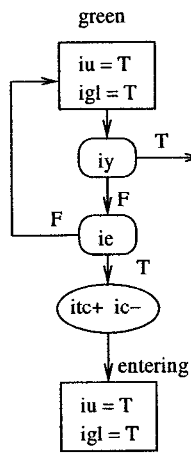


Figure 6.5: Transitions from the State *green* (ILC)

```
val ic_plus = Define` ic_plus = Concrete_Var  "ic_plus" bool`;
val ic_min  = Define` ic_min  = Concrete_Var   "ic_min" bool`;
val equz    = Define` equz    = Cross_Fun    "equz" [tc] bool`;
val dec     = Define` dec     = Abstract_Fun "dec" [tc] wordn`;
val inc     = Define` inc     = Abstract_Fun "inc" [tc] wordn`;
```

Once the above algebraic specifications are defined, the state transition diagrams can be easily transformed into a set of directed formulae. First, let us consider the formula representing the transition relation for the ILC. The transitions from the state *green* are given by the Figure 6.5, and specified by the formula:

```
( (is=green)    /\ (iy=zero)  /\  (ie=zero) /\  (n_is=green))     \/
( (is=green)    /\ (iy=zero)  /\  (ie=one)  /\  (n_is=entering)) \/
( (is=green)    /\ (iy=one)   /\                (n_is=red))
```

Similarly, we consider the different states of the ILC, to extract the formula representing the state variable *is*. The transition relation of the ILC is then specified by the formula $t0$ below.

```
val t0 =
( (is=green)    /\ (iy=zero)  /\  (ie=zero) /\  (n_is=green))     \/
( (is=green)    /\ (iy=zero)  /\  (ie=one)  /\  (n_is=entering)) \/
```

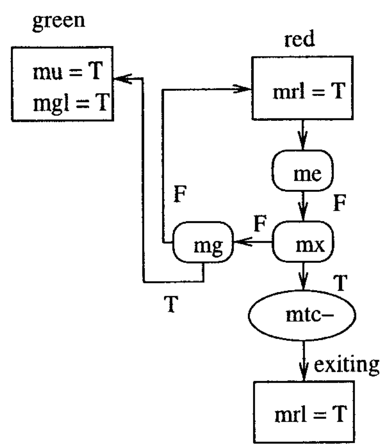


Figure 6.6: Transitions from the State *red* (MLC)

```
( (is=green)     /\ (iy=one)    /\                       (n_is=red))        \/
( (is=entering) /\ (ie=zero)   /\                       (n_is=green))      \/
( (is=entering) /\ (ie=one)    /\                       (n_is=entering)) \/
( (is=red)       /\ (ig=zero)   /\  (ix=zero) /\  (n_is=red))        \/
( (is=red)       /\ (ig=one)    /\  (ix=zero) /\  (n_is=green))      \/
( (is=red)       /\ (ix=one)    /\                       (n_is=exiting))  \/
( (is=exiting)  /\ (ix=zero)   /\                       (n_is=red))        \/
( (is=exiting)  /\ (ix=one)    /\                       (n_is=exiting));
```

The transition relation of the MLC (Figure 6.3) is translated to the formula *t1*. If we consider the state *red*, the possible transitions are specified in the Figure 6.6 and are represented by the following formula.

```
( (ms=red) /\ (mg=zero) /\ (mx=zero) /\ (n_ms=red)      )  \/
( (ms=red) /\ (mg=one)  /\ (mx=zero) /\ (n_ms=green)    )  \/
( (ms=red) /\ (mx=one)               /\ (n_ms=exiting) )
```

The relation between *ms* and its next-state variable is specified by the formula below.

```
val t1 =
 ( (ms=green)    /\ (lessn=zero) /\ (n_ms=red) )       \/
 ( (ms=green)    /\ (my=zero)    /\ (me=zero)    /\
                    (lessn=one)  /\ (n_ms=green) )     \/
 ( (ms=green)    /\ (my=zero)    /\ (me=one)     /\
                    (lessn=one)  /\ (n_ms=entering) ) \/
```

```
( (ms=green)     /\ (my=one)      /\ (lessn=one) /\
                                     (n_ms=red) )        \/
( (ms=entering) /\ (me=zero)      /\ (n_ms=green) )      \/
( (ms=entering) /\ (me=one)       /\ (n_ms=entering) ) \/
( (ms=red)       /\ (mg=zero)     /\ (mx=zero)    /\
                                     (n_ms=red))         \/
( (ms=red)       /\ (mg=one)      /\ (mx=zero)    /\
                                     (n_ms=green) )      \/
( (ms=red)       /\ (mx=one)      /\ (n_ms=exiting) )   \/
( (ms=exiting)   /\ (mx=zero)     /\ (n_ms=red) )        \/
( (ms=exiting)   /\ (mx=one)      /\ (n_ms=exiting) );
```

6.2 Invariant Checking

We list below some examples properties that we verified. For all the properties verified, the initial state of ILC and MLC, if not explicitly stated, is given by the following formula.

```
val initial = (is=red) /\ (ms=red);
```

6.2.1 Properties

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 = one)  /\ (mgl=zero) )  \/
         ( (igl = zero) /\ (mgl=one)  )  \/
         ( (igl = zero) /\ (mgl=zero) );
```

To verify this property we used the embedded invariant checking procedure of Section 5.2. The transition relation to be used is the conjunction of the transition relations of ILC and MLC. The output relation is the formula specifying the behavior of *igl* and *mgl* which is the conjunction of the following formulae.

```
( (is=red)      /\ (igl=zero) ) \/
( (is=green)    /\ (igl=one)  ) \/
( (is=exiting)  /\ (igl=zero) ) \/
( (is=entering) /\ (igl=one)  );

( (ms=red)      /\ (mgl=zero) ) \/
( (ms=green)    /\ (mgl=one)  ) \/
( (ms=exiting)  /\ (mgl=zero) ) \/
( (ms=entering) /\ (mgl=one)  );
```

Property 2

Property 2 states that the access to the tunnel is not granted by the tunnel controller to the island and the mainland at the same time.

```
val P2 = ( (ig = one)  /\ (mg=zero) )  \/
         ( (ig = zero) /\ (mg=one)  )  \/
         ( (ig = zero) /\ (mg=zero) );
```

Property 3

Property 3 states that the light on the island side is never set to green if no grant is received from the controller. This is specified by the following.

```
val P3      = ( (igl=zero) );
val initial = ( (is=red) /\ (ig=zero) );
```

Property 4

If the light on the island side is green, it stays green as long as the tunnel is not requested from the mainland.

```
val P4      = ( (igl=one) );
val initial = ( (is=green) /\ (iy=zero) );
```

Property 5

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 P5      = ( (mgl=one) );
val initial = ( (ms=green) /\ (my=zero) );
```

Property 6

This property corrects property 5 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 the island and the number of allowed cars is not exceeded.

```
val P6      = ( (mgl=one) );
val initial = ( (ms=green) /\ (my=zero) /\ (lessn=one) );
```

Property 7

Property 7 states that the island counter is never signalled to increment and decrement simultaneously. This is specified by:

```
val P7 =  ( (ic_min=one)   /\ (ic_plus=zero) ) \/
          ( (ic_min=zero) /\ (ic_plus=one)   ) \/
          ( (ic_min=zero) /\ (ic_plus=zero) );
```

Property 8

The tunnel counter is never signalled to increment simultaneously by ILC and MLC.

which is written as follows.

```
val P8 =  ( (itc_plus=one)   /\ (mtc_plus=zero) ) \/
          ( (itc_plus=zero) /\ (mtc_plus=one)   ) \/
          ( (itc_plus=zero) /\ (mtc_plus=zero) );
```

Property 9

The green light must be off if there is a car exiting the tunnel.

```
val P9 = (ix=zero) \/ (igl=zero);
```

In this case, the transition relation we consider is the one of ILC.

6.2.2 Experimental Results

To verify the mentioned properties, we used the invariant checking procedure of Section 5.2. 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.

The function *InvariantChecking*, first, builds the graphs of the transition relations, the initial states and the invariant. Then generates the input graph and the renaming substitution function. The outputs are computed and checked for the invariant for all the reachable states of the system. The verification results, run on an Ultra2 Sun workstation with 296Mhz CPU and 768MB memory, are reported in Table 6.1. A "*" beside a property means that this latter failed in the invariant checking.

Property	**CPU**$_{s\ (system)}$	**CPU**$_{s\ (runtime)}$	**Memory**$_{MByte}$
Property1	0.32	101.9	0.220
Property2	0.060	72.0	0.058
Property3	0.060	44.8	0.035
Property4	0.010	44.3	0.020
*Property5	0.005	52.8	0.013
Property6	0.050	54.9	0.077
Property7	0.065	63.9	0.039
Property8	0.065	64.2	0.039
Property9	0.060	45.4	0.035

Table 6.1: Property Checking Results using *InvariantChecking*

The memory usage statistics were retrieved from the MDG package in terms of the addition of the memory used to build the different graphs, while the CPU time is retrieved using specific ML functions. The statistics for the CPU time represent both the time to perform the reachability analysis (CPU $_{system}$) and the time to translate the HOL specification to MDG files (CPU $_{runtime}$). It is clear that the verification is much faster than doing the proof interactively with HOL. Our approach may be slower than using model checking but only for examples that can be handled automatically. Hence, our approach proves its importance for large systems that require combination of theorem proving and model checking. To summarize, we are not concerned with performance, instead, we focus on broadening the class of systems that can be verified.

Using HOL to specify the problem gives the user more capabilities to handle the verification task by using the available facilities such as deduction. After interpreting the results returned using the MDG embedding, respective HOL theorems can be created.

Chapter 7

Conclusion 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 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 thesis 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 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.

Future Research Directions

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 using it for formal verification:

- Optimizing the package: The MDG package we provide is written in Prolog which is the language of fast prototyping. Garbage collection is not used in the package and the graph array cannot fully exploit structure sharing. Hence the package can be optimized if written in C or Java. Besides, interfacing to ML will be, in this case, faster;

- HOL terms simplification: The multiway decision graphs represent canonically well-formed terms. This can be used to provide tactics that simplify HOL terms by building their MDGs which will be reduced by construction and then retrieving the directed formula that is represented by the graph. The obtained formula will be reduced because the redundant nodes of the graph are eliminated and the graph ensures structure sharing;

- Model reduction: While checking the properties, we specify only the transition relations (DFs) of the model under verification that are involved in the properties. This is done manually. The idea here is to write a script which will automatically extract the transition relations based on the variables used in the properties to check;

- Using LCF style: The *Fully Expansive* LCF style [16] of HOL means that theorems can be derived only by using axioms and inference rules. This can be applied for our embedding when constructing the MDGs. Hence, 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 [15];

- Formal proof of the soundness of the MDG algorithms: Our embedding of the MDG data structure and operators, would allow the formal specification and verification in HOL of MDG applications and algorithms such as model checking. A similar work was done in [7] to verify a SPIN model checking algorithm.

Bibliography

[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] S. Bose and A. L. Fisher. Automatic Verification of Synchronous Circuits using Symbolic Simulation and Temporal Logic. In *Proc. of the IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pages 759–764, Leuven, Belgium, November 1990.

[4] R. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions in Computers*, 35(8):677–691, August 1986.

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

[6] J. R. Burch, E. M. Clarke, and K. L. McMillan. Symbolic model checking: 10^{20} states and beyond. In *Proc. of IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., USA, June 1990.

[7] 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.

[8] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In *Logics of Programs*, volume 131 of LNCS, pages 52–71. Springer-Verlag, May 1981.

[9] 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.

[10] O. Coudert, J. C. Madre, and C. Berthet. Verifying Temporal Properties of Sequential Machines without Building Their State Diagrams. In *Computer Aided Verification*, volume 531 of LNCS, pages 23–32. Springer-Verlag, June 1990.

[11] L. A. Dennis, G. Collins, M. Norrish, R. Boulton, K. Slind, G. Robinson, M. Gordon, and T. Melham. The PROSPER Toolkit. In *Tools and Algorithms for the*

Construction and Analysis of Systems, volume 1785 of LNCS, pages 78–92, Berlin, Germany, April 2000. Springer-Verlag.

[12] 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.

[13] 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.

[14] 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.

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

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

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

[18] J. Hurd. Integrating Gandalf and HOL. In *Theorem Proving in Higher Order*, volume 1690 of LNCS, pages 311–321. Springer-Verlag, April 1999.

[19] 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.

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

[21] 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.

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

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

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

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

[26] 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.

[27] L. C. Paulson. *Isabelle: A Generic Theorem Prover.* Springer-Verlag, 1994.

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

[29] A. Pnueli. A Temporal Logic of Concurrent Programs. *Theoretical Computer Science*, 13(1):45–60, January 1981.

[30] J. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CAESAR. In *Programming*, volume 137 of LNCS, pages 337–351. Springer-Verlag, 1982.

[31] 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.

[32] 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.

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

[34] Z. Zhou. *MDG Tools (V1.0) Developer's Manual.* Computer Science Department, University of Montreal, Canada, 1996.

[35] Z. Zhou. *Multiway Decision Graphs and Their Applications in Automatic Formal Verification of RTL Designs.* PhD thesis, Computer Science Department, University of Montreal, Canada, 1996.