A Hybrid Tool for Linking HOL Theorem Proving with MDG Model Checking

Rabeb Mizouni

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

Montréal, Québec, Canada

April 2003

CONCORDIA UNIVERSITY

Division of Graduate Studies

This is to certify that the thesis prepared

By: **Rabeb Mizouni**

Entitled: **A Hybrid Tool for Linking HOL Theorem Proving with MDG Model Checking**

and submitted in partial fulfilment of the requirements for the degree of

Master of Applied Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________ Dr. M. Reza Soleymani

______________________ Dr. Otmane Ait Mohamed

______________________ Dr. Patrice Chalin

______________________ Dr. Sofiène Tahar

Approved by ______________________

Chair of the ECE Department

__________ 2003 ______________________

Dean of Engineering

ABSTRACT

A Hybrid Tool for Linking HOL Theorem Proving with MDG Model Checking

Rabeb Mizouni

Nowadays, the formal verification of hardware is gaining a lot of importance in the design flow of micro-electronics systems. There exists several formal hardware verification approaches each with its own advantages and drawbacks. Hence, the idea of linking different approaches to benefit from their advantages has emerged as a potential ultimate solution. In this thesis, we describe a hybrid tool for formal hardware verification that links the HOL (Higher-Order Logic) theorem prover and the MDG (Multiway Decision Graphs) model checker. Our tool supports abstract datatypes and uninterpreted function symbols available in MDG, allowing the verification of high level specifications. For this purpose, we embedded in HOL the grammar of the hardware description language, MDG-HDL, used to represent models to be verified. Furthermore, we provided an embedding of the first-order temporal logic $\mathcal{L}_{mdg}$ used to express properties for the MDG model checker. Furhtermore, we have developed an interface which reads a HOL goal, generates the required MDG files, calls the MDG model checker, and generates the HOL theorem on successful verification.

Our tool also handles design hierarchies by reducing the model to its subsystem according to the property to be verified. Verification with the hybrid tool is faster and more tractable than using either tool separately. This has been illustrated via a number of simple hardware benchmark examples as well as a more elaborated design case study.

A mes Parents qui me manquent tellement ...

A mon cher mari Anis...

ACKNOWLEDGEMENTS

I would like to express my gratitude to all those who gave me the ability to complete this thesis.

I am deeply indebted to my supervisor Dr. Tahar for his precious instruction. His dynamic thinks, and his broad and profound knowledge have given me a great help.

I would like to express my special gratitude to Dr. Curzon from Middlesex University for his guidance and stimulating suggestions during my research work.

I am also obliged to Dr. Ait-Mohamed for his valuable discussions and hints.

The HVG group colleagues provided a nice atmosphere for discussions and research, I thank them all, especially, Amr Talaat for his great help and valuable feedback on the first draft of this thesis.

Last but not least, I would like to give my gratitude to Dr. Ben Ayed, who introduced me to the formal methods subject and made me really enjoy it.

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

With the ever increasing growth in the design of digital systems, and the size of microelectronics circuits, the role of design verification has gained a lot of importance. Nowadays, simulation is considered the main testing approach. Nevertheless, serious design errors often remain undetected despite the major efforts to improve simulation techniques. To overcome these limitations, *formal verification* has been introduced [34]. *Formal verification* relies on a strong mathematical background. It tries to mathematically prove that an implementation of a system fully satisfies its specification. There exist today several formal verification approaches like theorem proving, model checking, equivalence checking, etc. Each has advantages and drawbacks. In this thesis, we present our efforts in formalising and implementing a way for allowing the HOL (Higher-Order Logic) theorem prover to support the proof procedure of the Multiway Decision Graphs (MDG) model checker.

The MDG [5] system is a decision diagram based verification tool, primarily designed for hardware verification. It supports both equivalence checking and model checking. It is based on multiway decision graphs which extend Reduced-Ordered Binary Decision Diagrams (ROBDD) [3] with abstract sorts and uninterpreted function symbols.

HOL [25] is an interactive theorem prover based on higher-order logic. It can handle very large circuits for verification, without any restriction on the size. However, since it implements a white box verification approach (user interactivity), it is time-consuming and needs a high expertise of the user. Therefore, it does not fit the current VLSI industry needs, which is more time-to-market oriented, where fast, efficient, and trusted techniques are adopted for testing products. The idea of developing hybrid approaches, integrating an interactive theorem prover and an automated tool to reduce the verification efforts, emerges. Such hybrid approaches benefit from the high expressiveness and scalability of the theorem prover, and the automation of the model checker.

1.1 Formal Verification

Formal Hardware Verification is the proof that a circuit or a system (the implementation) behaves according to a given set of requirements (the specification) [34]. Any *formal verification approach* requires three components:

- The circuit (system) under investigation (called the implementation)
- The set of requirements this circuit should obey to (called the specification);
- The formal verification tool which is responsible of the verification process (Figure 1.1).

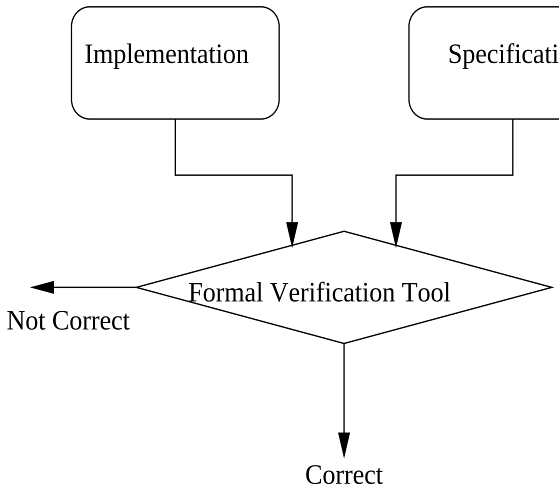


Figure 1.1: Formal Verification Approach

In practice, one needs to model both the implementation and the specification in the tool, and then uses one of the formal verification algorithms of the tool to check the correctness of the system or in some cases also to give a kind of trace (called counter-example) to where the error is. Formal methods have long been developed and advocated within the computing science research community as they provide sound mathematical foundation for the specification, implementation and verification of computer systems. These methods exploit representations with formally defined semantics in order to describe abstractly (independent of details of implementation) the desired functional behavior of a system [34]. Such formalisation methods provide precise and unambiguous system specifications which can be checked for completeness and internal logical consistency. The mathematical nature of these specifications enable reasoning about consistency (i.e., whether the system dynamics are consistent with system's static properties) and the deduction of consequences of the specification. Simulation, although widely used as a way of testing, could never give the verification coverage needed. Directive test benches, and random test benches are the ways adopted by simulation to get over this problem, but it is becoming clear that the quality of the validation achieved by traditional simulation is rapidly deteriorating microelectronic technology progresses. Thus, formal verification is proposed as a method to help certify hardware and software, and consequently, to increase confidence in new designs. Formally verifying designs may be cost effective in safety critical applications, for systems in high volume or remotely placed systems, and for systems that will go through frequent redesign because of changes in technology. Formal Verification however, is not the golden solution for circuit testing because of some limitations [26]. A correctness proof cannot guarantee that the real device will never malfunction; the design of the device may be proved correct, but the hardware actually built can still behave in a way unintended by the designer (this is the case for simulation too). Wrong specifications can play a major role in this, because it has been verified that the system will function as specified, but it has not been verified that it will work correctly. Defects in physical fabrication can cause this problem too. In formal verification a model of the design is verified, not the real physical implementation. Therefore, a fault in the modeling process can give false negatives (errors in the design which do not exist). Although sometimes, the fault covers some real errors.

Formal verification approaches can generally be divided into two main categories: reachability analysis, and deductive methods. Model checkers and equivalence checkers are examples of the first approach. Many different theorem provers (such as HOL) have been used for deductive verification.

1.1.1 Decision Diagram Based Methods

Reachability analysis approaches are internally categorised into two main flows: model checking and equivalence checking.

Model checking: In this approach, a circuit is described as a state machine with transitions to describe the circuit behavior. The specifications are described as properties that the machine should or should not satisfy. Traditionally, model checkers used explicit representations of the state transition graph, for all but the smallest state machines. To overcome this capacity limitation, different representations of BDDs (Binary Decision Diagrams) are used to represent the state transition graphs and this allows model checkers (such as SMV [24], and VIS [30]) to verify much larger systems. Still, these model checkers face the state space explosion problems while verifying large circuits [34].

Equivalence checking: In recent years, many CAD vendors offer equivalence checking tools for design verification. For example, Formality from Synopsys [17] performs logic equivalence checking of two circuits based on structural analysis. The common assumption used in the equivalence checking is that two circuits have identical state encoding (latches). With this assumption, only the equivalence of the combinational portions of two circuits must be checked. However, these tools cannot handle the equivalence of designs with no structure similarity. Another drawback of equivalence checkers is that they all need golden circuits, used as the reference to be compared with during the verification process. However, the correctness of golden circuits is still questionable.

The major advantage of the reachability analysis verification approaches is automation. The machine (tool) is usually responsible for building the whole model and automatically verifying either the equivalence or a property. But reachability analysis verification has two main drawbacks, namely, first the state exploration problem, where large designs (or deep datapaths) saturate the tool, stopping it from continuing the verification process, and second, is the problematic description of specifications as properties, specially in model checking, this description needs experience and sometimes may not give full system coverage.

1.1.2 Theorem Proving

With theorem proving, an implementation and its specification are usually expressed as first-order or higher-order logic formulae. Their relationship, stated as equivalence or implication, is regarded as a theorem to be proven within the logic system, using axioms and inference rules. Thus, theorem proving is a powerful verification technique. It can provide a unifying framework for various verification tasks at different hierarchical levels. However, the task of proving complex theorems needs expertise. A theorem prover or proof checker is a tool developed to partially automate the proof process or to check a manual proof. Theorem proving systems are being widely used on an industrial scale for hardware and software verification. Some of the well-known ones are HOL (Higher-Order Logic) [25], and PVS (Prototype Verification System) [6]. Theorem proving is considered a very strong verification tool because mathematical formulae can express nearly all design levels. The proof procedures are very efficient if they are constructed by experts. Also, hierarchical modeling is used to give theorem provers nearly unlimited power; especially in handling deep datapath designs, which can be modeled efficiently. The main problem with theorem proving techniques is the lack of expertise and documentation. It takes a considerably long time to learn and use theorem proving. Besides, there is a strong need for libraries of specifications to be established, and more automated tools and approaches.

1.2 Motivation

1.2.1 Verilog-HOL-MDG Project

As described before, each of the verification techniques has advantages and drawbacks. Hence, the combination of them in hybrid tools is expected to decrease the verification complexity. The work described in this thesis is part of a larger project to link Verilog [33], HOL and MDG as shown in Figure 1.2. Here, a Verilog model is passed through a HOL generator to get an equivalent model in HOL. The MDG tool provides four kind of verification approaches: the combinational equivalence checking, the sequential equivalence checking, the invariant checking, and the model checking. Within HOL, we use HOL tactics (proof scripts), called MDG_EQ_TAC and MDG_MC_TAC, to generate the required MDG files [1] and complete either the verification of combinational/sequential equivalence or model checking of the system. Since both Verilog and HOL provide the hierarchy, the description model will be written in a hierarchical way.

1.2.2 Linking HOL and MDG Equivalence Checker

In [28], and later [20] a hybrid tool and a methodology tailored to perform hierarchical hardware verification have been developed by the *Hardware Verification Group* of

[1] The contents and functions of these files will be explained in Chapter 2

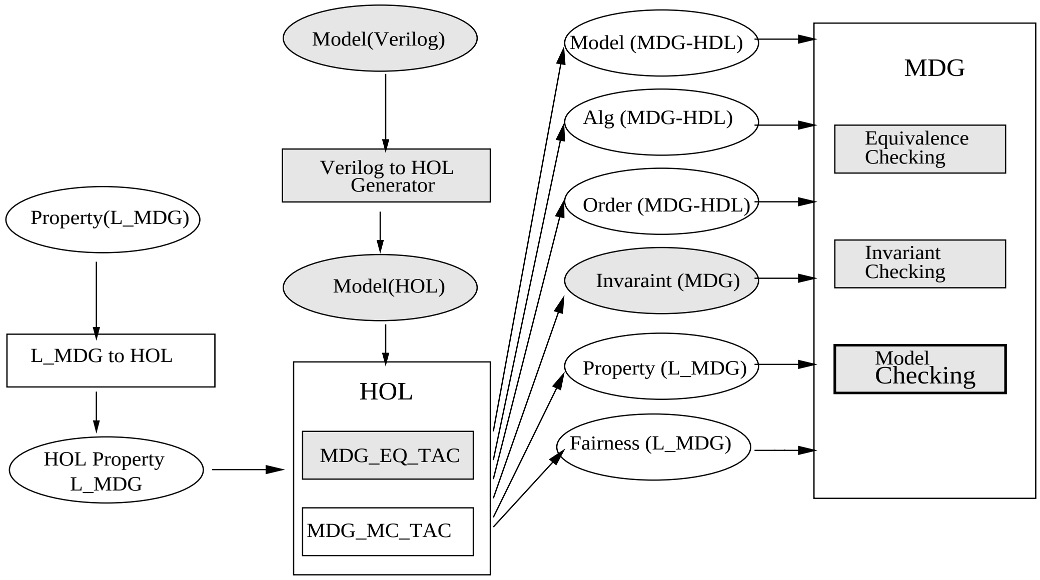


Figure 1.2: Intended Verilog-HOL-MDG Project Skeleton

Concordia University. They integrate the HOL theorem prover to the MDG equivalence checker. Similar to the project we are presenting, 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 structural specification of hardware implementation implies its behavioral specification, rather than checking properties or a partial specification. In fact, they use MDG to prove combinational or sequential equivalence. The hybrid tool integrates automated hardware verification with interactive hierarchical hardware verification. Verification using the hybrid tool proceeds as shown in Figure 1.3 [20]. An initial HOL goal is set to prove that the model implementation implies its behavioral specification. First, 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 are written in a hierarchical way, a tactic HIER_VERIF_TAC is called to break the design in sub-blocks. The same procedure is recursively applied if necessary. At any point, the goal proof can be done in HOL.

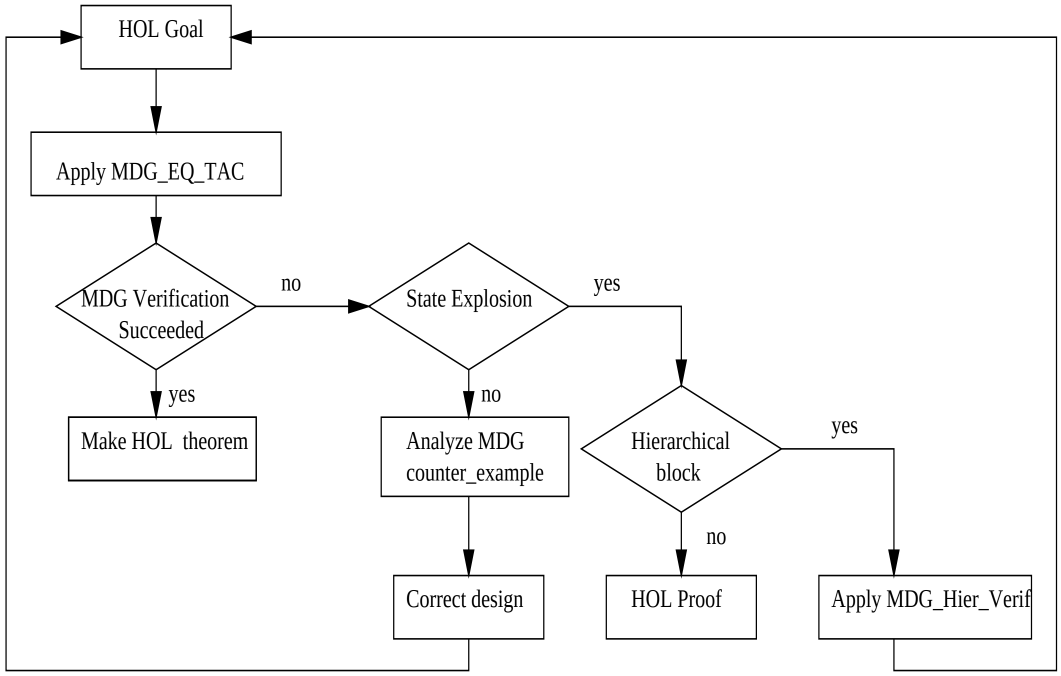


Figure 1.3: Hybrid HOL-MDG Tool for Equivalence Checking[20]

1.2.3 Linking HOL and MDG Model Checking

The earlier work presented above implements a tool linking the equivalence checking part of the MDG tool with HOL. Although this can be a step for the automation of HOL theorem prover, equivalence checking, as a technique, suffers from some drawbacks. Furthermore, it is sometimes useful to have the possibility to check properties within the theorem prover rather than the whole behavior. Usually to reduce the

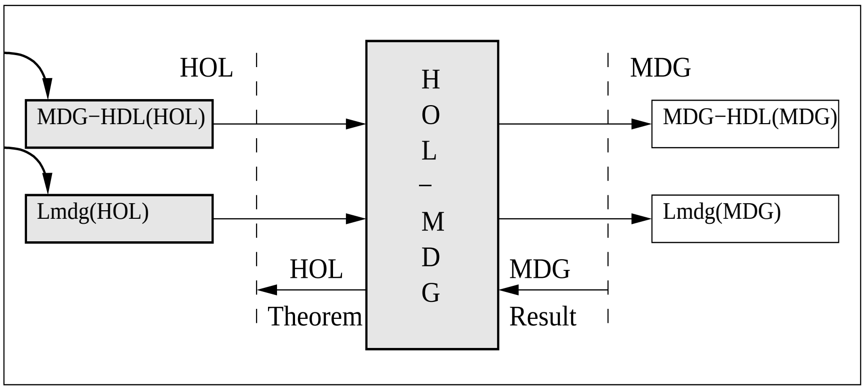


Figure 1.4: HOL and MDG Model Checker Interface

verification complexity, abstraction techniques and hierarchical verification are used. However, the tool in [20], although efficient in many ways, remains limited especially when it comes to abstract types.

The main contribution of MDGs is that they make the integration of implicit state enumeration and the use of abstract datatype and uninterpreted functions possible [5]. In order to benefit from the abstraction of MDG, we need to formalise the full input language for model description, MDG-HDL [40]. This formalisation is introduced in HOL as a new theory. Besides, we have to able to express MDG-like properties in the theorem prover. Since the input language of properties in MDG is the $\mathcal{L}_{mdg}$ language, we have to embed it in HOL as well. Finally, we need to implement the interface ensuring the communication between the two tools (cf. Figure 1.4).

1.3 Related Work

Since higher-order logic based theorem provers suffer from lack of automation, many projects were undertaken with the aim of linking theorem provers with model checkers or by embedding model checking packages into theorem provers. In our work, we choose the first alternative, as we proppose an interface linking the HOL theorem prover with the MDG model checker, allowing the definition and the verification of properties in HOL.

An impressive earlier hybrid system was the pioneering work of Joyce and Seger [18, 19] combined the theorem prover HOL with the symbolic trajectory evaluation tool VOSS. Symbolic trajectory evaluation provides a rigorous technique for verifying temporal relationship between node values, in addition of treating node values symbolically. In their system, several predicates were defined in the HOL system. 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 in SML as a remote function. It calls the VOSS system that is then run as a child process of the HOL system. A VOSS assertion can be expressed as a term of higher-order logic. Symbolic trajectory evaluation is used to decide whether or not the assertion is true. If it is the case, the assertion will be turned to a HOL theorem which can be used to proceed with further verification procedures. Zhu et al. [41] successfully applied HOL-VOSS for the verification of the Tamarack-3 microprocessor. As a continuation of HOL-VOSS, Aagarad *et al.* developed the Voss-ThmTac system combining ThmTac 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. The Voss-ThmTac system has been used to verify successfully an IA-32 Instruction length decoder.

Rajan *et al.* [29] proposed an approach for the integration of propositional μ-calculus model checking, based on BDDs, within an automated proof system PVS [6]. They used μ-calculus as a medium for communicating between PVS and the model checker. It was formalised by using the higher-order logic of PVS. The temporal operators that apply to arbitrary state spaces are given the customary fixed-point definitions using the μ-calculus. These expressions were translated to the form required by the model checker. This later was then used to verify the subgoals generated within PVS. In [13], a complicated communication protocol was verified by means of abstraction, used to extract a finite-state abstraction of the protocol that preserves the property of interest.

The MEPHISTO system [21] was developed to manage the higher levels of a verification, producing a first-order subgoals to be proved by the FAUST first order theorem prover. MEPHISTO is a hardware-specific tool that converts the original goal into a set of simpler subgoals, which are then automatically solved by a general-purpose theorem prover. MEPHISTO gives some support for hierarchical proof procedures providing a library of pre-proved modules.

In a later work, Schneider and Kropf [32] presented a verification method which combines the advantages of deduction style proof systems like HOL with those of traditional model checking approaches. Datapath oriented verification goals involving abstract datatypes are expressed by a class of higher-order logic, which allows a unified description of hardware structure and behavior at different levels of abstraction.

Hurd [15] used PROSPER [2] [9] to combine the Gandalf, a first-order theorem prover, with HOL. A HOL tactic, GANDALF_TAC, is used to enable first-order HOL goals to be proved by Gandalf and mirror the resulting proofs in HOL. Gandalf is a PROPSPER plug-in that can be called over a network, and a Gandalf server may be set up servicing multiple HOL clients. GANDALF_TAC takes the original goal, converts it to a normal form, writes it in 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.

Scheinder and Hoffmann [31] 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. HOL terms are exported to SMV through the PROSPER plug-in interface. On successful model checking, the results are returned to HOL and turned to theorems. This hybrid tool allows SMV to be used as a HOL decision procedure. The deep

[2]Prosper provides an open proof architecture for the integration of different verification tools in a uniform higher-order logic environment

embedding of the SMV specification language in HOL allows LTL specifications to be manipulated in HOL.

Gordon [10] integrated HOL with the BuDDy BDD package. His aim was to provide a platform for implementing intimate combinations of deduction and algorithmic verification, like model checking. HOL was used to formalise the Quantified Boolean Formulae of BDDs. By using a higher-order rewriting tool, the formulae can be interactively simplified to get a smaller BDDs. The mapping of simplified formulae to BDDs was done using a table. The BDD algorithm can also strengthen its deductive ability in this system. In a later work [12], Gordon describes some experiments in adding simple model checking infrastructure to the HOL98. The main difference between this approach and other tools mentioned above is that the tool provides a secure and general programming infrastructure to allow users to implement their own BDD-based verification algorithms and then to integrate them with existing HOL98 system.

Similar to [15, 19, 29], we integrate a theorem prover (HOL) to an existing hardware verification tool (MDG) rather than embedding an external package within the system as done in [10] and [31]. We work within the proof system but using the specification style of the automated tool. This is done by embedding the languages of the automated verification tool within the proof system. An additional novel aspect in our work is the explicit support of model reduction based on the natural design hierarchy and the specification to verify. The use of MDG as the automated tool compared to related BDD tools is to our opinion a big asset that opens up interesting possibilities of making use of MDG features for data abstraction. Thus pushing up the abstraction level of what can be passed to the automated tool from the theorem prover and ultimately allowing larger datapaths to be dealt with automatically.

More recently, Gordon [11] presented an embedding of the semantics of the properties specification language Sugar2.0 [16] in higher-order logic supported by HOL. The motivation of this work is mainly proving meta-theorems with a theorem prover to provide a deeper kind of sanity checking, and developing machine readable semantics. Another advantage is the fact that Sugar provides ways to specify properties in both simulation and formal verification, providing the users with an interface to combine formal verification techniques, both theorem proving and model checking, with simulation techniques. Similar to our project, this embedding gives a way to specify properties in HOL, the $\mathcal{L}_{mdg}$ language in our case. While [11] focuses on the formalisation of Sugar in HOL, in our project we further enable the verification of the property outside HOL, using the MDG model checker.

1.4 Scope of the Thesis

The remainder of this thesis is organised as follow. In Chapter 2, we overview the MDG and HOL verification systems, emphasising the difference between theorem proving and model checking approaches, the advantages as well as the verification process of each. In Chapter 3, we describe our HOL-MDG linkage approach explaining the way we embedded MDG input languages into the logic of the HOL interactive theorem prover. Chapter 4 presents the implementation of the tool, its structure and its functionality. Chapter 5 illustrates the advantages of our hybrid approach through a case study on an Island Tunnel Controller (ITC). And finally, conclusions and future work will be discussed in Chapter 6.

Chapter 2

HOL and MDG

In this chapter, we give an overview of the linked tools: the HOL theorem prover and the MDG system.

2.1 The HOL Theorem Prover

The HOL theorem prover, developed by Gordon [25], is an interactive proof assistant that has been under development since mid-1980's, and is based on ideas from the Edinburgh LCF project [23]. The LFC approach implements a logic in a strongly typed programming Meta Language (ML) [27]. The HOL system is based on higher-order logic and was originally intended for hardware verification. Thanks to its generality, HOL is being currently used in a variety of application areas. The basic interface to the system is ML [27].

HOL offers two proof styles: *forward* and *goal-directed backward* proofs in a natural-deduction-style calculus by creating theorems and applying inference rules to the already created theorems. In the forward proof style, inference rules are applied in sequence to previously proved theorems until the desired theorem is obtained [25]. This approach has some problems since it is hard to know where to state the proof and, for large proofs, to determine which sequence of rules to apply [26]. In backwards proofs, the user sets the desired theorem as a goal, applies tactics to split it in subgoals in such a way that if a corresponding inference rule was applied to the subgoals, the theorem of the goal will be obtained. A tactic is an ML function that when applied to a goal reduces it to a list of subgoals, along with a justification function mapping a list of theorems to a theorem [25]. In practice, a mixture of these two proof styles is used, with forwards proof interspersed within backwards proofs.

Our tool links HOL98 to MDG model checker. HOL98 is the third version of HOL system. Its key idea is that theorems are represented as an abstract ML types whose only pre-defined values are axioms, and whose only operations are inference rules. Theorems in HOL are built either by setting axioms or by applying rules of inference to axioms or to existing theorems; also a proved goal is set to theorem; hence the consistency of the logic is preserved [2]. The HOL98 system provides a range of pre-proved theorems and a set of pre-defined tools, which represents a rich initial environment. In addition, users can enrich it by building their own theories. A theory defines a set of types, operators, axioms, and rules to deal with them.

Usually, a theory is not independent as it needs to interact with other theories.

HOL allows hierarchical verification wherein design modules are divided into submodules and the submodules are divided too until the lowest implementation level is reached. To prove that the implementation of a module implies its specification, the user should prove the implication of the implementation and the specification of each submodule. The main advantage of hierarchical theorem proving is the ability to deal with large scale design. Despite of the expressiveness power of higher-order logic and the feature HOL system is offering, the verification process is still a cumbersome task since it needs very deep understanding of the design structure and a googd mastering of higher-oder logic and HOL, which make the verification time-consuming.

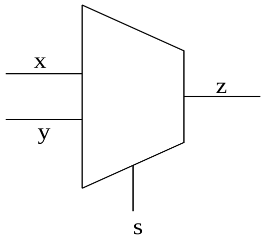
2.2 The MDG System

The MDG system is a decision diagram based verification tool, primarily designed for hardware verification. It is based on Multiway decision Graphs [5], which are an extension of ROBDD (Reduced Ordered Binary Decision Diagrams) [3] by abstract sorts and uninterpreted functions. The MDG tool is written in the logic programming language Prolog. Also, it runs under Quintus Prolog V3.2. The advantage of this tool is the fact of implementing a black box verification technique.

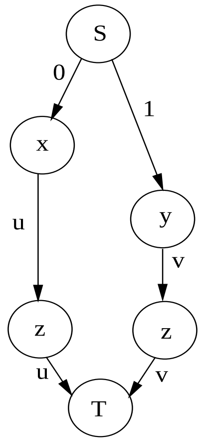
Multiway decision graphs [?] represent a new class of decision diagrams, proposed to overcome the limitation of the ROBDD-based methods. These latter require a binary representation of the circuits. The idea behind MDGs is to introduce abstract sorts and uninterpreted functions such that the model checking can be done on larger state spaces.

The MDG language is based on an ordinary many-sorted first order logic. The vocabulary consists of sorts, constants, variables and function symbols. The conjunction, disjunction or composition of the latter are defined as *terms*. Since the logic is typed, for each defined term, a type is assigned. For ROBDDs, the formulae are of propositional logic and the leaf nodes of their associated diagrams labelled by 0 or 1. The extension to MDGs is done in such a way that the leaf nodes are labelled by formulae, allowing the nodes to range over abstract sorts. An MDG is a *finite acyclic directed graph* G where leaf nodes are labelled by formulae, the internal ones are labelled by terms, and the edges issuing from an internal node N are labelled by terms of the some sorts as the label node N. Each formula P is represented by a graph G. So, when from as node there is a multiple edge $B_1,\ B_2...$ issuing from it corresponding and leading to the subgraphs $G_1,\ G_1...$, which represent respectively to the formulae $P_1,\ P_2\ ...$, then the whole graph G is obtained by the conjunction of all the subgraphs.

As an example, we show the MDG of a multiplexer (Figure 2.1). We declare the input signals and the output to be of the same abstract sort, then we define abstract constant for each input and finally we set the order of variables to take into account in constructing the MDG.



(a) MUX



(b) MDG

Figure 2.1: Multiplexer Example

This MDG is suitable for any signals of type *wordn* we define.

The MDG tools accept hardware description in Prolog-style Hardware Description Language, called MDG-HDL [37], which allows the use of abstract variables for representing data signals. This MDG-HDL description is then compiled into the ASM (Abstract State Machine) [4] model represented by internal MDG data structures. An ASM is defined as a tuple D= (X, Y, Z, F_I, F_T, F_O) where X represents the set of input variables, Y represents the set of state variables, Z represents the set of output variables, F_I denotes the set of initial variables, F_T represents the transition relation, and finally F_O denotes the output relation.

MDG-HDL supports structural descriptions, behavioral ASM descriptions, or a mixture of both. The MDG tool contains mainly a combinational verification module, sequential verification module, reachability analysis module and an MDG package. The latter implements manipulation algorithms for MDGs. The reachability analysis algorithm checks that an invariant holds in all the reachable states of an ASM using the abstract implicit enumeration technique.

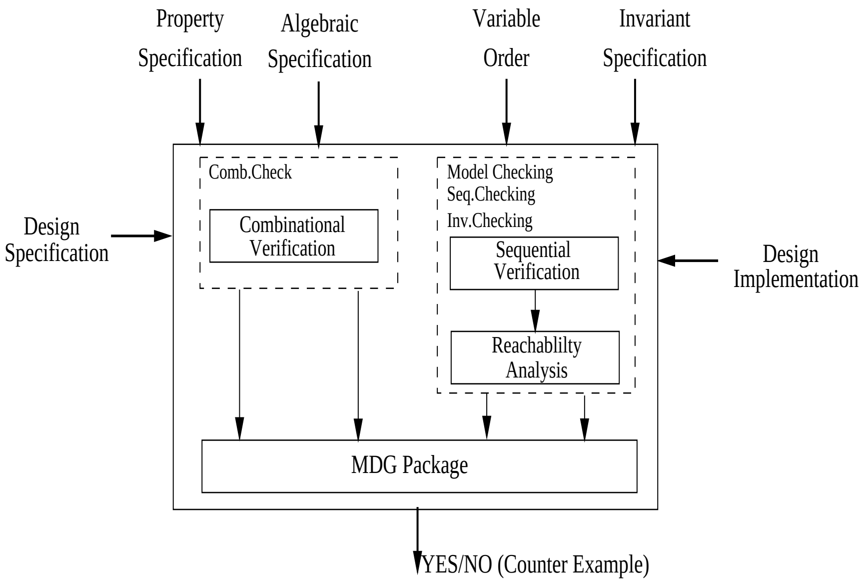


Figure 2.2: The MDG Tool

Interacting together, these modules provide four applications: combinational/sequential equivalence checking, invariant checking, and model checking. For the two first operations, the user should provide the tool with :

- A behavioral model: it is given by a tabular representation of the transition/output relation or a truth table.

- A variables ordering file: it contains a total order of all variables, functions to follow in the construction of the MDGs.

- An algebraic file: it contains the different sorts, functions and terms specified in the description model.
- An implementation file: it is usually a netlist of components (predefined in MDG-HDL) connected by signals.
- An invariant file: Since from the two models to verify, we build a product machine, we impose that a given output should be equal to its correspondence in the second circuit.

In addition, both behavioral and structural description files contain the signals and their sorts, the output partition and the next partition in the case of sequential circuit. However, the verification algorithms are different. For the combinational verification, an MDG of each model is computed. Thanks to the canonicity of the MDGs, the equivalence checking holds if the obtained MDGs are the same. If it is not the case, the equivalence checking is failed. However, for sequential checking, the verification is achieved by forming a circuit out of two circuits, feeding the same inputs to them and verifying an invariant asserting the equality of the corresponding outputs in all reachable states.

Invariant checking is achieved by using the symbolic reachability analysis [39] technique. The algorithm makes sure that a given invariant holds over all the reachable states. Using this operation, the verification of safety properties become possible.

Finally, MDG encapsulates a model checker for safety and liveness properties. It is based on implicit enumeration technique of the abstract state machine. The input files are either the implementation or the specification of the circuit, the algebraic description, the order variables and the property to be verified. In MDG model checker, the design is represented by ASMs and the properties to be verified are expressed by formulae in the first order ACTL-like [37] temporal logic, called $\mathcal{L}_{mdg}$[36]. The ASM model of $\mathcal{L}_{mdg}$ is composed of the original design model along with a simplified invariant. This model is constructed before interacting with the MDG. Furthermore, additional information is required to verify the property: the user should provide the tool with the type of the property as well as the fairness constraints imposed. Finally, the simplified invariant is checked on the composite machine using the implicit abstract enumeration of ASMs. More information are required by the model checker to verify the property. When a verification fails, the MDG tool returns a counterexample, consisting of the state trace from the initial state to the faulty one. However, this feature is still not provided in the tool for the model checking operation.

As part of the MDG software package, the user is provided with a large set of pre-defined modules such as logic gates, multiplexers, registers, bus drivers, etc. Besides the logic gates which only use Boolean signals, all the other components allow signals with concrete as well as abstract types. Moreover, a special structure is defined called tables. Tables can be used to describe functional blocks in both implementations and specifications. A table is similar to the truth table, it has as entry values first order terms in the rows. It is composed of a list of rows. Each row is a list of inputs values and their corresponding output. A default value of the output is defined if the inputs sequence we have doesn't fit the defined rows. Some constraints are imposed in the table inputs. The first list contains variables and cross-terms (where the output of a given function is concrete while its inputs contain at least is o abstract sort). The last element of the list must be a variable (either concrete or abstract). The other variables in the list must be concrete variables. The remaining lists consist of the sets of values that the corresponding variables or cross-terms can take. The last element in the list of values could be a first-order term. This represents an assignment to the output variable. The other values must be either 'don't care' (represented by '*') or individual constants in the enumeration of their corresponding variable sort. The last element in a table is the default value. To illustrate this, we present in table 2.1 the table representation of the multiplexer described above. The table is describing the behaviour of an abstract multiplexer.

IsEqual(x,u)	**IsEqual(y,v)**	**select**	**z**
1	*	0	u
*	1	1	v

Table 2.1: **Mux MDG table**

Due to the constraints explained before, the entry of the table couldn't be an abstract term. To overcome this, we define cross-term function *IsEqual* that takes as input x and its generic constant u and returns a boolean type. The same thing is done for the input y. So, the table structure becomes:

```
table([[select, IsEqual(x,u), IsEqual(y,v),z],
        [0,1,*,u],
        [1,*,1,v]|u])
```

We choose u to be the default value. This table description is further internally translated into an MDG (decision diagram) with the variable ordering s, x, y and z. However, we have no restriction on the inputs and outputs in definition of components. The MDG component definition is :

component(mux1, mux(sel(s), inputs([(0,x),(1,y)]), output(y)))

Chapter 3

The Formalisation of the MDG Input Languages in HOL

The aim of our work is linking the HOL theorem proving with the MDG model checking. Both the model description and the property are given to HOL system. For processing a model checking operation, the theorem prover has to interact with MDG model checker and pass the required files to the latter. So, in a certain way, we are restricted by the input languages of the MDG tool. To express an MDG like specification and properties in HOL, we have to embed two languages:

- MDG-HDL(for model description).
- $\mathcal{L}_{mdg}$ (for properties specification).[1]

3.1 Formalising MDG-HDL Grammar in HOL

As presented in Chapter 2, a special module called table is used to specify behavioral description in MDG. In [8], the table structure as well as the MDG-HDL components

[1]Subset of the embedded Hol theories is presented in the Appendix A and B

library has been embedded in HOL, allowing the specification of concrete circuits descriptions. Since there is no embedding of the MDG grammar, we were unable to define descriptions, where abstract sorts, uninterpreted functions, and cross-terms are declared. In the following section, we present the subset MDG-HDL library formalisation previously embedded, then we expose the HOL theory we embedded to cover the full MDG-HDL grammar.

3.1.1 Formalising the MDG Library in HOL

MDG-Tables Definition in HOL

The proposed embedding of MDG table in HOL consists of considering the MDG-table specified by five arguments. The first argument is a list of the inputs, the second is the single output, the third is a list of table rows. Each row is a list itself, giving one allocation of values to the inputs. The entries in the list can be either actual values or a special don't-care marker. The latter matches any value the input could hold. The fourth argument is a list of output values. Each is the value on the output when the inputs have the values in the corresponding row. The final argument is the default value, taken by the output if the input values do not match any row. The first step in formalising this definition is to define the matching of input values. These can be either a normal value of arbitrary type or a don't-care value. This latter expresses the fact then the output is independent from the value of this input in the current inputs row. The values taken by a table are defined as a new HOL type, with associated destructor function to access the value.

```
⊢def Table_Val = TABLE_VAL of 'a   |  DONT_CARE

⊢def TableVal_to_Val(TABLE_VAL(v:'a))= v
```

The first HOL expression define a new HOL datatype "Table_Val", which has two constructors : *TABLE_VAL* and *DONT_CARE*. The former can take any type. Curzon *et al.* [8] defined the matching of input values to table values. A match occurs if either the table value is don't-care, or the value on the input is identical to the table value. This property must hold for each table entry. It is defined recursively by the function *table_match*.

```
⊢def (Table_match inputs [ ] (t:num) = T)

        ∧ (Table_match inputs (CONS v vs) t) =

                     (((HD(inputs) t) = TableVal_to_Val (v:'a Table_Val))

                     ∨ (v = DONT_CARE))

                     ∧ (Table_match (TL inputs) vs t)
```

HD and TL are two predefined HOL functions which return respectively the head and the tail of a list. The test is first done on the first element in the input list. It is repeated after that on the rest of the list, until reaching the empty list. Moreover, if there is a match on a given row, the output has the corresponding value. Otherwise, it must check the next row. If there is no match, the output equals the default value. This is defined in a recursive manner on the input list as the relation table:

```
⊢def (table inps (out:num -> 'b) ( [ ]:('a Table_Val list) list)
                        V_out default t =  (out t = default t))
        ∧ (table inps  out (CONS v vs) V_out default t =
        ((Table_match inps v t) →  (out t = (HD V_out)t)))
        |(table inps out vs (TL V_out) default t)))
```

A given table will relate a given input to a given output, if the table relation is true at all the times:

```
⊢def TABLE inps (out:num -> 'b) (V_outs:('a Table_Val list) list)
        V_out default = ∀t. table  inps out V_outs V_out default t
```

The given definition is less flexible than the MDG tables one since, here, all the input variables must be of the same type, while they can be from different sorts in the MDG system. This is why Curzon *et al.* [8] choose to reserve a list for the output instead of specifying the input like in the MDG tool: the last element in the row correspond to the output value of the corresponding inputs.

The MDG Components Definition in HOL

The MDG library comes with a predefined set of components. Since the abstract sort is not handled in the tool developed by Kort *et al.* [20], only components, where their inputs and outputs are of concrete sorts, are defined. All the inputs and outputs are declared as signal from type number to Boolean, dependent from the variable t (time). For example, the *fork* component [8] was defined in a concrete way:

```
mdg_fork x y = ∀t. (x:num->bool) t = y t
```

Besides, the behavior of each component is defined in term of tables. As an example, the corresponding table of the *fork* component is:

```
FORK_TABLE x y = TABLE x:num->bool] (y:num->bool)
                                  [ [ TABLE_VAL F];
                                  [ TABLE_VAL T]]
                                  [ FSIG;TSIG] FSIG
```

With the above exposed formalisation of the MDG-HOL library, there is no possibility to express MDG terms containing abstract functions, generic constants, or abstract variables. Therefore, we propose an embedding of the MDG *grammar* syntax in HOL. This required major modifications to the pre-introduced theory. In next section, we first describe the grammar of the MDG-HDL and then we expose its corresponding embedding in HOL.

3.1.2 Embedding the MDG-HDL Grammar in HOL

MDG-HDL Grammar

MDGs incorporate variables of abstract types to denote data values and uninterpreted function symbols to denote data operations. MDG terms are well formed first-order term.The wellformedness condition prescribes that MDG formulas should be in the form of directed formulas [5]. Let $\mathcal{F}$ be a set of function symbol and ν a set of variables. We denote the set of terms freely generated from $\mathcal{F}$ and ν by $\tau(\mathcal{F}$,$\nu)$.

The syntax of a directed formula is then given by the grammar below [1]:

$$
\begin{array}{lcl}
Sort\ \mathcal{S} & ::= & S \mid \underline{S} \\
Abstract\ Sort\ S & ::= & \alpha \mid \beta \mid \gamma \mid ... \\
Concrete\ Sort\ underlineS & ::= & \underline{\alpha} \mid \underline{\beta} \mid \underline{\gamma} \mid ... \\
Generic\ Constant\ C & ::= & a \mid b \mid b \mid ... \\
Concrete\ Constant\ underlineC & ::= & \underline{a} \mid \underline{b} \mid \underline{c} \mid ... \\
Variable\ mathcalX & ::= & V \mid \underline{V} \\
Abstract\ Variable\ V & ::= & x \mid y \mid z \mid ... \\
Concrete\ Variable\ underlineV & ::= & \underline{x} \mid \underline{y} \mid \underline{z} \mid ... \\
Directed\ Formula & ::= & Disj ::= Conj \vee Disj \\
Conj & ::= & Eq \wedge Conj \mid Eq \\
Eq & ::= & \underline{A} = \underline{C}(A \in \tau(\mathcal{F}, \nu)) \\
 & & \mid \underline{V} = \underline{C} \\
 & & \mid V = A(A \in \tau(\mathcal{F}, \nu)) \\
 & & \mid \top \\
 & & \mid \bot
\end{array}
$$

As in ordinary many-sorted first-order logic, the vocabulary consists of a generic constants, concrete constants, abstract variables, concrete variables and function symbols. Directed formulae are always disjunctions of disjunctions or conjunctions of equations. The conjunction *Conj* is defined as be conjunction of at least two equations *Eq*. Atomic formulae are the equations, generated by the clause Eq, plus $\top$ (truth) and $\perp$ (false). The equation can be the equality of concrete term and a concrete constant, the equality of a concrete variable and a concrete constant,or the equality of an abstract variable and an abstract term.

Embedding in HOL

In HOL, we define an abstract sort to be of type α to *string*. The second parameter in this definition is specified mainly to permit the user to impose a specific MDG sort. A concrete sort (Boolean sort included) is defined by the list of its enumerated values.

```
⊢def MDG_sort =   ABSTRACT of 'a ->string
                     |CONCRETE of string ->string list
```

Next, predicates are defined to specify the type of the sort we are dealing with.

```
⊢def  (IsConcreteSort (ABSTRACT Abs MDG_name) = F )∧
                              (IsConcreteSort (CONCRETE Conc val_list) = T)
```

IsConcreteSort returns true if the type is of concrete sort. Similary, we define a predicate to determine abstract sorts.

```
⊢def  (IsAbstractSort (ABSTRACT Abs MDG_name) = T )∧
            (IsAbstractSort (CONCRETE Conc val_list)= F)
```

A variable is defined according to its type, concrete or abstract. It is defined as a new Hol datatype:

```
⊢def MDG_var = MDG_VAR of string → MDG_sort
```

To test the sort of the variable, we should fix its MDG_sort:

$\vdash_{def}$ `IsConcreteVariable (MDG_VAR name sort)= IsConcreteSort sort`

A function is defined by its domain which is a list of concrete variables, abstract variables or a mixture of both, and its range, which is a unique output. The type of the function is determined according to its domain and range. If the output is from an abstract sort, so the function is defined to be an abstract function. If all the inputs and the output are from concrete sort, so the function is defined to be concrete. And finally, if the output of the function is concrete, and at least one of its inputs is abstract, the function is defined to be cross function.

$\vdash_{def}$ `MDG_Fun = MDG_FUN of  string →  MDG_VAR list →   MDG_VAR`

Some predicates are set to determine the kind of the function we define: abstract, concrete or a cross function. Since the domain of the function is a list of variables, to test if the function is concrete, we should test if the inputs and the outputs are of concrete sort. So, we define a predicate to determine recursively if the list is of concrete variables. The test is first done on *h*, the head of the list, and is repeated recursively on *tl*, the tail of the list, until reaching the empty list.

```
⊢def  ConcreteVarList(h::tl) = ((IsConcreteVar h)    ∧
                                (ConcreteVarList tl))∧
                                (ConcreteVarList  [ ] = T)
```

Hence, a function is concrete if both its domain and its range are concrete:

```
⊢def  concreteFunc (MDG_FUN name InputVarList OutputVar) =
          (ConcreteVarList InputVarList) ∧
          (IsConcreteVariable OutputVar)
```

After defining one by one the different elements of the MDG vocabulary, it is possible to define the different kinds of *MDG_terms*. An *MDG_term* is either:

- a concrete constant, *CONC_Const*, one of the concrete sort enumeration,
- a generic constant, *GEN_Const*, constant defined for an abstract sort,
- a variable, *VAR_Term*, either from concrete sort or abstract sort, or
- a function, *FN_Term*, from the *MDG_Fun* HOL datatype defined above.

The latter is done using the constructor *TERM*. It takes as argument a defined *MDG_Term* and returns an *MDG_Term*.

The HOL definition is:

```
⊢def MDG_term = GEN_Const of 'a
                  | CONC_Const of string
                  | VAR_Term of  MDG_VAR
                  | FN_Term of  MDG_Fun
                  | TERM of MDG_term => MDG_term
```

The overall structure of the table defined in Section 3.1.1 will not be changed. However, we impose that the entry of the table should be either a don't care or an MDG term.

```
⊢def Table_Val = TABLE_VAL of  'a MDG_term  | DONT_CARE
```

With the above embedding of the MDG-HDL grammar, it is now possible to define components of MDG library that contain abstract variables and functionsymbols.

For instance, we added the components multiplexer, register, and transform. For example, the multiplexer component is defined as follow:

```
⊢def mdg_mux  x1  x2 (y:num->bool)  z  =

        ∀ t . z(t) = if ( y t) then (x2 t)

                     else x1(t)
```

In the following section, we present an illustrative example of an abstract counter using the above embedded theory. We proved, using HOL, the equivalence of the specification and the implementation of the counter.

3.1.3 Example of an Abstract-Counter

We Consider a synchronous circuit which consists of a data register *count*, two multiplexers *mux1* and *mux2*, and three functional blocks symbols *inc*, *dec*, and *eqz*. The uninterpreted functions *inc* and *dec* take as input *count* of abstract sort and produce an abstract output *inc(count)* and *dec(count)*, respectively. The cross-term *eqz* takes as input *count* and produces a concrete output of sort *bool*. *y*, the select signal of the multiplexer, is the input of the counter. We consider *count* the output of the counter. The transition relation of this machine is as follow:

$$\begin{aligned}\mathcal{R} \equiv \; & [((\underline{y} = \underline{0}) && \wedge count' = inc(count)) \quad \vee \\ & [((\underline{y} = \underline{1}) \wedge eqz(count) = \underline{\mathit{0}} && \wedge count' = dec(count)) \quad \vee \\ & [((\underline{y} = \underline{1}) \wedge eqz(count) = \underline{\mathit{1}} && \wedge count' = count)\end{aligned}$$

Our objective is to verify in HOL that the implementation of the counter implies its specification.

Counter specification

The HOL specification of the abstract counter contains an abstract sort, two abstract functions, and a cross function. The behavior of the abstract counter is summarised in Table 3.1, where *state* and *n_state* represent the *count* and *count'* respectively, *pc_val*, *inc_pc_val*, and *dec_pc_val* are generic constant of the same abstract sort *PC*. *eqz* represents *eqz(count)*.

In HOL, the output of the tables should be defined as signals. Hence, we define them as function of time [2] :

```
⊢def pcSIG   = λ(t:num).pc_val

⊢def decSIG = λ(t:num).dec_pc_val

⊢def incSIG = λ(t:num).inc_pc_val

⊢def dec_incSIG = λ(t:num).pc_val
```

Before writing the table, we have to homogenise its inputs. Therefore, we define functions to map from the initial type to the desired type. As an example, we give here the definition the *bool_to_MDGTerm* function, which maps the boolean type to a concrete MDG type.

```
⊢def bool_to_MDGTerm:(bool-> string MDG_term) b =

          if (b = T) then  (CONC_Const "T")

          else

          (CONC_Const "F")
```

The table of the counter specification is shown below:

[2] λ t. x means that x is function of t.

```
⊢def COUNTER_TABLE (state) (v:(num->bool)) (y:(num->bool))(n_state) =
 TABLE  ; bool_to_MDGTerm  o  v ;bool_to_MDGTerm o y ]
      (n_state o SUC)
  [[ TABLE_VAL (pc_val); DONT_CARE;TABLE_VAL (CONC_Const "F" ) ];

    [ TABLE_VAL (inc_pc_val);TABLE_VAL (CONC_Const"F");
      TABLE_VAL (CONC_Const"T") ];

    [ TABLE_VAL (pc_val);TABLE_VAL (CONC_Const"F");
      TABLE_VAL(CONC_Const"T") ] ;

    [ TABLE_VAL (pc_val);TABLE_VAL (CONC_Const"T");
      TABLE_VAL(CONC_Const"T") ] ]

  [ incSIG;dec_incSIG;decSIG;pcSIG ] pcSIG
```

The first tree rows represent the possible inputs combination. The fourth list represents the output for each row respectively. And finally, the *pc_val* is the default value of the output if the input sequence is different from what is specified. In terms of truth table, the counter table specification is equivalent to the Table 3.1.

state	**eqz**	**y**	**n_state**
pc_val	*	F	inc_pc_val
inc_pc_val	F	T	pc_val
pc_val	T	T	pc_val
pc_val	F	T	dec_pc_val

Table 3.1: **Abstract Counter Behavior**

Counter Implementation

The implementation is composed of two multiplexers, one register for the *n_state*, two (black-box) uninterpreted functions *Dec* and *Inc*, and finally one transform function for the cross operator *Eqz*. In addition, we need to initialise the state value so, we add an Initial predicate that sets the variable state at *pc_val* (c.f Figure 3.1).

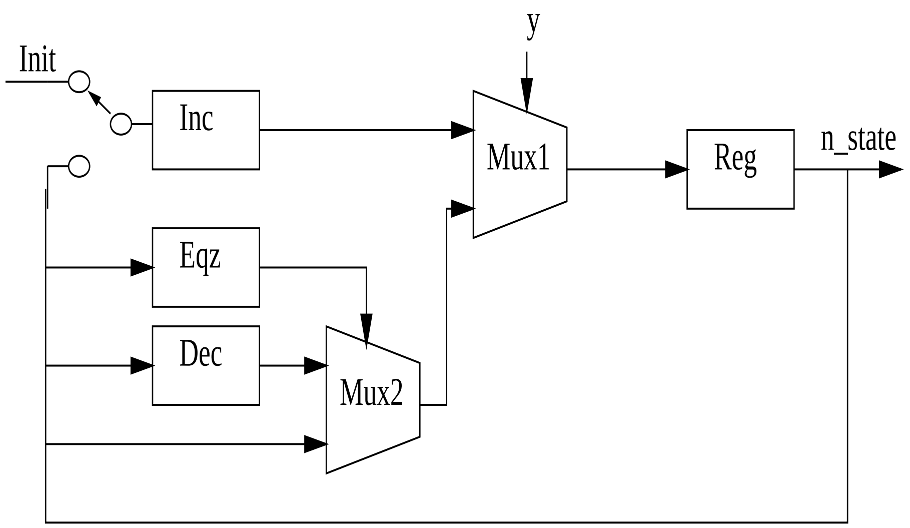


Figure 3.1: Abstract Counter Implementation

```
⊢def  Counter_IMP (v) (y:num -> bool) (n_state) =
      ∃x  z w (state:num->string MDG_term)  state1.
          (Reg state1 n_state) ∧
          (Mux1  x z (y:num -> bool)  state1 ) ∧
          (Inc n_state x) ∧
          (Mux2  n_state w (v)  z) ∧
          (Dec ( n_state) (w)) ∧
          (Eqz ( n_state) (v)) ∧
          (Initial state)
```

Counter Verification

The goal to be proven in HOL is stated as the folowing implication:

```
!state v y n_state pc_val .

  (Counter_IMP  v y n_state) ==> (COUNTER_TABLE   state v y n_state)
```

The proof should be done for any generic constant our counter takes. The proof was conduct such that all definitions are first rewritten, and then using a combination of two predefined HOL tactics: the ARITH_TAC tactic and PROVE_TAC. The First one is used to split the goal to several subgoals. After that each subgoal is proven individually. The original goal is proven when all the subgoals are proven.

3.2 Formalising $\mathcal{L}_{mdg}$ into HOL Syntax

3.2.1 The $\mathcal{L}_{mdg}$ Syntax

CTL (Computation Tree Logic), is a propositional branching time temporal logic, widely used as a property specification language for model checking. In CTL, each linear time operator (F, G, X, or U) must appear after a path quantifier A (for all paths), and E (there exists a path). CTL* extends CTL by allowing temporal operator in which a path quantifier is followed by an arbitrary linear time formula. Thereafter, properties such $\mathbf{A(p \wedge Xq)}$ are allowed in CTL* while, not allowed in CTL. Xu [37] defined an Abstract_CTL* logic, called $\mathcal{L}_{mdg}$. This logic extends CTL* by using the first-order logic rather than the propositional logic. $\mathcal{L}_{mdg}$, however,is a subset of the first order Abstract_CTL* [36]. $\mathcal{L}_{mdg}$ is the properties specification language using for the MDG model checker. The properties allowed in $\mathcal{L}_{mdg}$ can have the following templates:

Property :

A(Next_let_formula)

| AG(Next_let_formula)

| AF(Next_let_formula)

| A(Next_let_formula)U(Next_let_formula)

| AG((Next_let_formula) $\Rightarrow$ (F(Next_let_formula)))

| AG((Next_let_formula) $\Rightarrow$((Next_let_formula) $\Rightarrow \cup$ (Next_let_formula)))

Only the universal path quantification is possible with the current version of MDG model checker. The syntax of the existential path is still not defined. The Next_Let_Formula is defined to be a nesting formula, or a basic formula.

Next_let_formula:

X(Next_let_formula)

| LET (Let_equation) IN (Let_equation)

| Next_let_formula (with concrete variables only) ⇒ Next_let_formula

| Next_let_formula Next_let_formula

| Next_let_formula (with concrete variables only)

| Basic_formula

Basic_formula:

Lterm = Rterm

| True

| False

Lterm : ASM_variable_Name

Rterm : ASM_variable_Name

| OrdVar_Name

| IntegeConstant

| SymbolicConstant

| Function

Let_Equation ::=

Let_equation Let_equation

|(Let_equation)

| OrdVar_Name = ASM_variable_Name

Function ::= Function_Name (parameter_list)

The parameters of the function can be either ordinary variables or functions. The *Let_equation* can be the disjunction of *Let_equations* or an equality of an ordinary variable and an *ASM variable*. The *Basic_formula* is true, false or the equality of *LTerm* and *RTerm*. An *Rterm* can be a variable , constant or a function. However, the *LTerm* is an *ASM Variable*. ASM variables, constant variables and the symbolic constants represent both the set of variables (concrete and abstract) and the set of constants (concrete and abstract).

3.2.2 Embedding $\mathcal{L}_{mdg}$ in HOL

In order to embed the $\mathcal{L}_{mdg}$ in HOL, it is important to respect the semantics of the original language [37]. All properties are defined according to two notions: *path* and *state*. A *path* is a sequence of states. The latter is an assignment to the set of state, input and output variables. A full path starting from a state s_i is denoted by:

$$\pi_i \;=\; (s_i,\; s_{i+1},\; s_{i+2},\; ...)$$

All formulas in $\mathcal{L}_{mdg}$ are *path formulas*. Hence, given a property in $\mathcal{L}_{mdg}$ on an ASM under a given interpretation ψ, the property holds on the ASM if and only if the property is true for all paths starting from each initial state. The semantics of the AG operator will be :

$$(\pi,\; \sigma) \;\models\; Gp\; iff\; (\pi_j,\; \sigma\;) \models\; p\; for\; all\; j \geq i$$

Since $\mathcal{L}_{mdg}$ is a CTL* like language [37], we divide the properties in two classes: the first is the CTL like properties and the second is the LTL (Linear Time Temporal Logic) like properties. For the latter ones, we define a property according to the predicate we want to verify:

Each logical proposition is a function of the path, expressed here by s which can be formulated as a history function keeping trace of the states among the path, and the current state.

$\vdash_{def}$ `LMDG_G p s = ∀t. p s t`

The linear temporal operator F, is defined to be a function of p and s such that exists t, where the property holds

$\vdash_{def}$ `LMDG_F p s   = ∃t. p s t`

In addition, the conjunction, the implication, and the disjunction of predicates are defined as function of the proposition

```
⊢def  LMDG_IMP p1 p2 s t = ¬(p1 s t) ∨  p2 s t
```

The second class of properties is the CTL like ones. Here, we define the property according to the predicate we want to define as well as its circuit.

```
⊢def LMDG_AG R p = ∀s. ((R s) ∧ (∀t. (p s t)))
```

Our objective is to precise that each path, considered in the property, belongs to the circuit description we have. The composition of the different kind of templates is done manually by the user. Therefore, the embedding we have is more expressive than the original $\mathcal{L}_{mdg}$.

When verifying liveness properties, one is usually interested only on the so-called fair infinite computation paths. A *fair* computation path is a path along which the states satisfy the fairness condition infinitely often. In MDG, if we consider H as a fairness constraint, the formula representing the exception condition H is called *H_formula*. Its syntax is defined by the equality of two ASM_variables or an ASM_variable and a constant, the conjunction, disjunction, implication of two H_formulas, the negation or the nesting of H_formulas. However, only concrete ASM_variables may appear in the H_formula. All fairness constraints imposed are stored in a file, which is interpreted before the model checking procedure is invoked.

In HOL we represent fairness constraints by a predicate mentioning that the condition should holds in each state. When fairness conditions are imposed, we add it as a conjunction to the property as we will present in the next section.

$\vdash_{def}$ LMDG_FAIR p s = ∀t. p s t

Chapter 4

MDG-HOL Linking (The Hybrid Tool)

In this chapter, we will focus on the implementation part of the link between HOL and the MDG model checker. The implemented tool inputs are the model description, the property and the HOL goal. It generates automatically all the required MDG files, which are then communicated to the MDG tool where the verification is done. The obtained result is transmitted to HOL. Therefore, either a HOL theorem is set or hand is given to the user to do the proof interactively.

4.1 The Hybrid Tool Behavior

4.1.1 Overview

The tool developed is an interface between the HOL theorem prover and the MDG model checker. During the verification procedure, the user deals mainly with HOL. As shown in Figure 4.1, the user starts by giving the HOL design (specification or implementation), the HOL property and the goal to be proven. If this goal fits the required pattern (our tool accepts only *implication* goals), the respective MDG files are generated. The latter are sent to the MDG tool for model checking.

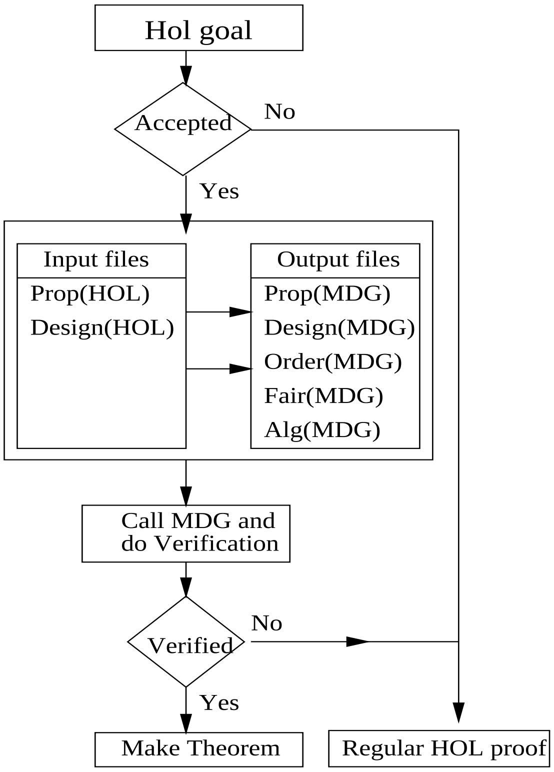


Figure 4.1: Verification Procedure with the Hybrid Tool

If the property holds, a HOL theorem is created. However, if the verification within the MDG tool fails, we have to perform the proof interactively using HOL. The tool does not accept any arbitrary HOL specification. It accepts only MDG-styles specifications and properties. We use the embedded HOL theories to express both the model and the properties descriptions. In the next sections, we detail the feature of the input files to the generated files by our tool.

4.1.2 Use of Hierarchy

Usually, hardware systems under verification are described (in HOL) in a hierarchical fashion. The main modules of the specification are divided into submodules. The submodules are repeatedly subdivided until eventually the logic gate level is reached. This is achieved by defining the structure "block" in a recursive manner.

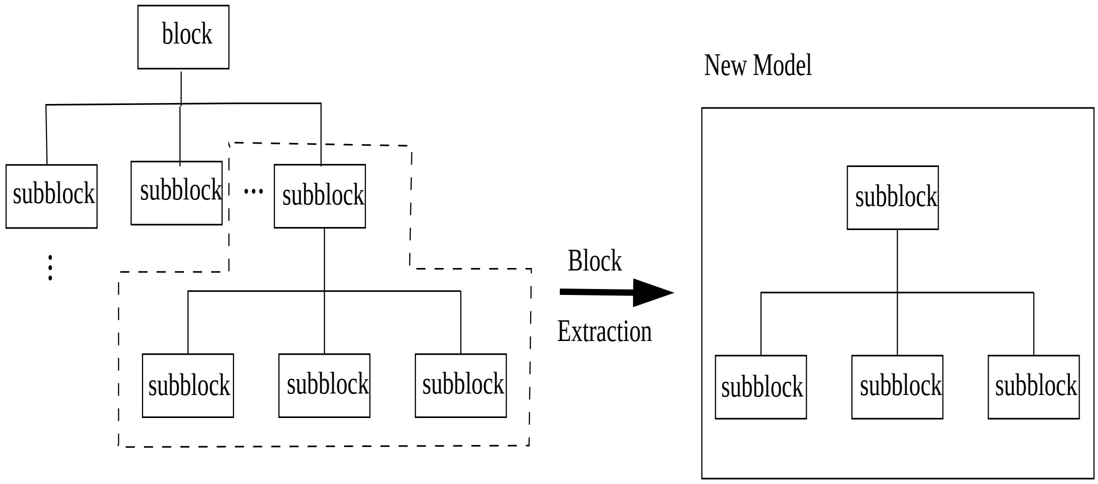


Figure 4.2: Block Extraction

The advantage of having such hierarchy is the ability to extract the block about which we want to check a property (see Figure 4.2). Hence, the model checker deals with the specification of the considered block only, not the whole design. As a result, we save on model size without constraining the user to write another specification for the appropriate block.

4.1.3 The Input Files

Design Specification File (HOL)

Design models are provided as a normal file of HOL definitions. They are written in a hierarchical structure. Since the model definition must be analyzed by the tool and ultimately converted into MDG, it should follow a specific form : it consists of a conjunction of tables, which input and output arguments must be explicitly typed and declared as MDG terms. This implies that all sorts (abstract and concrete), variables, constants and functions must be specified. Structural models are written in a subset of the HOL logic similar to that for behavioral specifications. However, they are not limited to tables but can include any component of the MDG component library.

Property Specification File (HOL)

Properties are provided as normal HOL definitions. They are written according to the $\mathcal{L}_{mdg}$ theory we embedded in HOL. The fairness constraints are added as a conjunction to the main property formula. The hybrid tool will extract the fairness constraints and put them in a file before proceeding with the adequate treatment.

Proof Goal Specification (HOL)

There are different ways to specify a goal in HOL. However, when using our tool, the goal should be an implication according to this form :

$$\vdash Design \supset Property$$

looking to proof that the design verifies the property. Since the verification is done in MDG, we need to formalise the (MDG) result in HOL. Therefore, we convert the MDG results into a form that can be used [35]:

$$\vdash FormalisedMDGresult \supset Model \supset Property$$

The general conversion theorem into HOL has been proved [35]. The result given by MDG tool can be interpreted and a HOL theorem can be instantiated for any design and any property under consideration.

4.1.4 The Generated Files

Design Specification File (MDG-HDL)

It contains:

- The signals appearing in the design model and their sorts assignments.
- The output partition specifying the design output signals.
- A network of tables and/or MDG-HDL components.

- We also give the set of initial states and transition/output relation partition strategy.

Order File (MDG-HDL)

The order file contains the order of the variables with which the multiway decision graph is built. In our case, the order is generated statically. However, some restrictions are imposed for abstract variables and functions name.

Algebraic File (MDG-HDL)

In the algebraic file, all concrete sorts used in the design specification are listed. It also includes the declaration of all used functions (concrete, cross-function and abstract). In addition, any generic constants (of abstract type) defined in the design model should be mentioned here.

Property File ($\mathcal{L}_{mdg}$)

It has the form of a property acceptable by MDG. It follows the syntax described in Section 3.3.

The MDG Fairness Files ($\mathcal{L}_{mdg}$)

In the HOL given property, the fairness constraints are part of the property, the hybrid tool takes care of separating them from the property core before processing them for the adequate treatment. This will be explained in the next section.

4.2 The Hybrid Tool Structure

Our hybrid tool is written in SML. It is composed of five main modules: the *Hybrid Tool Interface*, the *Property Module*, the *Description File Module*, the *HOL Goal Parser Module* and the *MDG Interaction Module* (cf. Figure 4.3). The user's interface to the hybrid tool is a Java GUI, responsible for getting the HOL goal, the property file and the model description file, passing them to HOL, loading the $\mathcal{L}_{mdg}$ and MDG-HDL theories and at the end of the verification process, communicating the result to the user [14]. In the second module, the *Property Parser* generates as output a data structure from which the *MDG File Generator* produces the MDG property file, and the *Property Type Generator* provides the property type. On the other side, in the *Description File Module*, the specification is first flattened.

When parsing the goal, we get the name of the property and the block we want to check. The latter can be either the main module in the specification or one of its submodules. Since the specification is written in a hierarchical way, it is possible to extract the target module, and its submodules, and to discard the others. The *Block Extraction Module* achieves this task. In the next step, the corresponding MDG files are generated (Algebraic, Order and Specification/Implementation). In order to proceed with the model checking operation, these files should be used for generating ASMs before interacting with MDG. Since the communication between the linked tools is done automatically, we implemented a special module to take care of the ASM generation task : *ASM Generation Interface*

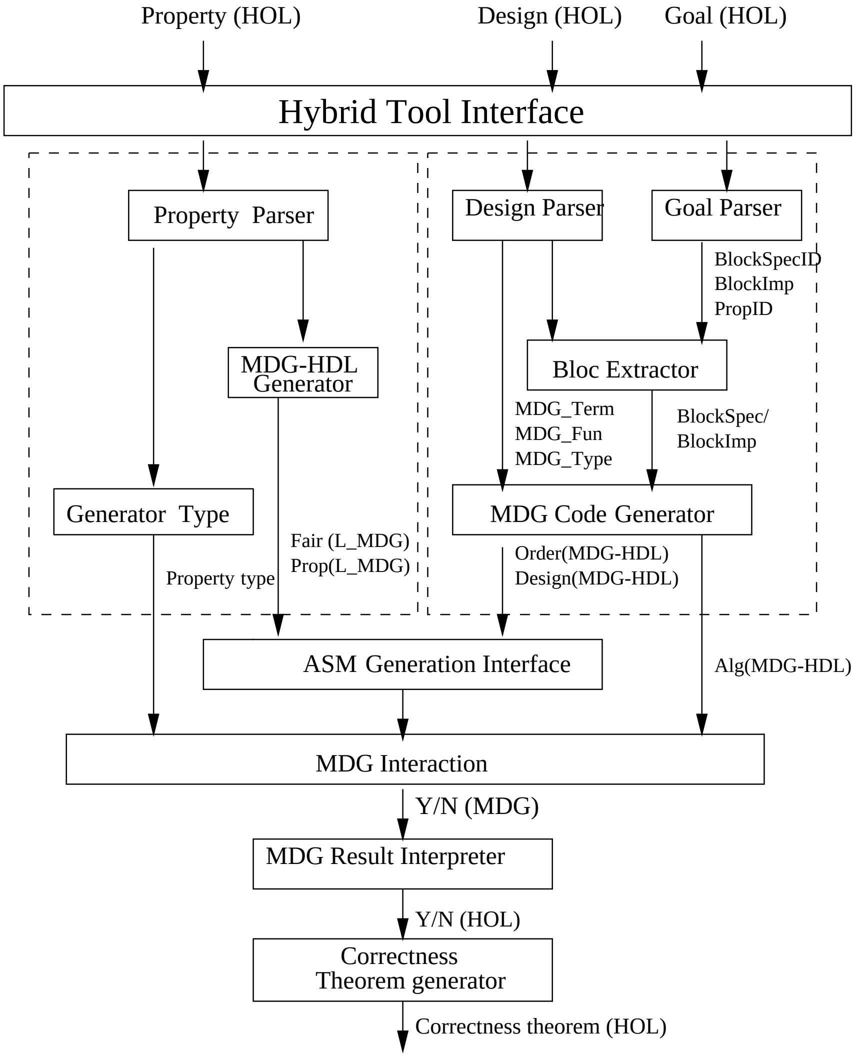


Figure 4.3: Hybrid Tool Structure

ASM Generation Interface Before interacting with MDG, two steps need to be executed. The first one is to automatically build additional ASMs that represent the $\mathcal{L}_{mdg}$ property (cf. Figure 4.4).

The *Next Manager* is an implemented module in the MDG tool that takes care of achieving this treatment. The *New_prop* represents the new property file generated where ASMs representing the property are added to the specification. The second

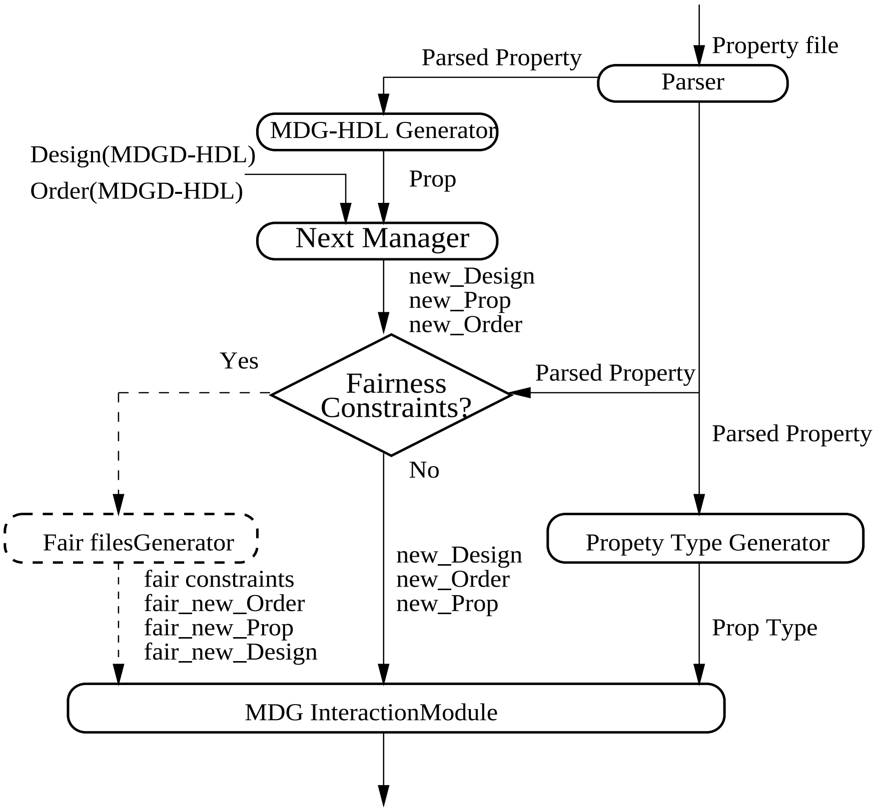


Figure 4.4: Property Module Structure

step is to test if fairness constraints are imposed. We check this on the parsed property. If so extra ASMs are constructed and connected to the original ones, *fair_new_Model*.

The *MDG Interaction Module* ensures the communication with MDG. it takes all the generated MDG files, the property type and the fairness number. The latter is provided by the property parser module. All these files are supplied to the MDG tool which applies the verification process and passes the result to HOL through the*MDG Result Interpreter Module*. If the property holds, a theorem is generated in HOL.

4.3 Application: The Timing Block

The timing block is one of the blocks composing the Fairisle ATM (Asynchronous Transfer Mode) switch fabric [22]. The Fairisle switch fabric is a real switch fabric designed and used at the University of Cambridge for multimedia applications. Curzon [7] formally verified this ATM switching element hierarchically using the HOL system. Kort *et al.* [20] presented the verification of the ATM switching using the HOL-MDG Hybrid tool where the submodules were verified using the MDG tool. Also, Pisini *et al.* [28] presented the equivalence checking of the timing block module using the hybrid tool . We present the formalisation of the state transition diagram of the timing block in terms of MDG tables according to our new theory, as well as the experimental result of some properties checked within our tool.

4.3.1 Timing Block Structure

The timing block controls the timing of the arbitration decision based on the frame start signal and the time the routing bytes arrive. The implementation of the timing block is shows in Figure 4.5. Its HOL specification is given as follow:

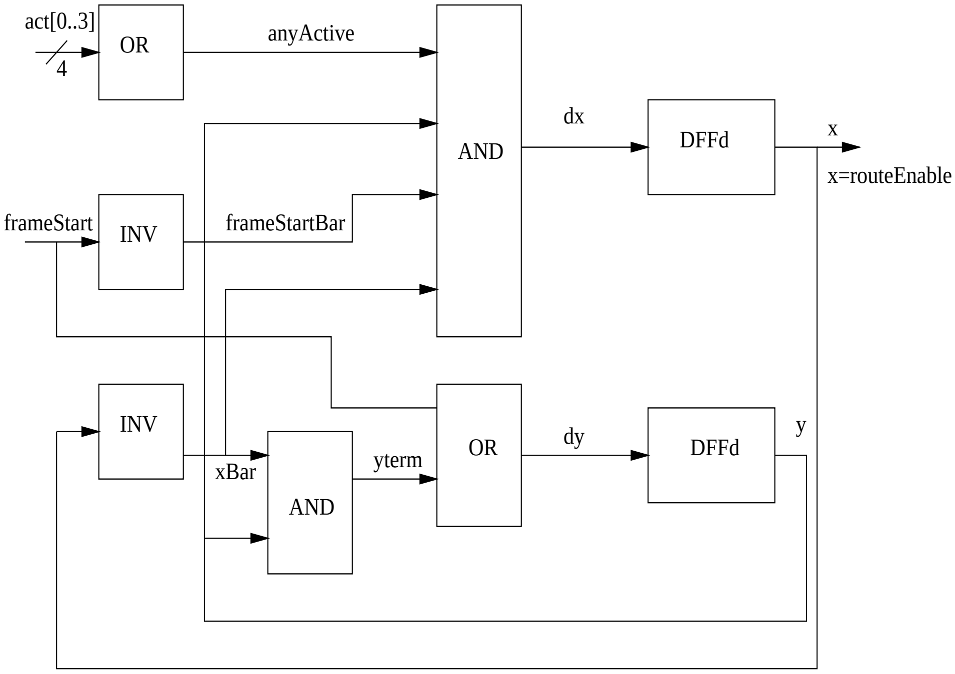


Figure 4.5: Timing Block Implementation

$\vdash_{def}$ ∀act0 act1 act2 act3 fs routeEnable_i.

TIMING_IMP (act0,act1,act2,act3,fs) routeEnable =

∃anyActive_i fsBar qxBar yterm dx dy qx qy.

mdg_or4 (act0,act1,act2,act3) anyActive

∧ mdg_not fs fsBar

∧ mdg_not qx qxBar

∧ mdg_and (qy,qxBar) yterm

∧ mdg_and4 (anyActive_i,qy,fsBar,qxBar) dx

∧ mdg_or (fs,yterm) dy

∧ mdg_reg dx qx

∧ mdg_reg dy qy

∧ mdg_fork qx routeEnable

4.3.2 Timing Block Behavior

Figure 4.6 shows the finite state machine of the behavior of the timing block, which consists of three symbolic states (*Run, Wait, Route*), and has two inputs (*frameStart* and *anyActive*) and one output (*routeEnable*).

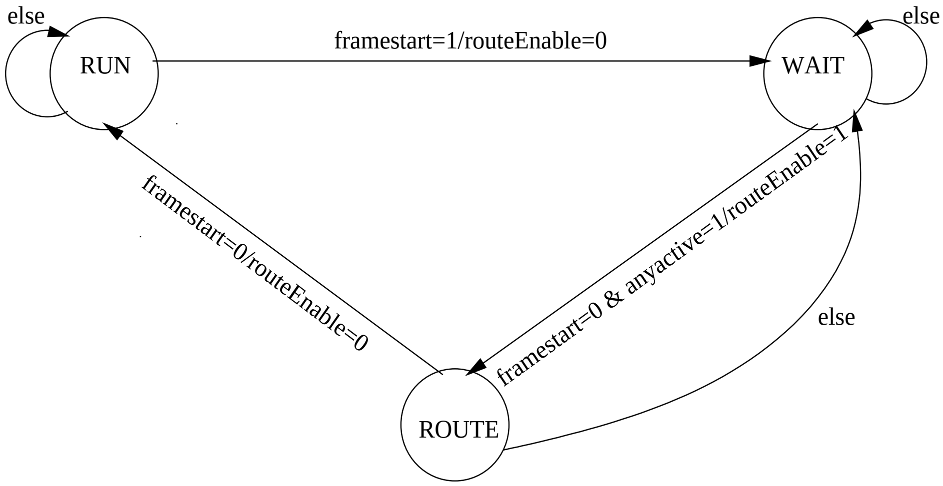


Figure 4.6: Timing Block State Machine

While the input and the output are all of Boolean sort, the state and next state variables are of concrete sort with the enumeration : *Run, Wait*, and *Route.* We hence create a concrete MDG type :

$\vdash_{def}$ `state_Type = CONCRETE   "state_Type" "run";"wait"; "route"]`

The concrete constants generated from this types are:

$\vdash_{def}$ `run  = CONC_Const "state_Type"`

$\vdash_{def}$ `cwait  = CONC_Const "state_Type"`

$\vdash_{def}$ `route  = CONC_Const "state_Type"`

Since the inputs of MDG table should be of the same sort, we use the same function *bool_to_MDGTerm* to homogenise the input types of the tables. The former converts the Boolean type t to the *MDG_term* type we defined. The HOL defined table for the transition relation of the timing block is defined in HOL as follows :

```
⊢def transition ((anyActive:num->bool),(fs:num->bool),(state:num->string
      MDG_term )) ( n_state:num->string MDG_term) =
TABLE bool_to_MDGTerm o anyActive; bool_to_MDGTerm o fs; state]
(n_state o SUC)
 [  [ DONT_CARE; TABLE_VAL(CONC_Const"F"); TABLE_VAL(run)];
 [ TABLE_VAL(CONC_Const"T"); TABLE_VAL(CONC_Const"F"); TABLE_VAL(wait)];
         [ DONT_CARE; TABLE_VAL(CONC_Const"F"); TABLE_VAL(route)]]
 [ runSIG; routeSIG; runSIG] waitSIG
```

The outputs are defined as signals: function of time t as mentioned before for the true signal and the false signal.

The MDG generated Table is:

```
⊢def component(tab_s1,table([ [ anyActive,fs,state,n_state],
                        [ *,0,run,run],
                        [ 1,0,wait,route],
                         [ *,0,route,run]wait])).
```

To specify the Timing Block behavior, we defined three tables: *active*, *transition* and *output*. The HOL specification is represented by the conjunction of those tables:

```
⊢def Timing (act0 ,act1, act2, act3, fs) routeEnable =
     ∃ anyActive state n_state.
     (active (act0,act1,act2,act3) anyActive) ∧
     (transition (anyActive,fs,state) n_state)∧
     (output state routeEnable)
```

4.3.3 Timing Block Verification

The model checking is done within the MDG tool. We provide the tool with the MDG-HDL like specification and the $\mathcal{L}_{mdg}$ properties. By calling the HOL tactic for model checking, the MDG files are generated, and the property treatment is processed. Finally, the model checking is run in MDG and the result is returned back to HOL. The following properties were verified.

- The first property is a liveness one showing that the system will be in the state *Run* in some of the computation paths. The initial state is *Wait*. The HOL property is defined as :

  $\vdash_{def}$ `Timing_property1 (state) =`

  `LMDG_AF (CONVERT timing state)  (LMDG_X((`λ` state t. state t = run)))`

  The first part of the property makes reference to the circuit we want to check. *CONVERT* is a function of the circuit and the state which express the fact that each path considered during the verification process belongs to the computatoin tree of the considered model. The MDG property derived is :

  ```
  AF(X (state = run))
  ```

- The second property is a safety one. We checked that if the system is on state run and the *FrameStart* signal is set then in the next state, the state will be Wait and the output of the timing is set to 0:

```
A((state = crun & fs= 1) -> (X(state = cwait & routeEnable = 0))
```

- In the third property, we verify fairness constraint. In HOL, the constraint is added as a conjunction to the property:

```
Timing_property1_fair (state)  =
        (timing_property1)
     /\ (LMDG_FAIR (\ fs (t:num) . ~(fs t  = 1)))
```

  However, in MDG, fairness are expressed separately in a different file, e.g.,

```
 ! (fs = 1)
```

- In the fourth property, we did the model checking on the implementation of the time block circuit.

```
A((anyActive_i_1= 0) -> (X(routeEnable_o = 1)) )
```

  The property mentions that if the signal *anyActive_i_1* is equal to 0, then in the next state the output will be set to 1. MDG checked the property and returns that the model does not verify this property (when the anyActive_i_1 is set to 0, the output in the next state is always equal to 0). So, the HOL goal is not proved. Unfortunately, the model checker does not provide a counter-example.

The model checking of these properties succeeded, and we summarise the results given by MDG in the table bellow:

Property	**CPU_s**	**$Memory_{Byte}$**	**Nodes**	**Components**	**Signals**
Property1	0.15	66908	123	9	17
Property2	0.18	72212	145	11	20
Property3	0.19	68812	116	10	17
Property4	0.20	98644	226	16	23

Table 4.1: **Model Checking Results of the Timing Block**

To be more accurate, the user should allow for an extra two to three minutes, required time for the tool to load the HOL theories and the input files, generate the MDG ones and finally interact with MDG system. The model checking within the tool is definitely faster than proving directly with HOL. Yet, the direct proof should be feasible in the theorem prover. For the failed property, the HOL goal is not proved. However, it is still possible to set the negation of the theorem. While the Timing BLock is a small illustrative example, in the next Chapter, we will present a significantly larger case study.

Chapter 5

Case Study: Island Tunnel Controller

In this chapter, we illustrate our methodology using the Island Tunnel Controller (ITC) [40] as a case study. It is ideal for illustrating purposes since its specification contains abstract sorts and uninterpreted functions. Some properties are verified using our hybrid tool HOL-MDG.

5.1 Island Tunnel Controller Description

The Island Tunnel Controller is depicted in Figure 5.1. It controls the traffic lights at both ends of a tunnel connecting the mainland and the island. Four sensors are installed at both ends of the tunnel to detect the vehicles presence: one at the tunnel entrance *(ie)* and one at the tunnel exit *(ix)* in the island side, and one at the tunnel entrance *(me)* and one at the tunnel exit *(mx)* on the mainland side. It is assumed that all cars are finite in length, that no cars gets stuck in the tunnel, that no cars do not exit the tunnel before entering the tunnel, that cars do not leave the tunnel entrance without travelling through the tunnel, and that there is sufficient distance between two cars such that the sensors can distinguish the cars. The

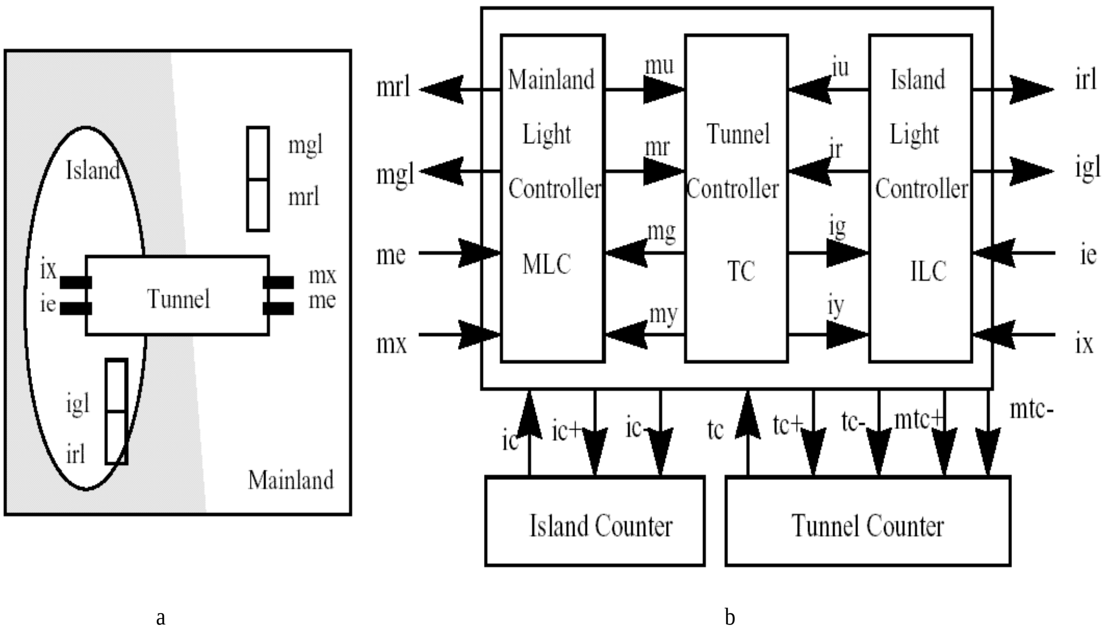


Figure 5.1: Island Tunnel Controller Structure

island tunnel controller is composed of five modules: The *Island Light Controller*, the *Tunnel Controller*, the *Mainland Light Controller*, the *Island Counter* and the *Tunnel Counter* (refer to [37] for the state transition diagrams of each component). The Island light Controller (ILC) has four states: *green*, *entering*, *red* and *exiting*. The outputs *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 occupying the tunnel, and *ir* indicates that ILC is requesting the tunnel. The input *iy* requests the ILC to release control of the tunnel, and *ig* grants control of the tunnel from the island side (cf. Figure 5.2). A similar set of signals is defined for the Mainland Light Controller

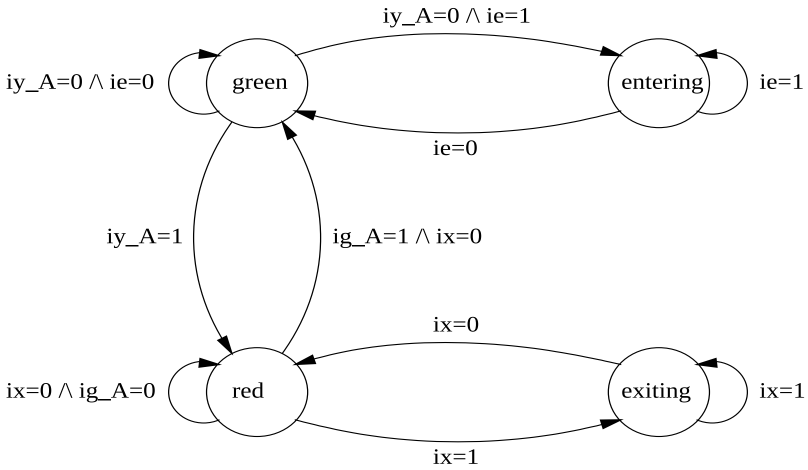


Figure 5.2: The Island Controller

(MLC). However, in this module, the behaviour depends on a cross-function *lessn* which takes as input an abstract sort and generates as output a Boolean type (cf. Figure 5.3).

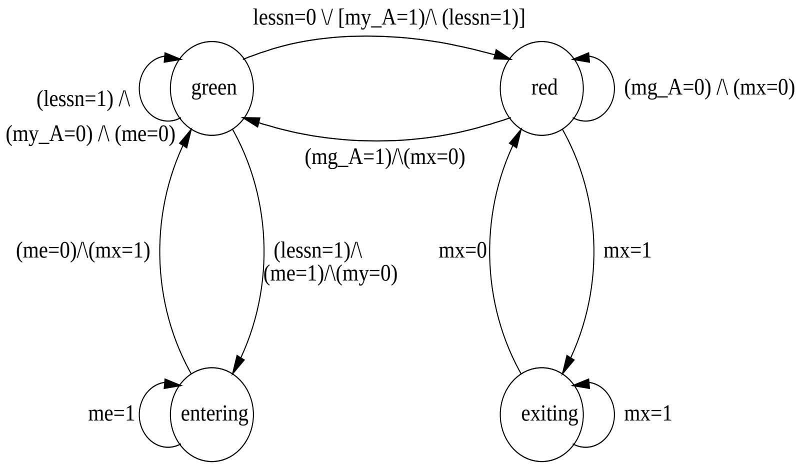


Figure 5.3: The Mainland Controller

The Tunnel Controller (TC) processes the requests for access issued by the ILC and MLC. The Island Counter and the Tunnel Counter keep track of the car's number currently on the island and in the tunnel, respectively. For the tunnel controller, the counter *tc* is increased by 1 depending on *tc+* or decremented by 1 depending on *tc-* unless it is already 0. The *Island Counter* operates in a similar way, except that the incrementation and to decrement depend on *ic+* and *ic-*, respectively.

5.2 Specification and Properties Definitions

For the two modules ILC and MLC, we need to define an enumerated type : *state_type* which takes one of the different possible sates of the system. Here, the suffix *c_* is added to the names.

```
state_Type = CONCRETE  "state_Type" ["c_green";"c_red"; "c_entering";"c_exiting"]

c_green  = CONC_Const "state_Type"
```

In our specification, all the types are concrete except one abstract sort of type *wordn* used to describe the tunnel controller and the tunnel counter. The two uninterpreted function *inc* and *dec* are specified as well. We also define two cross-functions *lessn* and *equz*. The design specification is written hierarchically, where blocks are represented by the conjunction of their respective tables. The whole system is the conjunction of the five blocks mentioned above.

Next, we have specified and verified a number of properties on the Island Tunnel Controller. In the following, we describe four samples for illustration purposes, where the symbols “`&`”, “`!`” and “`->`” mean logical and, negation, and implication respectively (using $\mathcal{L}_{mdg}$ syntax). Experimental results of a larger set of properties are displayed in Table 5.1.

- **Property1:** Let *is* be the state variable of the ILC, and *igl* its the green light variable. The green light of the ILC module must be off if there is a car exiting the tunnel.

```
LMDG_AG ((CONVERT ILC_comp is)

      (LMDG_IMP

      (LMDG_AND ( is (t:num).(ix = true)) ( is (t:num). (is t  = c_red)))

      ( is (t:num).(igl t = true))))
```

  The derived MDG property is:

```
AG ((is = c_red & ix = 1) -> (igl = 0))
```

- **Property2:** Let *mgl* be the green light variable of the MLC and *ms* its state variable. The green light of the MLC module should be on in a future state during the computation. This property verifies the liveness of the component.

```
LMDG_AG (MLC_comp mgl) ( ms (t:num). (mgl  t= true))
```

  The derived MDG property is:

```
AF (mgl = 1)
```

- **Property3:** Let *mtc_min* be the signal to decrement the counter of the MLC module, and *mgl* its green light variable. These two signals can never be set to 1 at the same moment, because when exiting the tunnel, the counter is decremented; hence, the red light must be on.

```
 LMDG_AG(( CONVERT (MLC_comp mgl))
      (LMDG_NOT( LMDG_AND (  ms (t:num). mtc_min =true)
                          (  ms (t:num). mgl = true))))
```

The derived MDG property is:

```
AG ( ! (( mtc_min = 1) & (mgl = 1)))
```

- **Property4:** Let *my* and *iy* be the signals to increment the mainland, and the island counters, respectively. The TC should never increment both counters.

```
LMDG_AG ( CONVERT  ILC_comp(igl)  MLC_comp (mgl))
    (LMDG_NOT ( LMDG_AND
      ((  is (t:num). ( igl = true)))
       (  ms (t:num). ( mgl = true))))
```

The derived MDG property is:

```
AG( !((my = 1) & (iy = 1 )))
```

5.3 Experimental Results

We set up HOL goals, which denotes that the properties we proposed imply the design specification. Table 5.1 describes the model checking results of these and other properties, including CPU time, memory usage and number of MDG nodes generated. We also report the number of components and signals of the reduced (extracted) design model effectively used for model checking in MDG.

Property	**CPU$_s$**	**Memory$_{Byte}$**	**MDG Nodes**	**Components**	**Signals**
Property1	0.32	0.66	318	18	32
Property2	0.36	0.77	313	13	31
Property3	0.41	0.73	401	16	34
Property4	1.12	1.91	1266	13	29
Property5	0.91	1.26	1027	10	26
Property6	0.93	1.77	1166	13	29
Property7	1.15	1.39	11002	16	33
Property8	1.15	1.39	11002	16	33

Table 5.1: **Model Checking Results with Block Extraction**

It is clear here that the verification is much faster than doing the proof interactively with HOL. Moreover, since the user has just to give the tool the HOL goal, specification and property, the HOL-MDG communication and verification are done automatically. All properties were successfully verified. After interpreting the MDG result, respective HOL theorems were created.

In addition, since our specification is written in a hierarchical way, during each property verification, our hybrid tool extracts the module to be proved. To see the advantage of such block extraction, we checked the first two properties on the whole

ITC design (See Table 5.2).

Property	**CPU$_s$**	**Memory$_{Byte}$**	**MDG Nodes**	**Components**	**Signals**
Property1	0.74	1384668	830	26	62
Property2	0.87	1467908	1027	24	60

Table 5.2: **Model Checking Results without Block Extraction**

The obtained results demonstrate that the block extraction performs savings in memory usage and CPU run time by more than 50 %.

In summary, our hybrid tool has a better verification performance than the MDG model checker and the HOL theorem prover individually. However, the enhancement achieved with the block extraction in the MDG side is not guarantied when a global property on the whole design is verified.

Chapter 6

Conclusion and Future Work

Formal verification approaches are more and more used in the verification process of digital systems. Despite of their efficiency, existing tools still have some drawbacks. Theorem proving, for instance, suffers from interactivity and the need of continuous guidance, while model checking, though full automatic, suffers from state space explosion. To improve the verification process, many hybrid approaches were proposed as a way to take advantage of different formal verification techniques.

In this thesis, we proposed a tool for linking HOL theorem proving and MDG model checking. HOL, a theorem prover based on high-order logic, allows the use of hierarchical verification and abstraction. MDG, a decision diagram based tool, provides features for model checking and the use of a relatively limited abstraction. The hybrid tool we proposed generates the required MDG files and communicates them to the MDG tool, where the property is checked. Thereafter, the verification result is returned back to HOL. The verification of the properties within the MDG tool introduces automation to HOL since the proof is not conducted interactively. Moreover, our tool allows a block extraction feature, which increases the performance of the model checker by reducing the memory usage and the CPU time.

An interface between the two tools is implemented using MoscowML. The input languages of the model checker were first embedded into HOL allowing the definition of MDG like specification and properties. The user communicates mainly with the theorem prover. Second, the tool described in [20] has been extended to handle abstract datatypes. Our purpose from this expansion is to have the possibility to do equivalence checking of complex circuits which might cause problems in the model checker.

The embedding of the $\mathcal{L}_{mdg}$ language provides a way to write properties in HOL. The embedding of the MDG-HDL grammar provides the tool with abstract types needed, benefiting from the abstraction allowed in HOL as well as the abstract sorts, the uninterpreted functions, featured by the MDGs.

We presented the Timing Block of an *ATM switch fabric* as an example, then the *Island Tunnel Controller* as a case study illustrating the verification within the built tool. The obtained results shows that the verification is more tractable than using each tool individually.

Our hybrid tool can help in resolving the problem of non-termination in MDG. This problem has been discussed in [5, 38]. In [1], Ait *et al.* proposed an approach for solving the non-termination problem by using the schematization method, namely -terms. As, we are using a hierarchical approach, when considering a model, some modules are verified in the MDG tool, and for the other modules that will generate an infinite set of states, HOL theorem prover will be used. Here, for each module, rather than the specification of its behavioral, we should specify the environment of the module. The abstract counter we presented is an example illustrating the feasibility of this approach. Since it is specified with abstract terms in table inputs, its verification with MDG leads to the non-termination problem, while we prove in HOL its equivalence. Future examples can be tackled to ensure the usability of the tool for such problems.

Another future issue of our project is the integration of reachability analysis in HOL. In fact, we are defining a property in HOL according to the model we want to check. HOL should be able to compute the different design paths and verify the property according to the input MDG tables we have as input language for the specification. Actually, if the MDG tool does not give a result for a property because of state explosion, the user will have hard time to prove such module within HOL. Since, there is no way to compute paths from the table structures.

Developing HOL tactics to treat the conjunction and the disjunction of properties can be a good extension of the current tool. This means, rather than sending only one property to be verified, the user would have the possibility to specify many properties in HOL. Then, the tactic will split the property resulting from conjunction (or disjunction) to separated properties and treat each property independently. The verification of each property will be done within MDG. The obtained results(theorems) are later on conjuncted (or the disjuncted). Finally the result is returned back to HOL.

Appendix A

$\mathcal{L}_{mdg}$ HOL Theory

As explained in Chapter 3, we proposed an embedding in HOL of the MDG property input language : $\mathcal{L}_{mdg}$. It represents a subset of the Abstract_CTL* [37].

Since $\mathcal{L}_{mdg}$ is a ACTL* [37] like language, we divide the properties in two classes: the first is the CTL* like properties and the second is the LTL like properties.

A.1 CTL* like Properties

```
⊢def  val LMDG_AG = Define ‘

LMDG_AG R p = ∀s. ((R s) ∧ (∀t. (p s t)))‘;

⊢def  val LMDG_AF = Define ‘

LMDG_AF R p = ∀s. ((R s) ∧ (∃t. (p s t)))‘;

⊢def  val LMDG_A = Define ‘

LMDG_A R p t= ∀s. ((R s) ∧ (p s t))‘;
```

Only the univesal path quantifier is allowed in MDG, however, we embedded in HOL the existantial path quantifier too.

$\vdash_{def}$ `val LMDG_EG = Define ‘`

`LMDG_EG R p =` $\exists$`s. ((R s)` $\wedge$ `(`$\forall$`t. (p s t)))‘;`

$\vdash_{def}$ `val LMDG_EF = Define ‘`

`LMDG_EF R p =` $\exists$`s. ((R s)` $\wedge$ `(`$\exists$`t. (p s t)))‘;`

A.2 LTL like Properties

$\vdash_{def}$ `val LMDG_G = Define ‘`

`LMDG_G p s =` $\forall$`t. p s t ‘;`

$\vdash_{def}$ `val LMDG_F = Define ‘`

`LMDG_F p s t =` $\exists$`t1. p s t1 ‘;`

$\vdash_{def}$ `val LMDG_U = Define ‘`

`LMDG_U p1 p2 s =` $\exists$`t. (p2 s t` $\wedge$ `(`$\forall$`t1. t1 < t` $\Longrightarrow$ `p1 s t1))‘;`

$\vdash_{def}$ `val LMDG_X = Define ‘`

`LMDG_X p s t = p s (t+1)‘;`

$\vdash_{def}$ `val LMDG_NOT = Define ‘`

`LMDG_NOT p s t = ¬ p s t ‘;`

$\vdash_{def}$ `val LMDG_IMP = Define ‘`

`LMDG_IMP p1 p2 s t = ¬(p1 s t) ∨ p2 s t ‘;`

$\vdash_{def}$ `val LMDG_AND = Define ‘`

`LMDG_AND p1 p2 s t = (p1 s t) ∧ p2 s t ‘;`

$\vdash_{def}$ `val LMDG_OR = Define ‘`

`LMDG_OR p1 p2 s t = p1 s t ∨ p2 s t ‘;`

$\vdash_{def}$ `val LMDG_FAIR = Define ‘`

`LMDG_FAIR p = ∀s t. (p s t) ‘;`

$\vdash_{def}$ `val LMDG_VAL = Define ‘`

`LMDG_VAL v s t = v ‘;`

$\vdash_{def}$ `val LMDG_VAR = Define ‘`

`LMDG_VAR x s t = s x t ‘;`

$\vdash_{def}$ `val LMDG_IS = Define ‘`

`LMDG_IS q1 q2 s t = (q1 s t = q2 s t) ‘;`

$\vdash_{def}$ `val LMDG_TRUE = Define ‘`

`LMDG_TRUE s t = T ‘;`

$\vdash_{def}$ `val LMDG_FALSE = Define ‘`

`LMDG_FALSE s t = F ‘;`

$\vdash_{def}$ `val LMDG_START = Define ‘`

`LMDG_START p s = ∀s. p s 0 ‘;`

Appendix B

MDG-HDL HOL Theory

We present bellow a subst of the embedded MDG-HDL grammar and the table structure in HOL. **MDG sorts**

$\vdash_{def}$ `Hol_datatype ‘MDG_sort = ABSTRACT of 'a → string`

`CONCRETE of string → string list‘ ;`

Predicates to determine the type of a given sort

$\vdash_{def}$ `Define ‘( IsConcreteSort (ABSTRACT Abs nme) = F )∧`

`( IsConcreteSort (CONCRETE Conc z) = T)‘ ;`

$\vdash_{def}$ `Define ‘( IsAbstractSort (ABSTRACT Abs nme) = T )∧`

`( IsAbstractSort (CONCRETE Conc z)= F)‘ ;`

$\vdash_{def}$ `Define ‘bool_type = CONCRETE "bool_type"  [ "true";"false"]‘;`

MDG Variables

```
⊢def  Hol_datatype ‘MDG_VAR = MDG_VAR of string → MDG_sort‘;
```

MDG functions

```
⊢def  Hol_datatype ‘MDG_Fun = MDG_FUN of string → MDG_VAR list → MDG_VAR ‘;
```

MDG Terms

```
⊢def MDG_term = GE_Const of 'a
                  | CONC_Const of string
                  | VAR_Term of  MDG_VAR
                  | FN_Term of  MDG_Fun
                  | TERM of MDG_term => MDG_term
⊢def  Define ‘( IsGenericConstant (GE_Const Abs) = T )
              ∧ (IsGenericConstant (CONC_Const z)= F)
              ∧ (IsGenericConstant (VART var ) = F)
              ∧ (IsGenericConstant (FN fun ) = F)
              ∧ (IsGenericConstant (TERM term1 term ) = F)‘;
⊢def  Define ‘(IsConcreteConstant (GE_Const Abs) = T )
              ∧ (IsConcreteConstant (CONC_Const z)= F)
              ∧ (IsConcreteConstant (VART var ) = F)
              ∧ (IsConcreteConstant (FN fun ) = F)
              ∧ (IsConcreteConstant (TERM term1 term ) = F)‘;
```

```
⊢def  Define ‘( IsFunction (GE_Const Abs) = F )
              ∧ (IsFunction (CONC_Const z)= F)
              ∧ (IsFunction (VART var ) = F)
              ∧ (IsFunction (FN fun ) = T)
              ∧ (IsFunction (TERM term1 term ) = F)‘;
⊢def  Define ‘( IsVariable (GE_Const Abs) = F )
              ∧ (IsVariable (CONC_Const z)= F)
              ∧ (IsVariable (VART var ) = T)
              ∧ (IsVariable (FN fun ) = F)
              ∧ (IsVariable (TERM term1 term ) = F)‘;
```

MDG Table Structure

```
⊢def  val TABLE_VAL_AX =
   Hol_datatype ‘Table_Val = TABLE_VAL of  'a MDG_term  DONT_CARE‘;
⊢def  val TableVal_to_Val  =   Define
  ‘(TableVal_to_Val (TABLE_VAL (v:'a MDG_term)) = v)‘ ;
⊢def   val Table_match = Define ‘ (Table_match inputs [  ] (t:num) = T)
         ∧ (Table_match inputs (CONS v vs) t =
            (( (HD(inputs) t) = TableVal_to_Val (v:'a Table_Val) )
         ∨ (v = DONT_CARE)) ∧ (Table_match (TL inputs) vs t) )‘;
⊢def (table inps (out:num −> 'b) ( [ ]:('a Table_Val list) list)
             V_out default t =  (out t = default t))
         ∧ (table inps  out (CONS v vs) V_out default t =
            ((Table_match inps v t) →  (out t = (HD V_out)t)))
             |(table inps out vs (TL V_out) default t)))
⊢def  TABLE inps (out:num −> 'b) (V_outs:('a Table_Val list) list)
       V_out default = ∀t. table  inps out V_outs V_out default t
```

Definition of some MDG Tables

```
⊢def  bool_to_MDGTerm1:bool-> string MDG_term) b =
         if (b = T) then  (CONC_Const "T" )  else (CONC_Const "F"));
⊢def  FSIG =λ(t:num) . (CONC_Const "F" );
⊢def  TSIG =λ(t:num) . (CONC_Const "T" );
⊢def  NOT_TABLE (x:num->bool) (y:num->bool) =
          TABLE [  bool_to_MDGTerm1  o x] (bool_to_MDGTerm1  o y)
                 [  [ TABLE_VAL( CONC_Const "T")]]
                 [  FSIG] TSIG ;
⊢def  NOT_TABLE2 (x:num->bool) (y:num->bool) =
      TABLE  [ bool_to_MDGTerm1 o x] (bool_to_MDGTerm1 o y)
             [  [ TABLE_VAL (CONC_Const  "F")];
             [ TABLE_VAL (CONC_Const  "T")]] [  TSIG;FSIG] (ARB);
⊢def val AND_TABLE = new_definition("AND_TABLE",
   --`AND_TABLE (x1:num->bool) (x2:num->bool) (y:num->bool) =
   TABLE  [ bool_to_MDGTerm1 o x1;bool_to_MDGTerm1 o x2] (bool_to_MDGTerm1 o y)
          [ [ TABLE_VAL(CONC_Const "F");TABLE_VAL(CONC_Const "F")];
          [ TABLE_VAL(CONC_Const "F"); TABLE_VAL(CONC_Const "T")]
          [ TABLE_VAL(CONC_Const "T"); TABLE_VAL(CONC_Const "F")];
          [ TABLE_VAL(CONC_Const "T"); TABLE_VAL(CONC_Const "T")]]
          [ FSIG;FSIG;FSIG;TSIG] TSIG  `--);
```

$\vdash_{def}$ AND4_TABLE(x1:num->bool)(x2:num->bool)(x3:num->bool)(x4:num->bool)(y)=

```
     TABLE[ bool_to_MDGTerm1 o x1;bool_to_MDGTerm1 o x2;
     bool_to_MDGTerm1 o x3;bool_to_MDGTerm1 o x4](bool_to_MDGTerm1 o y)
     [  [ TABLE_VAL (CONC_Const"T"); TABLE_VAL(CONC_Const "T");
    TABLE_VAL(CONC_Const"T"); TABLE_VAL(CONC_Const"T")]] [ TSIG] FSIG
```

$\vdash_{def}$ OR_TABLE (x1:num->bool) (x2:num->bool) (y:num->bool) =

```
  TABLE [ bool_to_MDGTerm1 o x1;bool_to_MDGTerm1 o x2] (bool_to_MDGTerm1 o y)
         [  [ TABLE_VAL (CONC_Const  "F"); TABLE_VAL (CONC_Const  "F")];
         [ TABLE_VAL (CONC_Const  "F"); TABLE_VAL (CONC_Const  "T")];
  [ TABLE_VAL (CONC_Const  "T"); DONT_CARE]]
          [ FSIG;TSIG;TSIG] (ARB);
```

$\vdash_{def}$ OR4_TABLE (x1:num->bool) (x2:num->bool) (x3:num->bool)

```
         (x4:num->bool) (y:num->bool) =
    TABLE  [ bool_to_MDGTerm1 o x1;bool_to_MDGTerm1 o x2;
       bool_to_MDGTerm1 o x3;bool_to_MDGTerm1 o x4] (bool_to_MDGTerm1 o y)
           [  [ TABLE_VAL (CONC_Const "F"); TABLE_VAL (CONC_Const  "F");
          TABLE_VAL (CONC_Const "F"); TABLE_VAL (CONC_Const "F")]]
           [ FSIG] TSIG ;
```

$\vdash_{def}$ NAND_TABLE (x1:num->bool) (x2:num->bool) (y:num->bool)=

```
     TABLE [ bool_to_MDGTerm1 o x1;bool_to_MDGTerm1 o x2](bool_to_MDGTerm1 o y)
           [  [ TABLE_VAL (CONC_Const  "F"); TABLE_VAL (CONC_Const  "F")];
           [ TABLE_VAL (CONC_Const  "F"); TABLE_VAL (CONC_Const  "T")];
           [ TABLE_VAL (CONC_Const  "T"); TABLE_VAL (CONC_Const  "F")];
           [ TABLE_VAL (CONC_Const  "T");TABLE_VAL (CONC_Const  "T")]]
           [ TSIG;TSIG;TSIG;FSIG] (ARB);
```

$\vdash_{def}$ NOR_TABLE (x1:num->bool) (x2:num->bool) (y:num->bool) =

```
    TABLE [ bool_to_MDGTerm1 o x1;bool_to_MDGTerm1 o x2] (bool_to_MDGTerm1 o y)
     [  [ TABLE_VAL (CONC_Const  "F"); TABLE_VAL (CONC_Const  "F")];
        [ TABLE_VAL (CONC_Const  "F"); TABLE_VAL (CONC_Const  "T")];
        [ TABLE_VAL (CONC_Const  "T"); TABLE_VAL (CONC_Const  "F")];
        [ TABLE_VAL (CONC_Const  "T");TABLE_VAL (CONC_Const  "T")
        [ TSIG;FSIG;FSIG;FSIG] (ARB);
```

$\vdash_{def}$ XOR_TABLE (x1:num->bool) (x2:num->bool) (y:num->bool)=

```
    TABLE [ bool_to_MDGTerm1 o x1;bool_to_MDGTerm1 o x2] (bool_to_MDGTerm1 o y)
         [  [ TABLE_VAL (CONC_Const  "F"); TABLE_VAL (CONC_Const  "F")];
         [ TABLE_VAL (CONC_Const  "F"); TABLE_VAL (CONC_Const  "T") ];
         [ TABLE_VAL (CONC_Const  "T"); TABLE_VAL (CONC_Const  "F")];
         [ TABLE_VAL (CONC_Const  "T");TABLE_VAL (CONC_Const  "T")]]
         [ FSIG;TSIG;TSIG;FSIG] (ARB);
```

$\vdash_{def}$ FORK_TABLE (x:num->bool) (y:num->bool)=

```
    TABLE  [ bool_to_MDGTerm1 o x] (bool_to_MDGTerm1 o y)
           [  [ TABLE_VAL (CONC_Const  "F")];
           [ TABLE_VAL (CONC_Const  "T")]]
           [ FSIG;TSIG] FSIG;
```

Definition of MDG pre-defined components

```
⊢def  val mdg_not = new_definition("mdg_not",
     --`mdg_not x y = ∀(t:num) . y t = ¬(x t) `--);
⊢def  val mdg_and = new_definition("mdg_and",
     --`mdg_and (x1,x2) y = ∀(t:num) . y t = (x1 t) ∧ (x2 t) `--);
⊢def  val mdg_and3 = new_definition("mdg_and3",
     --`mdg_and3 (x1,x2,x3) y = ∀(t:num) . y t = (x1 t) ∧ (x2 t) ∧
(x3 t)`--);
⊢def  val mdg_and4 = new_definition("mdg_and4",
     --`mdg_and4 (x1,x2,x3,x4) y = ∀(t:num) . y t = (x1 t) ∧ (x2 t) ∧ (x3 t) ∧
(x4 t)`--);
⊢def  val mdg_or = new_definition("mdg_or",
     --`mdg_or (x1,x2) y = ∀(t:num) . y t = (x1 t) ∨ (x2 t) `--);
⊢def  val mdg_or4 = new_definition("mdg_or4",
     --`mdg_or4 (x1, x2, x3, x4) y = ∀(t:num) . y t = (x1 t) ∨ (x2 t) ∨
                                                  (x3 t) ∨ (x4 t)`--);
⊢def  val mdg_nand = new_definition("mdg_nand",
     --`mdg_nand (x1, x2) y = ∀(t:num) . y t = ¬(x1 t) ∨ ¬(x2 t) `--);
⊢def  val mdg_nor = new_definition("mdg_nor",
     --`mdg_nor (x1, x2) y = ∀(t:num). y t = ¬(x1 t) ∧ ¬(x2 t) `--);
⊢def  val mdg_nor3 = new_definition("mdg_nor3",
     --`mdg_nor3 (x1, x2, x3) y = ∀(t:num). y t = ¬(x1 t) ∧ ¬(x2 t) ∧
¬(x3 t)`--);
```

```
⊢def  val mdg_xor = new_definition("mdg_xor",

        --‘mdg_xor (x1, x2) y = ∀(t:num). y t =  ((x1 t) ∧

¬(x2 t)) ∨  ¬(x1 t)         ∧ (x2 t)) ‘--);

⊢def  val mdg_reg = new_definition("mdg_reg",

         --‘mdg_reg x y = ∀t. (y)(t+1) = x t‘--);

⊢def  val mdg_fork = new_definition("mdg_fork",

        --‘mdg_fork x y = ∀t. (x) t = y t‘--);

⊢def  val mdg_transform = new_definition ("mdg_transform",

        --‘mdg_transform x1  x2   = ∀ t .∃y.  x2(t) = y ( x1 t) ‘--);
```

Bibliography

[1] O. Ait Mohamed, X. Song, and E. Cerny. On the non-termination of MDG-Based Abstract State Enumeration. *Theoretical Computer Science Journal*, To appear.

[2] G. Birtwistle, B. Graham, and S. K. Chin. *new_theory'HOL'; An Introduction to Hardware Formal Verification in Higher Order Logic.* Laboratory for Applied Logic, Department of Computer Science,Brigham Young University, August 1994.

[3] R. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. In *International Conference on Computer-Aided Design*, pages 236–243, 1995.

[4] E. Cerny, F. Corella, M. Langevin, X. Song, S. Tahar, and Z. Zhou. *Automated Verification with Abstract State Machines Using Multiway Decision Graphs*, volume 1287. Formal Hardware Verification: Methods and Systems in Comparison. Lecture Notes in Computer Science, State-of-the-Art Survey, Springer

Verlag, 1997.

[5] F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny. Multiway decision graphs for automated hardware verification. *Formal Methods in System Design*, 10(1):7–46, 1997.

[6] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS, sri international. April 1995.

[7] P. Curzon. The Formal Verification of the Fairisle ATM Switching Element: an Overview. Technical Report 328, University of Cambridge, Computer Laboratory, March 1994.

[8] P. Curzon, S. Tahar, and O. Ait-Mohamed. Verification of the MDG Components Library in HOL. *Supplementary Proc. International Conference on Theorem Proving in Higher-Order Logics, Canberra, Australia*, September 1998.

[9] L. A. Dennis, G. Collins, M. Norrish, R. Boulton, K. Slind, G. Robinson, M. Gordon, and T. Melham. The PROSPER Toolkit. In *in Proccendings of the Sixth International Conference on Tools and Algorithms for the Construction and Analysis of Systems, LNCS 1785*, Springer Verlag, 2000.

[10] M. Gordon. Combining Deductive Theorem Proving with Symbolic State Enumeration. *21 Years of Hardware Formal Verification*, December 1998.

[11] M. Gordon. Using hol to study *sugar 2.0* semantics. *to be published as NASA Conference Proceedings CP-2002-211736*, 2002.

[12] M. Gordon. Reachability Programming in HOL98 Using BDDs. *Theorem Proving and Higher Order Logics*, LNCS 1125, Springer Verlag,2000.

[13] K. Havelund and N. Shankar. Experiments in Thoerem Proving and Model Checking for Protocol Verification. *Formal Methods Europe*, LNCS 1051:662–682, Springer Verlag, 1996.

[14] R. Hum, H. Yip, H. Li, R. Mizouni, and S. Tahar. A GUI for linking HOL to MDG. Technical report, ECE Dept., Concordia University, June 2002.

[15] J. Hurd. Integrating Gandalf and HOL. *In Theorem Proving in Higher Order*, LNCS 1690:311–321, Springer Verlag, 1999.

[16] I. Beer, S. Ben David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The temporal logic sugar. In *Computer Aided Verification*, volume LNCS 2102, pages 363–367, Springer Verlag, 2001.

[17] Synopsys Inc. *Static Timing and Formal Verification: Online Manual*. Synopsis Corporation, 2000.

[18] J. Joyce and C. Seger. Linking BDD-based Symbolic Evaluation to Interactive Theorem Proving. In *In proceedings of the 30th Design Automation Conference, Dallas, Texas, United States*, pages 469–474, June, 1993.

[19] J. Joyce and C. Seger. The HOL-Voss System: Model-Checking inside a General Purpose Theorem-Prover. *Higher Order Logic Theorem Proving and Its Applications*, LNCS 780:185–198., Springer Verlag, 1994.

[20] I. Kort, S. Tahar, and P. Curzon. Hierarchical Verification Using an MDG–HOL Hybrid Tool. *Correct Hardware Design and Verification Methods*, LNCS 2144:244–258, Springer Verlag, 2001.

[21] R. Kumar, K. Schneider, and T. Kropf. Structuring and Automating Hardware Proofs in a Higher-Order Theorem-Proving Environment. *Formal Methods in System Design*, 2(2):165–223, 1993.

[22] I. M. Leslie and D. R. McAuley. Fairisle: an atm network for the local area. In *ACM Communication Review*, volume 19(4), pages 327–336, 1991.

[23] M. Gordon, R. Milner, and C. Wadworth. Edinburgh lcf: A mechanized logic of computation. volume LNCS 78. Springer Verlag, 1979.

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

[25] T. Melham and M. Gordon. *Introduction to Higher Order Logic, Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[26] T. Melhem. *Higher Order Logic and Hardware Verification.* Cambridge University Press, 1993.

[27] L. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1996.

[28] V. Pisini, S. Tahar, O. Ait-Mohamed, P. Curzon, and X. Song. Formal Hardware Verification by Integrating HOL and MDG, March 2000, ACM Publications.

[29] S. Rajan, N. Shankar, and M. Srivas. An Integration of Model-Checking with Automated Proof Checking. *Computer Aided Verification*, LNCS 939:84–97, Springer Verlag, 1995.

[30] R.K. Brayton, G.D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: a system for verification and synthesis. In *Computer Aided Verification*, volume LNCS 1102, pages 428–432, New Brunswick, NJ, USA, Springer Verlag,1996.

[31] K. Schneider and D. Hoffmann. A HOL Conversion for Translating Linear Time Temporal Logic to ω-automata. *Theorem Proving in Higher Order Logics*, LNCS 1690:255–272, Springer Verlag, 1999.

[32] K. Schneider and T. Kropf. Verifying Hardware Correctness By Combining Theorem Proving and Model Checking. Technical report, University of Karlsruhe, Karlsruhe, Germany, December 1995.

[33] IEEE standard 1364-1995. Ieee standard description language based on the verilog hardware description language, 1995.

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

[35] H. Xiong, P. Curzon, and S. Tahar. Importing MDG Verification results into HOL. *Theorem Proving in Higher Order Logics*, LNCS 1690:293–310, Springer Verlag, 1999.

[36] Y. Xu. *MDG Model Checker User's Manual*. Dept. of Information and Operational Reaserch, University of Montreal, Montreal, Canada, October 1999.

[37] Y. Xu. Model Checking for a First-Order Temporal Logic Using Multiway Decision Graphs. PhD Thesis, University of Montreal, Canada, April 1999.

[38] Z. Zhou, X. Song, S. Tahar, E. Cerny, F. Corella, and M. Langevin. Formal verification of the island tunnel controller using multiway decision graphs. In *Formal Methods in Computer-Aided Design*, volume LNCS 1166, pages 233–247, 1996.

[39] Z. Zhou. *MDG Tools (V1.0) Developer's Manual*, 1996.

[40] Z. Zhou and N. Boulerice. *MDG Tools(V1.0) User's Manual*. University of Montreal, Dept. D'IRO, 1996.

[41] Z. Zhu, J. Joyce, and C. Seger. Verification of the Tamarack-3 Microprocessor in a Hybrid Verification Environment. *In Higher-Order Logic Thoerem Proving and Its Applications, LNCS 780*, pages 252–266., Springer Verlag, 1994.