Integration of HOL and MDG for Hardware Verification

Vijay Kumar Pisini

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfilment of the Requirements

for the Degree of Master of Applied Science at

Concordia University

Montréal, Québec, Canada

February 2000

ABSTRACT

Integration of HOL and MDG for Hardware Verification

Vijay Kumar Pisini

With the ever increasing complexity of the design of digital systems and the size of the circuits in VLSI technology, the role of design verification has gained a lot of importance. Theorem Proving based verification and Decision Diagram based verification are now-a-days the two main techniques used for formal verification. Each of them has its own advantages and disadvantages. In this thesis, we propose a hybrid approach for formal hardware verification which uses the strengths of the theorem prover HOL (Higher-Order Logic) with powerful mathematical tools such as induction and abstraction, and the advantages of the automated tool MDG (Multi-way Decision Graphs) which supports equivalence checking and model checking. We developed a linkage tool between HOL and MDG which uses the specification and implementation of a circuit written in HOL to automatically generate all required MDG files. It then calls the MDG equivalence checking procedure and reports the MDG verification result back to HOL. To illustrate the proposed HOL-MDG hybrid verification we use the Cambridge Fairisle ATM switch fabric as an example.

ACKNOWLEDGEMENTS

My sincere thanks are due to Dr. Sofiène Tahar, my thesis supervisor, for the constant encouragement and support he has extended to me throughout the period of my research work at the ECE Department of Concordia University. The discussions I had with him and the motivation he has given me to work in a group helped me a lot, and I am sure, it will help me in my further studies or work too! I am grateful to him as a person.

I am thankful to Dr. Otmane Ait-Mohamed for the foundation he has laid for my research work and for the constructive criticism about the tackling of my research problem. I specially thank Dr. Xiaoyu Song of University of Montreal, who along with my supervisor provided me the direction, financial support and the computing facilities at University of Montreal. Also, I thank Dr. Paul Curzon of Middlesex University, U.K. who constantly gave me good feedback on my work, as well as Dr. Skander Kort and Mr. Mohamed Hassan Zobair of Concordia University who helped me out with the technical details. I thank the HVG members at Concordia University and all other friends who have contributed to my knowledge and success.

Finally, with utmost respect, I thank my parents for the constant love, encouragement and support they have given me throughout the journey of attaining this degree. Not to mention, my wife's support and patience is appreciated whom I know from the beginning of my studies at Concordia University.

Dedicated To My Parents

Wisdom is better than Rubies – King Solomon

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

The rapid growth in the market gives rise to the demand in lesser and lesser design cycles of a VLSI circuit design, while the complexity of the design is constantly increasing. With ever increasing complexity of the design of digital systems and the size of the circuits in VLSI technology, the role of design verification has gained a lot of importance. Design errors can cause serious failures, resulting in the loss of time, money and it takes a very large amount of time and effort to correct the error, especially when the error is discovered late in the process. For these reasons, we need approaches that enable us to discover errors and validate designs as early as possible. Verification is defined as the validation of the circuit for its correctness. Validation techniques are simulation, testing, prototyping and formal verification. Simulation, which is the state-of-the-art is often used as the main approach for verification (see Fig. 1.1). Despite the major simulation efforts, serious design errors often remain

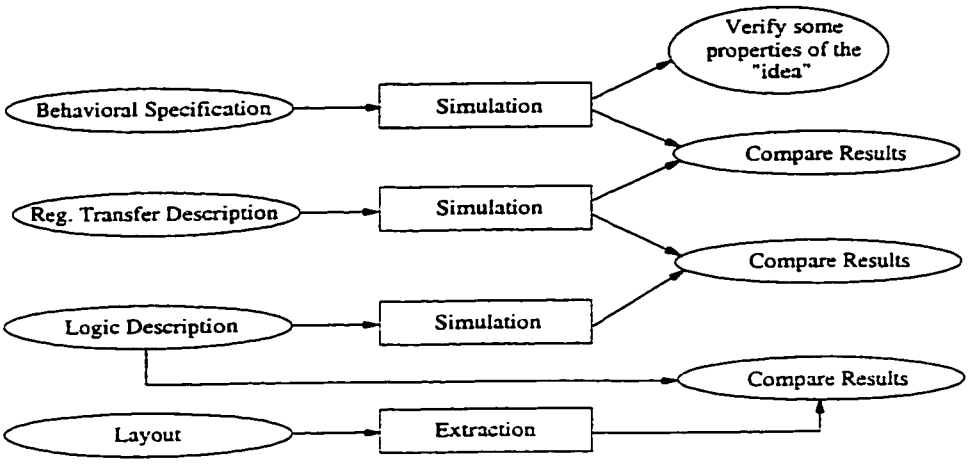

Figure 1.1: Verification by Simulation

undetected which resulted in the evolution of alternative verification approaches such as formal methods in verifying the hardware design. Formal methods have the potential for significantly reducing the number of design faults in VLSI and at the same time reduce the cost of that design. More over, not all the cases are covered with simulation but formal verification provides exhaustive coverage of the test cases. The cost of producing correct designs is increasing. They are especially important in safety-critical and security-critical applications. Formal methods require skills in specification and entail reasoning in some formal logic.

Formal methods have emerged as an alternative approach for traditional validation techniques such as simulation and testing. Some of the limitations of simulation and testing are overcome by their introduction. Formal verification methods fall into one of the two classes: theorem proving based methods and decision graph based methods. The application of formal methods has two main aspects in the

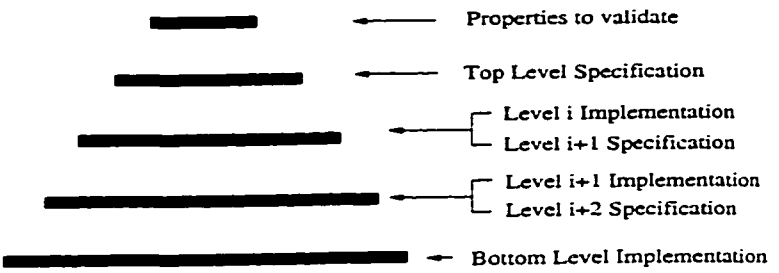

Figure 1.2: Hierarchical Verification

design process. The first is the formal framework used to describe the specification and implementation of a design and the second is the verification techniques and tools used to prove that the implementation satisfies the specification in a formal way. A specification refers to the description of the intended/required behavior of the hardware design. An implementation refers to the hardware design that is to be verified. Various formalisms, such as temporal logic, propositional logic, higher-order-logic, computational tree logic are used for the specification and implementation description framework. Coming to the second aspect, there are several approaches to formal hardware verification such as theorem-proving, model checking, equivalence checking and symbolic simulation [1, 2, 3]. Each of them has its own strengths and weaknesses. *Hierarchical Verification* is best achieved with theorem proving. In hierarchical verification, the lower level specification is equal to the higher level implementation as demonstrated in Fig. 1.2 [2].

$$Level\ i\ Implementation = Level\ i + 1\ Specification$$

In the following subsections of this chapter, each technique is briefly discussed.

1.1 Formal Verfication Techniques

1.1.1 Theorem Proving

In the theorem proving approach, the specification and implementation are usually expressed in first-order or higher-order logic formulae. The relationship between the specification and implementation is formed as a theorem to be proven within the system using the axioms and inference rules. The powerful mathematical techniques such as induction and abstraction are the strengths of theorem proving and make it a very flexible and powerful verification technique. It makes it possible to construct a model at almost every abstraction level and proves properties on all classes of systems.

Theorem-proving methods have been in use in hardware and software verification for a number of years in various research projects. Some of the well-known theorem provers are HOL (Higher-Order Logic) [4], PVS (Prototype Verifiation System) [5, 6], Boyer-Moore [7] and ACL2 [8, 9]. Eventhough they are so powerful, one needs expertise in using a theorem prover. One is expected to know the whole of the design to verify it, as theorem-proving is a white box verification. It is not fully automated and takes large amount of time to verify a circuit and hence the level of its acceptance in industry is very low.

1.1.2 Decision Diagram based Methods

Decision graph based methods use state space algorithms on finite-state models to check if the specification is satisfied. In case the verification fails, the user can track with the counter-example produced as to why it failed. The two main techniques in this category are property checking and equivalence checking. In property checking we have invariant checking where safety properties are verified, and model checking where safety properties as well as liveness properties can be checked. In equivalence checking, we have combinational equivalence and sequential equivalence checking.

Model Checking

Model checking [10] is a decision graph based verification technique. It is an automatic approach for verifying finite-state systems such as sequential circuits. Specifications are expressed in temporal logic and the system is modeled as a state-transition graph. A search procedure is used to determine automatically if the specifications are satisfied by the state-transition graph. The technique was originally developed by Clarke *et al* [11]. Model checking has several important advantages over mechanical theorem proving, most importantly, it is fully automatic. The model checker will either terminate with "true" indicating that the model satisfies the specification or gives a counter-example execution that shows why the formula is not satisfied. Model checkers are unable to handle very large designs due to the *state explosion* problem [10]. Verifying complex systems dramatically changed with

the discovery of representing the transition relation using *reduced ordered binary decision diagrams* (ROBDDs) [12]. Used with the original model checking algorithm, it is called *symbolic model checking* [13]. By using this combination, it is possible to verify large systems.

Equivalence Checking

Equivalence checking is used to prove functional equivalence of two design representations modeled at the same or different levels of abstraction. It can be divided into two categories: one is *combinational equivalence checking*, and the other is *sequential equivalence checking*. Combinational equivalence checking is based on the canonical representations of boolean functions or typically *binary decision diagrams* (BDD). Equivalence checking verifies for all input sequences that an implementation has the same outputs as the specification, both modeled as finite state machines (FSM).

In combinational equivalence checking, the functions of the two descriptions are converted into canonical forms which are then structurally compared. The major advantage of BDDs is their efficiency for a wide variety of practically relevant combinational circuits. The current designs which are clock-driven synchronized designs, need to be separated into small designs. The tool then maps each register of one model into another and compares their combinational circuits between every two consecutive registers. However, combinational equivalence checking cannot

handle the equivalence checking between RTL and behavioral models because these models are developed separately and it is not possible to map each register in the RTL model to that of the behavioral model.

Sequential equivalence checking is used to verify the equivalence between two sequential designs at each state. Sequential equivalence checking considers just the behavior of two designs while ignoring their implementation details such as register mapping. It can verify the equivalence between RTL and netlist or RTL and behavioral model which is very important in design verification. The disadvantage of sequential equivalence checking is that it cannot handle a large design because it encounters *state space explosion* problem very fast.

1.2 Formal Verfication Tools

There are many formal verification tools available at academic and industry level. Some of the main tools are HOL, PVS, SMV, VIS, FormalCheck, MDG along side others. Each of the mentioned tools are briefly described in the following subsections. Significant research work is going on at present to make the tools better to suit the needs of academia and industry. In this thesis, we integrated the HOL and MDG systems which are described in detail in Chapter 3.

1.2.1 HOL

The HOL System [4] is an environment for interactive theorem proving in a higher-order logic, developed at University of Cambridge, U.K. Its most outstanding feature is its high degree of programmability through the meta-language ML [14]. The system has a wide variety of uses from formalizing pure mathematics to verification of industrial hardware. Academic and industrial sites world-wide are using HOL. HOL has a formally defined syntax and semantics. It supports both 'forward' and 'goal-directed' proofs and it is secure, i.e., it can not prove false theorems. A more detailed description of HOL system is provided in Chapter 3 of this thesis.

1.2.2 PVS

The logic of PVS (Prototype Verification System) [5, 6], developed at SRI, California, is a strongly typed higher-order with a rich type system, which includes dependent

types and predicate subtypes. Type checking in this logic is undecidable and requires the assistance of the theorem prover and possibly human intervention to discharge automatically generated type correctness conditions. Definitions and theorems can be grouped into parameterized theories whose parameters may have constraints attached.

Proofs in PVS are backward proofs using a sequent representation for proof obligations. The inference rules operate at a higher level than the primitive inferences of higher-order logic and include instantiation, application of theorems, equality rewriting, as well as complex propositional rewriting. Higher-level proof strategies analogous to HOL tactics can be constructed from the basic inference rules using a specialized strategy language.

To discharge certain classes of obligations, PVS employs an integrated decision procedure for equality reasoning, linear arithmetic, arrays, etc., as well as a BDD-based procedure for propositional logic. The decision procedures are integrated with the type checker to make use of additional constraints available from information. PVS also includes a model checker for finite-state μ-calculus that can be accessed through a formalization of the μ-calculus within the PVS logic [15].

1.2.3 SMV

Symbolic Model Verifier (SMV) [13] is a model checker developed at Carnegie Mellon University. It allows to check that a finite-state system satisfies specifications

given in CTL (Computational Tree Logic) properties [16]. It uses the OBDD-based symbolic model checking algorithm. A specification for SMV is a collection of properties. When the tool fails to check properties of a certain model, it will produce a counter-example. A counter-example is a behavioral trace that violates the specified property. This makes SMV a very effective debugging tool as well as a formal verification system.

A newer version of SMV developed at Cadence [17] includes the concepts *compositional* verification [18] and abstraction which enhance the ease of verification. For large designs, especially those including substantial data path components, the user must break the correctness proof down into parts small enough for Cadence SMV to verify. This is known as *compositional* verification. Cadence SMV also provides a number of tools to help the user reduce the verification of large, complex systems to small finite state problems.

1.2.4 VIS

Verification Interact with Synthesis (VIS) [19], developed at University of California, Berkeley, integrates the verification, simulation and synthesis of finite-state hardware systems. It uses a Verilog front-end and supports model checking, combinational and sequential equivalence checking, cycle-based simulation, and hierarchical synthesis. VIS performs symbolic model checking and reports the failure with a counter-example which is called the "debug" trace. Also, VIS provides the

capability to check the combinational equivalence and sequential equivalence of two designs. Sequential verification is done by building the product finite-state machine, and checking that the outputs are equal for every reachable state of the product machine. VIS also provides traditional design verification in the form of a cycle-based simulator that uses BDD techniques. Since VIS performs both formal verification and simulation using the same data structures, consistency between them is ensured.

1.2.5 FormalCheck

FormalCheck [20] is an industrial model checking tool based on ω-automata [21]. It was first released in April 1997 by Bell Labs Design Automation and now part of the Cadence Design Systems Affirma tool set.

FormalCheck supports the synthesizable subsets of the industry standard hardware description languages, VHDL and Verilog. The user supplies FormalCheck with a set of queries to be verified on the design model written in synthesizableVerilog or VHDL code. The queries are simple temporal statements (formalizations) describing behavioral aspects of the specification. FormalCheck can handle larger designs by using a number of embedded reduction techniques.

1.2.6 MDG

Multiway Decision Graphs (MDGs) [22], developed at University of Montreal, represent and manipulate a subset of first-order logic formulae suitable for high-level

hardware verification. With MDGs, a data value is represented by a single variable of an abstract type and a data operation is represented by an uninterpreted function symbol. The MDG operators and verification procedures are packaged as MDG tools and implemented in Prolog [23]. The MDG tools provide facilities for invariant checking, verification of combinational circuits, equivalence checking of two state machines and model checking. More details about MDG system is described in Chapter 3 of this thesis.

1.2.7 Hybrid Tools

No single formal method is suitable for describing and analyzing every aspect of a complex system. A practical solution is to combine different methods. When combining methods it is important to find a suitable style for using different methods together, and also investigate how these different methods could be used together. Today, there exist a number of hybrid tools which are mainly the combination of theorem proving and model checking.

The remaining chapters of this thesis are organized as follows. Chapter 2 contains related work in the area and the scope of this thesis. Chapter 3 describes the HOL and MDG systems. In Chapter 4 the linking approach for the hybrid tool is presented as to how equivalence checking of an automated tool (MDG) is used in the proof process of an interactive theorem-prover (HOL). In Chapter 5, two examples are presented, the Timing block and the Acknowledgement block (for sequential and

combinational verification respectively) of the Fairisle ATM switch fabric, through which the advantages of this hybrid approach are illustrated. Chapter 6 finally summarizes the work and concludes the thesis.

Chapter 2

Related Work

There exist a number of hybrid approaches such as combining theorem proving with model checking [15, 24] and combining theorem proving and symbolic trajectory evaluation [25].

2.1 Overview of Related Work

HOL and Voss

Joyce and Seger [25] implemented a prototype software tool for their hybrid approach by means of an interface between the Voss system and HOL. A symbolic simulator can be used to verify assertions about the state of a circuit that results from a given sequence of inputs. An extension to symbolic simulation is symbolic trajectory evaluation. In this extension to symbolic simulation, it was made possible

to verify assertions about state trajectories, that is, sequences of states rather than just single states. In addition to treating node values symbolically, symbolic trajectory evaluation provides a rigorous technique for verifying temporal relationships between these node values. In their hybrid system, several predicates were defined in HOL whereby a mathematical link is established between both systems. They have implemented a tactic (SML function) called VOSS_TAC which calls the Voss system which does a part of the verification using symbolic trajectory evaluation to decide whether an assertion is true which in turn can be used by the HOL system to proceed with further verification procedures.

PVS and Model Checking

Rajan *et al* [15] described an approach where a BDD-based model checker for the propositional μ-calculus has been used as a decision procedure within the framework of the PVS proof checker. An extension of the μ-calculus is defined using the higher-order logic of PVS. The temporal operators are then given their customary fixpoint definitions using the μ-calculus. These temporal operators apply to arbitrary state spaces. In the instance when the state type is constructed in a hereditarily finite manner, μ-calcululs expressions are translated into input acceptable by a μ-calculus model checker. This model checker can then be used as a decision procedure within a proof to prove certain subgoals. The model checker accepts the translated input from μ-calculus expression. The generated sub-goals are verified by the model checker and

the results are used in the proof process of PVS.

HOL and Model Checking

Schneider *et al* [24] proposed an approach of invoking model checking from within HOL where properties are translated from HOL to temporal logic. A new class of higher-order formulae were presented, which allows a unified description of hardware structure and behavior at different levels of abstraction. Datapath oriented verification goals involving abstract data types can be expressed by these formulae as well as control dominated verification goals with an irregular structure. To ease the proofs of the goals in HOL, a translation procedure was presented which converts the goals into several CTL model checking problems, which are then solved outside HOL.

HOL and BuDDy

Gordon [26] described the integration of HOL98 with the BuDDy BDD package. HOL was used to formalize the Quantified Boolean Formulae of BDDs. By using a higher-order rewriting tool, the formulae can be interactively simplified to get simplified BDDs. Mapping the simplified formulae to BDDs was done by using a table. The BDD algorithms can also strengthen its deductive ability in this system.

HOL and Model Checking

Aagaard *et al* [27] constructed a system that integrates symbolic trajectroy evaluation based model checking with theorem proving in higher-order logic. The approach is made possible by using the same programming language *fl*, a strongly-typed functional language in the ML family [14], as both the meta and object language of theorem proving. This is done by "lifting" *fl* as they called it in their work [27], essentially deeply embedding *fl* in itself. The approach provides an efficient and extensible verification environment. Their goal in this approach was to move seamlessly between model checking, where *fl* functions are *executed*, and theorem proving, where they *reason* about the behavior of *fl* functions. Those goals are achieved via a mechanism that they referred to as *lifted fl*. The basic concept of *lifted fl* is to use *fl* as both the object and meta language of a proof tool. This approach is applicable to any dialect of the ML programming language and any model-checking algorithm that has inference rules for combining results.

HOL and Gandalf

Hurd [28] described GANDALF_TAC, a HOL tactic that proves goals by calling Gandalf [29] which is a first-order resolution theorem-prover optimized for speed and specializing in manipulations of large clauses, and mirroring the resulting proofs in HOL. Gandalf is a Prosper plug-in [30]. This call can occur over a network, and a Gandalf server may be set up servicing multiple HOL clients. GANDALF_TAC does

not go through all of the proof procedure of the goal, but rather is a component of an underlying proof infrastructure. GANDALF_TAC takes the input goal, converts it to a normal form, writes it in an acceptable format, sends the string to Gandalf, parses the Gandalf proof, translates it to a HOL proof, and proves the original goal.

LTL in HOL

More recently, Schneider and Hoffmann [31] described the embedding of linear time temporal logic (LTL) [32] in HOL together with a translation of LTL formulae into equivalent ω-automata [21]. The translation is implemented by HOL conversions. Its implementation is mainly based on pre-proven theorems. It runs in linear time in terms of the given formula. The main application of this conversion is the sound integration of symbolic model checkers as decision procedures in the HOL theorem prover. The conversion also enables verifying temporal properties within the HOL system.

2.2 Scope of the Thesis

More work is still being done to integrate formal verification tools. Since theorem proving approaches have the flexibility to express the behavior of the circuit at different levels of abstraction with ease which complements the reliability of the obtained results, the formal verification research community is looking more into the ways to integrate the automated tools with theorem provers to reap the advantages of

both. Among the developed hybrid tools, model checking is an automatic technique and is mostly integrated with theorem provers.

In difference to related work, in this thesis we combine theorem proving with automated equivalence checking. We proposed a methodology [33, 34] as to how equivalence checking of the automated MDG system [22] supports the proof process of the HOL theorem prover [4]. The implementation of the proposed methodology is achieved by building a linkage tool using Standard Meta Language (SML) to translate from HOL to MDG. Two tactics (SML functions) MDG_COMB_TAC and MDG_SEQ_TAC are built for translation from HOL to MDG and verification in MDG. Later the results from MDG are imported [35] to HOL.

The linkage tool is built to use the equivalence checking of MDG system, though model checking facility is availble in MDG. The usage of model checking of MDG for the hybrid tool is beyond the scope of this thesis. As a case study example, Fairisle ATM switch fabric [36] is considered. The Timing block of the Fairisle switch fabric is used to demonstrate the functioning of our hybrid tool for sequential verification. For combinational verification, the Acknowledgement block of the Fairisle ATM switch fabric is presented as an example. The Fairisle ATM switch fabric has been independently verified in HOL [37] and MDG [38].

The HOL theorem prover can handle very large circuits for verification but it is a cumbersome and time-consuming process and needs expertise in using it. We believe that the present VLSI industry, however needs the automation of the

verification process as much as possible without suffering the under-capability of the automated tools when it comes to handling large circuits. The integration of interactive and automated tools eases the verification complexity to a great extent as well as significantly reducing the verification time.

The work described in this thesis is a part of a larger project to link VHDL, HOL and MDG as shown in Figure 2.1. Here, the VHDL model is analyzed to get a data structure (Directed Acyclic Graph—DAG) of the model which is passed through an HOL Generator to get the HOL model. Within HOL, we use our hybrid tool which uses MDG system partially to prove some sub-goals as explained later in this thesis. In the case of property verification, an LTL property description (L_{MDG}) [39] is transformed into an equivalent VHDL or MDG-HDL circuit description that will either be fed into the analyzer or directly to the MDG system, respectively.

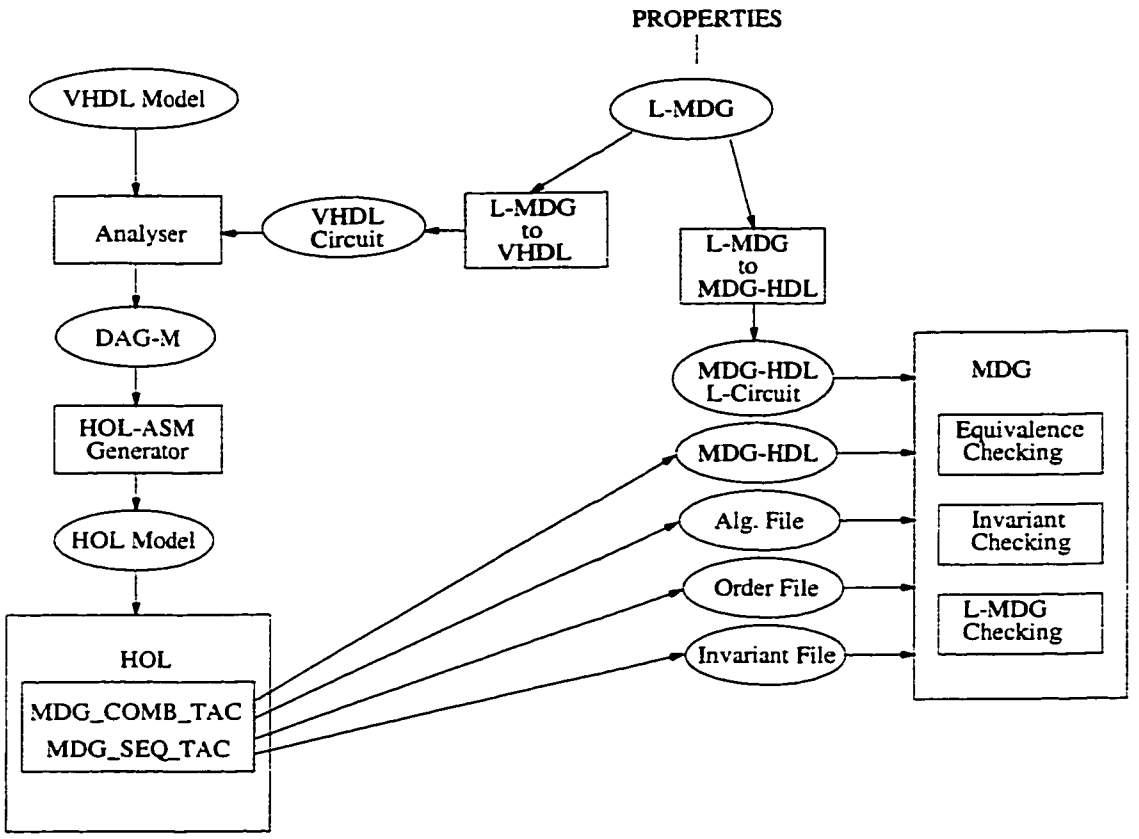

Figure 2.1: VHDL-HOL-MDG Project

Chapter 3

HOL and MDG Systems

3.1 HOL System

The HOL System is a theorem prover based on higher-order logic [4] which was originally intended for use in hardware verification, but now used in a variety of application areas since it is a general purpose proof system. In the theorem proving approach to verification, a system and its properties are described by means of logical formulae and the system is shown by means of a logical proof to entail the desired properties. It allows functions and relations to be passed as arguments to other functions and relations. It provides a wide range of proof commands of varying sophistication, including rewriting tools and decision procedures. Also, it is user programmable, allowing user-defined and application specific proof tools to be developed without compromising reliability.

The HOL system is a descendant of the earlier LCF system [40]. The basic interface to the system is a Standard ML interpreter. SML [14] is both the implementation language of the system and the meta-language in which proofs are written. Proofs are input to the system as calls to SML functions. Roughly speaking, HOL is equal to SML plus: some predefined SML programs (functions), and some data type declarations. Higher-order logic is very flexible and has a well-defined and well-understood semantics. The HOL system supports forward proof and goal-directed backward proofs in a natural-deduction-style calculus by creating theorems and applying inference rules to the already created theorems.

HOL has many built-in inference rules and ultimately all theorems are proven in terms of the axioms and basic inferences of the calculus. By applying a set of primitive inference rules, a theorem can be created. Once a theorem is proven, it can be used in further proofs without recomputation of its own proof. In the backward proof, the user sets the desired theorem as a goal. Tactics are applied to the goal to create sub-goals and inference rules are applied to prove the sub-goals which in turn proves the main goal. The system is guided by applying *tactics* to proof obligations; a tactic is an SML function that corresponds to a high-level proof step and automatically generates the sequence of elementary inferences necessary to justify the step. Tactics are used in backward proofs and inference rules are used for forward proofs. Tactics can be composed into even larger steps using *tacticals* such as "apply tactics A then B and then C repeatedly until no further

simplification is obtained." A notable aspect of the system is that user-defined tactics cannot compromise the soundness of a proof because the basic inferences operate on proof states. The results are strong and the user can have great confidence since the most primitive rules are used to prove a theorem. HOL system also has automatic recursive type definitions, structural induction tools, rewriting tools (from LCF), automatic primitive recursive definitions, built-in theories of arithmetic, lists, sets, tautology checker, automatic inductive definitions, parser and pretty-printer generator, online help facility and it has full documentation.

The applications of the HOL system can be found in hardware verification, reasoning about security, verification of fault-tolerant computers, reasoning about real-time systems. It is also used in compiler verification, program refinement calculus, software verification, modeling concurrency and automata theory. HOL also allows the use of hierarchical verification methodology wherein the modules are divided into sub-modules and even the sub-modules are divided until the lowest implementation level is reached. The behavioral and structural specifications of each module are expressed in higher-order logic and each module is verified by proving a theorem stating that the implementation implies the specification. Each sub-module is verfied, and its result is used to verify the other sub-modules as needed. HOL scales better than decision diagram based tools as illustrated by related work on microprocessor verification [41, 42, 43] which are beyond the capabilities of such tools due to *state space explosion* problem. To complete a verification, however,

a very deep understanding of the internal structure of the design is required, as it is a white-box approach. Modeling and verifying a system using HOL is very time-consuming [44, 45].

3.2 MDG System

The MDG system is a decision diagram based verification tool, primarily designed for hardware verification which allows equivalence checking and model checking. It is based on Multiway Decision Graphs (MDGs) [22] — an extension of the traditional ROBDDs [12] which accommodates abstract sorts and uninterpreted function symbols. The MDG verification approach is a black-box approach. During the verification the user does not need to understand the internal structure of the design being verified. The strength of MDG is its speed and ease of use. The MDG hardware verification system has been used in the verification of significant hardware examples [46]. A fundamental primitive of its hardware description language is the table which is a general form of the well-known truth tables. Used with don't-care and default values, next state variables and variable entries, it becomes a powerful specification construct that can be used to give behavioral specifications of hardware as *abstract state machines* (ASM) [22]. The MDG tools combine the advantages of representing a circuit at higher levels as is possible in a theorem prover, and the advantages of the automation offered by ROBDD based tools.

3.2.1 Multiway Decision Graphs

Multiway Decision Graphs (MDGs) have been proposed as a solution to the data path width problem of ROBDD based verification tools. The formal system behind MDGs is a subset of many-sorted first-order logic with distinction between *abstract sorts* and *concrete sorts*. Concrete sorts have finite *enumerations*, while abstract sorts do not. The constants occuring in enumerations are referred to as individual constants, and the other constants as generic constants. Concrete symbols must have explicit definition.

An MDG is a finite, directed acyclic graph (DAG). An internal node of an MDG can be a variable of a concrete sort, or it can be a variable of abstract sort or it can be a cross-term (whose top-level function symbol is a cross-operator). A cross-operator is a function which takes inputs of abstract sorts and/or concrete sorts and it's output is always of concrete sort. An MDG may only have one leaf node denoted by **T** which means all paths in the MDG are true formulae. Thus, MDGs essentially represent relations rather than functions. MDGs can also represent sets of states. Using MDGs, a data value can be represented by a single variable of an abstract sort, rather than by concrete variables. Variables of concrete sorts are used for representing control signals, and variables of abstract sorts are used for representing datapath signals. They are much more compact than ROBDDs for designs containing a datapath. Furthermore, sequential circuits can be verified independently of the width of the datapath. For circuits with large datapaths,

the representation of MDGs are much more compact than that of ROBDDs, thus greatly increasing the range of circuits that can be verified since the verification is independent of the width of the datapath.

A state machine is described using finite sets of input, state and output variables. The behavior of a state machine is defined by its transition/output relations, together with a set of initial states. Similar to the way ROBDDs are used to represent sets of states and transition/output relations for finite state machines (FSMs), MDGs are used to compactly encode sets of states and transition/output relations for ASM. Starting from the initial set of states, the set of states reached in one transition is computed by the relational product operation. Like ROBDDs, MDGs must be *reduced* and *ordered*.

3.2.2 MDG Verification Procedures

The MDG tools package the basic MDG operators and verification procedures [47]. The verification procedures are combinational and sequential verification. The combinational verification provides the equivalence checking of two combinational circuits. The sequential verification provides invariant checking and equivalence checking of two state machines. The MDG operators and verification procedures are implemented in Quintus Prolog [23].

- *Combinational Verification* : The MDGs representing the input-output relation of each circuit are computed using the relational product of the MDGs

of the components of the circuits. Then, taking advantage of the canonicity of MDGs, it is verified whether the two MDG graphs are isomorphic.

- *Invariant Checking* : Using symbolic reachability analysis, the state space of a given sequential circuit (an ASM) is explored in each state. In each state it is verified that the specified property is satisfied. The transition relation of the ASM is represented by the relational product which computes the product machine of the component ASMs.

- *Sequential Verification* : The behavioral equivalence of two sequential circuits (ASMs) is verified by checking that the circuits produce the same sequence of outputs for every sequence of inputs. This is achieved by forming a circuit consisting of the two circuits, feeding the same inputs to both of them, and verifying an invariant asserting the equality of the corresponding outputs in all reachable states.

- *Model Checking* : Model checking feature has been recently developed and incorporated into the existing MDG system [48, 49]. This provides both safety and liveness property checking using the implicit abstract enumeration of an ASM. The properties are represented in a first-order linear time temporal logic, called L_{MDG} [39]. The ASM model of the L_{MDG} formula is constructed, along with a simplified invariant. The ASM of the L_{MDG} formula is composed with the original model and the simplified invariant is checked on the composite

machine, using the implicit abstract enumeration of an ASM.

- *Counter – example Generation* : When invariant or model checking fails, the MDG tools generate a counterexample to help with identifying the source of the error. A counterexample consists of a list of assumptions, input and state values in each clock cycle, which provides a trace leading from the initial state to the faulty behavior.

3.2.3 MDG-HDL

MDG-HDL is the input language for MDG. It supports structural descriptions, behavioral ASM descriptions or a mixture of both. A structural description is usually a netlist of components connected by signals. A behavioral description is given by a tabular representation of the transition/output relation. The MDG-HDL comes with a large library of predefined, commonly used, basic components (such as logic gates, multiplexers, registers, bus drivers, ROMs, etc.) [47]. A circuit description includes the definition of signals, components and the circuit outputs. Signals are declared along with their sorts. Components are declared by the instantiation of the input/output ports of a predefined component module.

For example, a multiplexer with a control signal select of concrete sort having [0,1,2] as an enumeration, inputs: x0, x1, x2 of an abstract sort and output: y of the same abstract sort is defined as:

```
component(mux1,mux(sel(select),
            inputs([(0,x0),(1,x1),(2,x2)]), output(y))
```

Among predefined modules we have a special module called a table. Tables can be used to describe a functional block in the implementation, as well as in the specification. A table is essentially a series of lists, together with a single final default value. The first list contains variables and cross-terms. 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 last element in the list of values could be a first order term.

A table can be thought of as taking 5 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.

For example, a 2-input AND gate can be described as a table as:

```
table([[x1,x2,y], [0,*,0], [1,0,0] | 1])
```

The necessary files for verification in MDG for equivalence checking are: a

behavioral specification file, a circuit description file, an algebraic file, a symbol order file, and an invariant file [47]. The behavioral specification file declares signals and specifies the behavior of the circuit using tables as described above. The circuit description file declares signals and their sort assignments and describes the circuit netlist. The algebraic specification file defines sorts, function types and generic constants. The symbol order file provides the user-defined symbol order for all the variables and cross operators which would appear in MDGs. The invariant file takes the corresponding outputs from both behavioral specification and circuit description for equivalence checking using MDGs.

Chapter 4

Linking Methodology

4.1 Hierarchical Verification

In the hybrid approach, a hierarchical hardware verification methodology is followed. Generally, when we use HOL to verify a design, the design is modeled as a hierarchy structure with modules divided into sub-modules as shown in Fig. 4.1. The sub-modules are repeatedly subdivided until eventually the logic gate level is reached.

By proving a theorem saying that the implementation (structure) implements the specification (behavior), we accomplish the verification of each module. That is:

$$\vdash Implementation\_A \Longrightarrow Specification\_A \qquad (4.1)$$

The verification starts in HOL with a goal to be proved. The correctness theorem for each module states that its implementation down to the logic gate

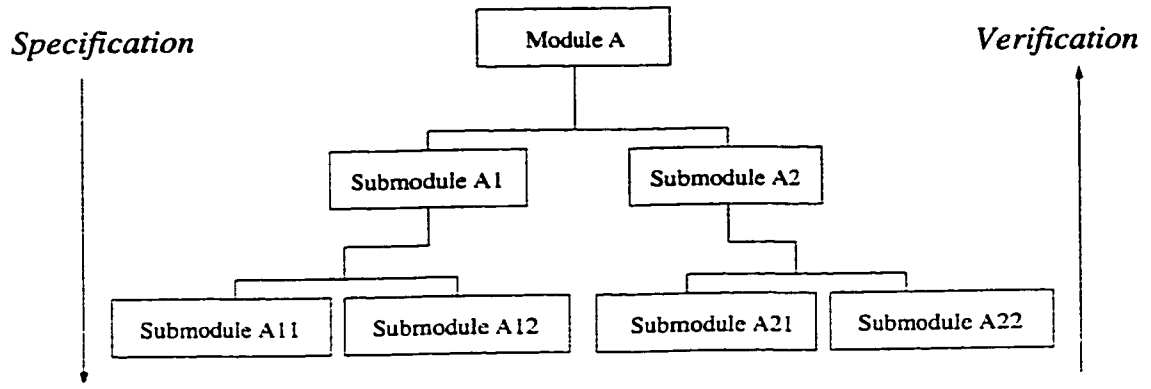

Figure 4.1: Hierarchical Verification of a Module

level satisfies the specification. The correctness theorem for each module can be established using the correctness theorems of its sub-modules. When the module is sub-divided, then we can write the theorem about the structural description as

$$\vdash Implementation\_A = Imp\_A1 \wedge Imp\_A2 \tag{4.2}$$

Now (4.1) can be written as

$$\vdash Imp\_A1 \wedge Imp\_A2 \Longrightarrow Specification\_A \tag{4.3}$$

The correctness statements of the sub-modules $A1$ and $A2$ can be used to prove the correctness theorem for the module A. Likewise we can prove independently for each sub-module that

$$\vdash Imp\_A1 \Longrightarrow Spec\_A1 \tag{4.4}$$

$$\vdash Imp\_A2 \Longrightarrow Spec\_A2 \tag{4.5}$$

Since these are implications, to prove (4.1), it is enough to prove that

$$\vdash Spec\_A1 \wedge Spec\_A2 \Longrightarrow Specification\_A \qquad (4.6)$$

Similarly, $A1$ is verified from its sub-modules $A11$ and $A12$, and $A2$ is verified from its sub-modules $A21$ and $A22$. Hence, we verify module A by independently verifying its sub-modules $A1$ and $A2$. Using this top-down approach, the main objective of this work is to identify and prove the correctness of certain sub-modules in an automatic fashion using the MDG system. In MDG, each selected sub-module will be proved by automatic verification that its implementation is equivalent to its specification, and the result is imported into HOL. In our hybrid system, the sub-module is treated as a black-box.

4.2 HOL and MDG Interface

In HOL, the specification and implementation are expressed in higher-order logic. The MDG system uses MDG-HDL to describe the implementation and the specification, the latter is written in the table form [50]. The sub-goals from the main goal are generated by HOL. The user decides if the sub-goal can be proved in MDG and its description is written in an MDG-acceptable form using the description predicates. In case a sub-goal cannot be expressed in the MDG acceptable form or the MDG verification fails, then the regular HOL proof procedure is followed. Once all the sub-goals are proved, it implies that a HOL proof for the main goal can

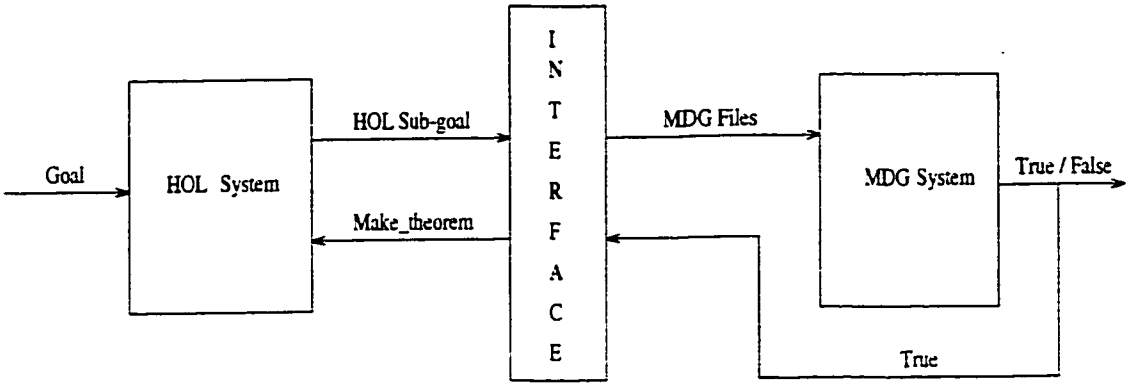

Figure 4.2: Block Diagram of the Hybrid System

be generated and the circuit is formally verified. As shown in the block diagram of the hybrid system in Fig. 4.2, the interface converts the HOL descriptions to equivalent MDG files and all required files for the MDG verification as specified in the following.

The sub-goal specification and implementation which are in two separate files are given as input to the interface which is built in SML. The two HOL files contain the inputs, outputs, intermediate outputs and their signal types. The potential user-defined types are provided in the HOL specification file. From the given two HOL files, corresponding MDG circuit description, specification, algebraic, order and invariant file are created automatically. In the case where the equivalence checking has succeeded, MDG returns "true", this result is imported into HOL in the form of a theorem (using the HOL built-in function *mk_thm* in HOL) and the main proof procedure continues in HOL with the next sub-goal to be proved.

Xiong *et al* [35] showed as to how the results of MDG can be imported into

HOL. [35] provides a formal proof for the soundness of imported verification results from MDG to HOL. Since one cannot assume that a piece of hardware verified in MDG can be taken as a theorem in a HOL proof, [35] shows a way as to how the MDG results are converted to appropriate HOL theorems as used in traditional HOL hardware verification in the style of Gordon [51]. Formalizations of MDG results in HOL were given based on the semantics of the MDG input language. Then theorems were derived which show that the results can be converted between these two forms. First it was needed to formalize the results of the MDG verification applications in HOL. To do that, a series of translation theorems (one for combinational verification, one for sequential verification, etc.) had to be proved that state how an MDG result can be converted to the traditional HOL form:

$$\vdash \mathit{Formalized\ MDG\ Result} \Longrightarrow (\mathit{Implementation} \Longrightarrow \mathit{Specification})$$

As part of the build-up of the mathematical interface between the two tools, we use above proof to safely import MDG equivalence checking results to HOL. Besides, we base our work on [50] where Curzon *et al* formally specified and verified the total MDG component library in HOL. They also suggested an embedding of MDG tables in HOL.

The HOL and MDG interface is finally summarized in Fig. 4.3 which explains the internal structure of the hybrid tool in detail. First, a tactic invokes the translation and the necessary files generation, and once the files are generated, the

verification begins and the obtained validation result is imported into HOL for further verification proof process. All the file generators (Algebraic, Order, Invariant, Specification, Implementation) shown in Fig. 4.3 are associated each to a specific generator in the translator implementation.

We developed two tactics, MDG_COMB_TAC (for combinational verification) and MDG_SEQ_TAC (for sequential verification) which start within HOL the translation of the files, calls the verification in MDG, and analyses the result to eventually generate a theorem. The operations performed by these tactics are shown as a flow-diagram in Fig. 4.4. These tactics are explained in the next section.

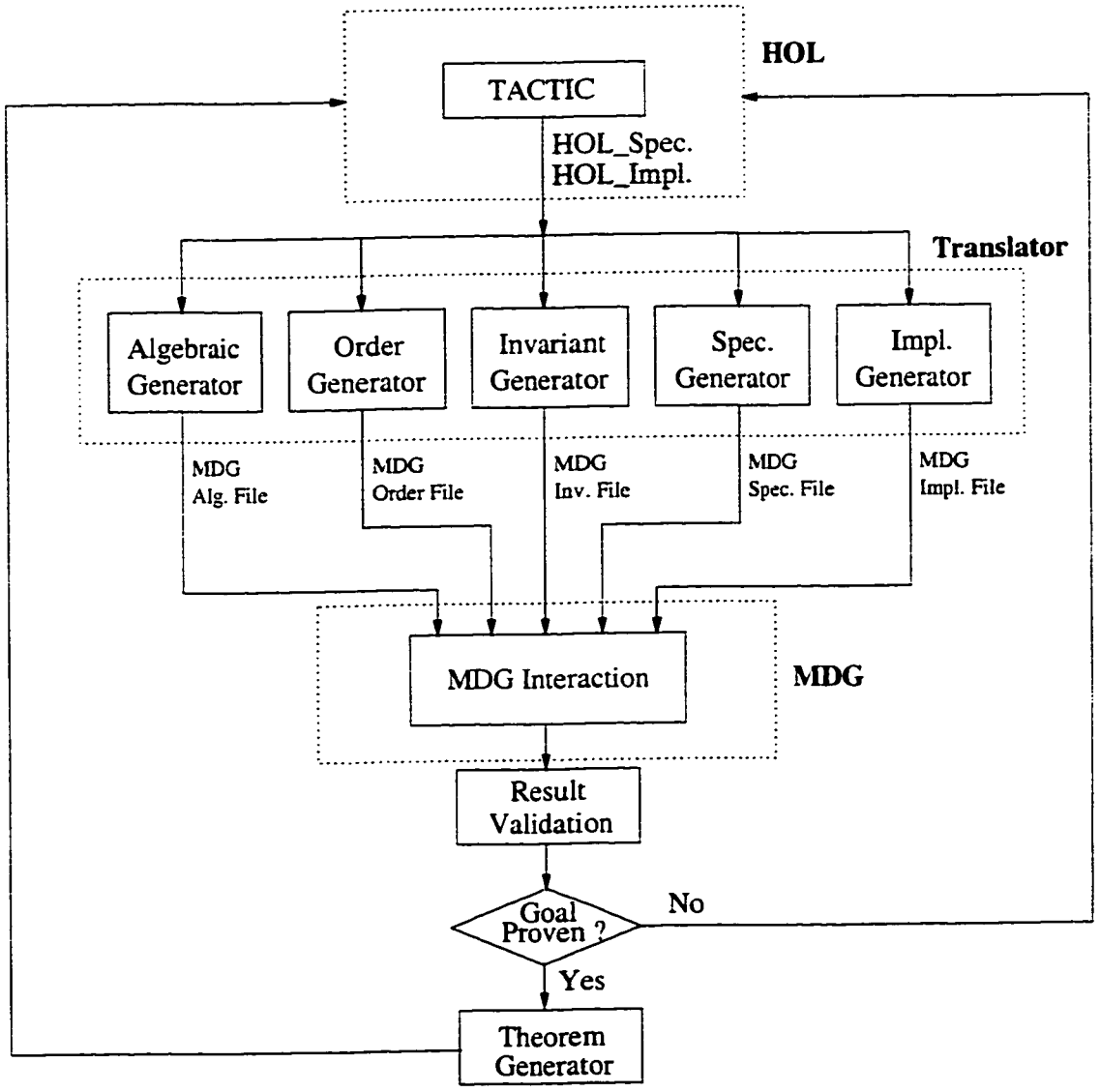

Figure 4.3: Internal Structure of the Hybrid Tool

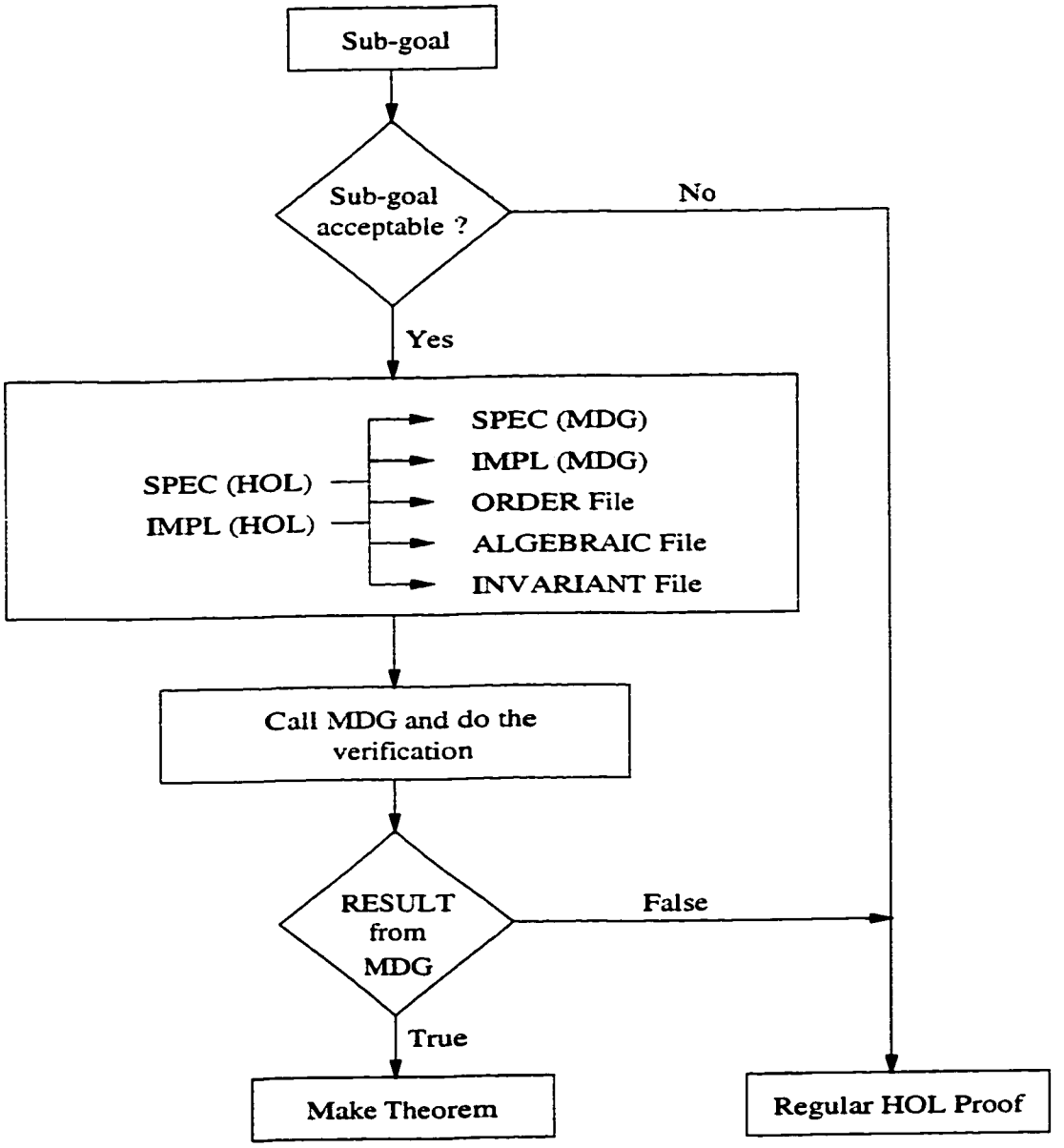

Figure 4.4: Task of MDG_COMB_TAC/MDG_SEQ_TAC

4.3 MDG_COMB_TAC and MDG_SEQ_TAC

The implementation language for HOL is the Standard Meta Language (SML) [14], and for MDG it is Prolog [23]. The translator was built in SML in order to facilitate the tactics MDG_COMB_TAC and MDG_SEQ_TAC to be invoked within HOL. SML is a *functional programming* language. Functional programming consists of functions operating on data structures. SML protects programmers from their own errors. Before a program may run, the compiler checks that all module interfaces agree and that data types are used consistently. SML supports a level of abstraction that is oriented to the requirements of the programmer. The SML system can preserve this abstraction, even if the program is faulty. It is a strictly typed language. SML was primarily designed for theorem proving.

Either MDG_COMB_TAC or MDG_SEQ_TAC, when invoked within HOL takes the files which contain HOL specification and implementation and processes the translation and verification in MDG. The implementation of both tactics is given below in detail.

Input needed: HOL descriptions (Specification and Implementation in HOL). The Specification also contains newly defined types if any.

Output produced: The MDG files are created and a window is opened for MDG verification.

HOL Specification contains:

- High level definition of circuit behavior using specification tables
- Initial values and initial variables (optional)
- New types definitions (optional)

HOL Implementaion contains:

- Definition of the circuit components as predicates based on the MDG component library
- Initial values and initial variables (optional)

HOL Specification Translation:

1. Read the file
2. Separate tables, initial values & initial variables and new types
3. Take tables and process each table

   Processing Tables:

   (a) Capture inputs, outputs and their types. Output may be a next state variable

   (b) Translate the values to 0's and 1's or an enumerated type

(c) Translate the default value and add to the outputs

(d) Format inputs and outputs

4. Capture the initial values and initial variables

5. Get the inputs and outputs from all tables and transfer them to an auxiliary order file. This auxiliary file which eventually contains inputs, outputs and internal signals from implementation file is used to create the order file.

6. From the high-level definition which includes all the defined specification tables, transfer the final outputs of the circuit to an auxiliary invariant file (in the case of sequential verification only). This auxiliary file which eventually contains the outputs from implementation file is used to create the invariant file.

7. Write all signals (inputs, outputs and internal signals) with types into the specification file

8. Join the formatted tables, formatted circuit outputs, next states, state to next state partitions as needed (for sequential or combinational circuit) and write into a file which is in an MDG acceptable file format

Algebraic File Generator:

1. Read the specification file and capture the new types

2. Write the formatted list of strings into a file to create the Algebraic File with other necessary details

HOL Implementation Translation:

1. Read the file
2. Separate initial values, initial variables and the definition of the circuit
3. Process the definition of the circuit to an acceptable MDG format
4. Get the inputs and outputs with their types of each predicate and transfer them to add to the auxiliary order file which was already created from the specification
5. From the definition of the circuit, get the circuit outputs and transfer them to add to the auxiliary invariant file which was already created from the specification
6. Write all signals (inputs, outputs and internal signals) with types into the specification file
7. Join the formatted tables, formatted circuit outputs, next states, state to next state partitions as needed (for sequential or combinational circuit) and write into a file in an MDG acceptable format

Order File Generator:

1. Read the auxiliary order file (created from specification) and create the order of variables by first placing the inputs, internal signals and outputs supplied by implementation into the Order File

2. Add to the file, the inputs and outputs supplied by the specification

   Note: This order respects the rules of MDG in writing the state & next state order, etc. [47]

Invariant File Generator (for sequential verification only):

1. Read the auxiliary file and join the corresponding outputs of the circuit using a fork for equivalence checking and write them into Invariant File which is acceptable by MDG

   Note: Number of outputs from specification *must* be equal to the number of outputs from implementation. Obviously they are of the same type.

Once all the five files (Specification, Implementation, Algebraic, Order, Invariant) are created, a window is opened with Prolog prompt and all these files are fed to it to invoke the MDG verification. The traces from MDG are captured in a file to analyze and validate the result from MDG to be imported into HOL.

Chapter 5

Case Studies

5.1 Sequential Verification

For illustration purposes, we show the verification of two sub-modules of the Fairisle ATM switch fabric [36] (see Fig. 5.1). Curzon [37] formally verified this ATM switching element using the theorem-prover HOL. Tahar *et al* reverified it using MDG [38]. The Fairisle switch fabric is a real switch fabric designed and in use at University of Cambridge for multimedia applications. The Fairisle switch forms the heart of the Fairisle network. Considering the fabric as the main module to be verified, it can be split into 3 sub-modules, namely Acknowledgement, Arbitration and Data Switch. Further dividing the Arbitration sub-module, we have the Timing, Decoder, Priority Filter and Arbiters as sub-sub-modules (Fig. 5.1). In our sequential verification example, we have taken the Timing block to be a sub-sub-module (one of

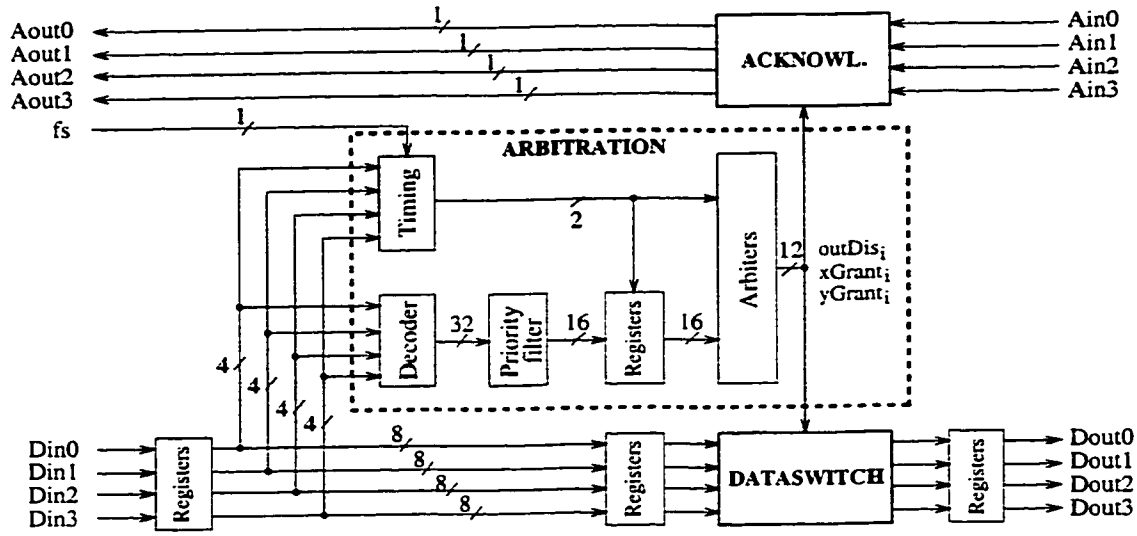

Figure 5.1: Fairisle ATM Switch Fabric

the sub-goals) and used our hybrid tool to achieve the desired verification.

5.1.1 Proof Structure of the Fairisle ATM Switch Fabric

The verification of the Fairisle switch fabric is arranged according to the division of the fabric in a hierarchical fashion as shown in Fig. 5.2.

The goal is to prove that

$$\vdash Fabric\_Imp \Longrightarrow Fabric\_Spec \qquad (5.1)$$

From Fig. 5.2 and the equations in Section 4.1, we have

$$\vdash Fabric\_Imp = Ack\_Imp \wedge Arb\_Imp \wedge DataSw\_Imp \qquad (5.2)$$

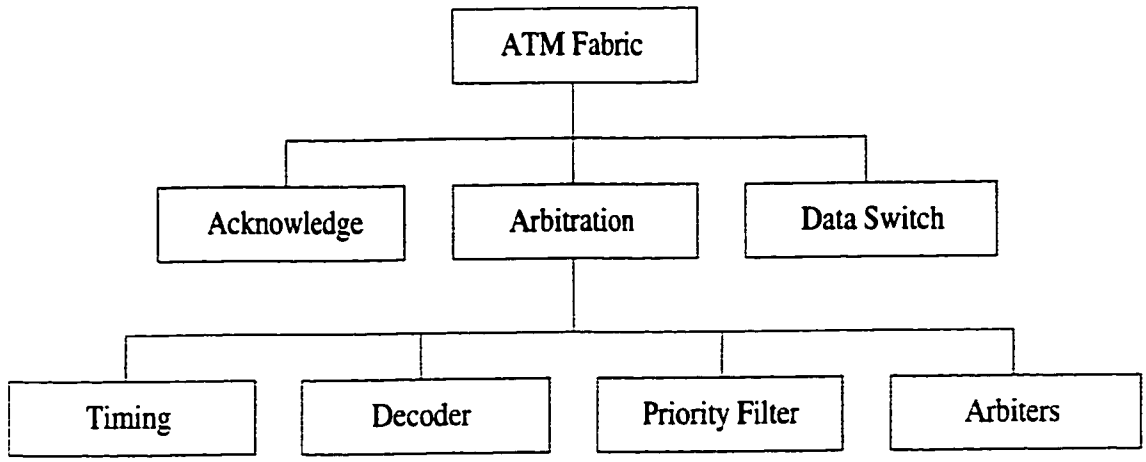

Figure 5.2: Hierarchical Verification of the Fairisle Switch Fabric

as in (4.4) and (4.5) of Chapter 4, we can prove that

$$\vdash Ack\_Imp \Longrightarrow Ack\_Spec \tag{5.3}$$

$$\vdash Arb\_Imp \Longrightarrow Arb\_Spec \tag{5.4}$$

$$\vdash DataSw\_Imp \Longrightarrow DataSw\_Spec \tag{5.5}$$

Now it is enough to prove that

$$\vdash Ack\_Spec \wedge Arb\_Spec \wedge DataSW\_Spec \Longrightarrow Fabric\_Spec \tag{5.6}$$

Likewise, at the next lower level the Arbitration block is proved in the same fashion. In this Arbitration block, one of the sub-modules or sub-goal is the Timing block. Instead of proving the implication in HOL, it can be proved using equivalence in MDG which we illustrate in the following section.

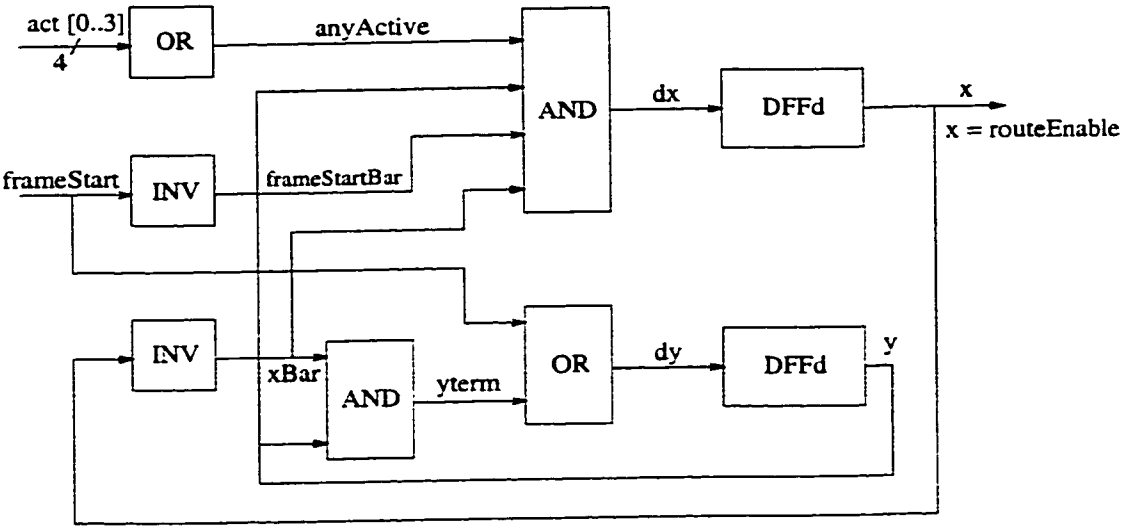

Figure 5.3: Implementation of the Timing Block

5.1.2 Timing Block Description

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 is shown in Fig. 5.3 and the FSM representation is shown in Fig. 5.4.

The specification and implementation were written and used by the hybrid tool to do the verification using equivalence checking of MDG. The implementation of the Timing block shown in Fig. 5.3 described in HOL is:

```
⊢ ∀ frameStart act0 act1 act2 act3 routeEnable.
    TIMING_IMP ((frameStart act0 act1 act2 act3))
               ((routeEnable)) =
    ∃ anyActive frameStartBar x xBar y yterm dx dy .
```

```
(or4 act0 act1 act2 act3 anyActive) ∧
(not frameStart frameStartBar) ∧
(not x xBar) ∧
(and xBar y yterm) ∧
(and4 anyActive y frameStartBar xBar dx) ∧
(or frameStart yterm dy) ∧
(reg dx x) ∧
(reg dy y) ∧
(fork x routeEnable)
```

where "or4" is a 4-input OR gate, "and4" is a 4-input AND gate. "not" and "or" are the regular logic components. "fork" is used to represent the equality of two signals and "reg" is a register. All these components are pre-defined in MDG component library and are defined [50] in HOL.

The resulting MDG-HDL implementation of the Timing block which is equivalent to that of HOL, as generated by MDG_SEQ_TAC is:

```
component(anyActive_impl,or4(input(act0,act1,act2,act3),
                output(anyActive))).
component(frameStartBar_impl,not(input(frameStart),
                output(frameStartBar))).
component(xBar_impl,not(input(x),output(xBar))).
```

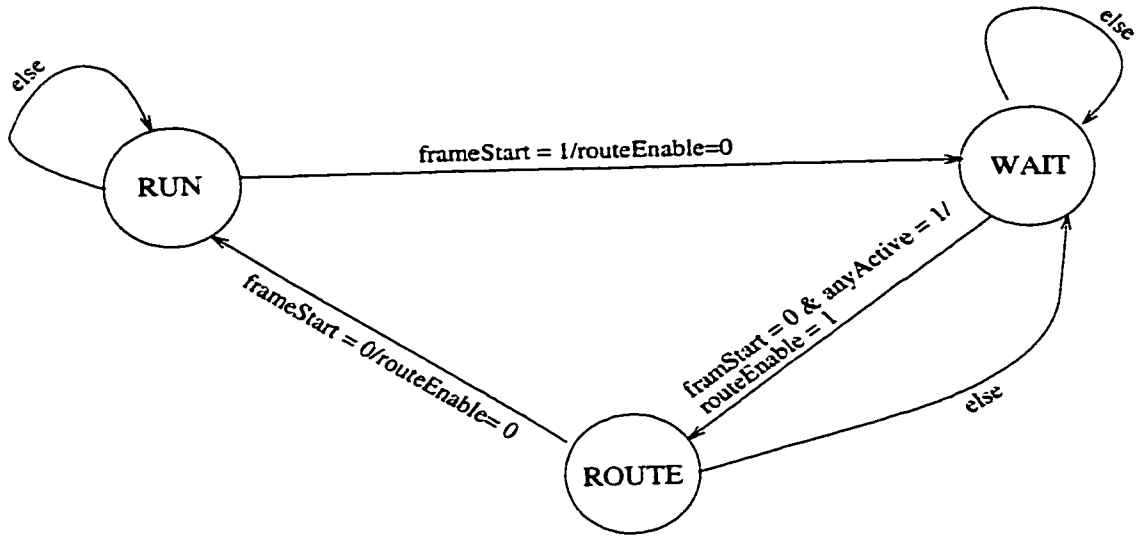

Figure 5.4: State Transitions of the Timing Block

```
component(yterm_impl,and(input(y,xBar),output(yterm))).
component(dx_impl,and4(input(anyActive,y,frameStartBar,xBar),
          output(dx))).
component(dy_impl,or(input(frameStart,yterm),output(dy))).
component(x_impl,reg(input(dx),output(x))).
component(y_impl,reg(input(dy),output(y))).
component(fork_for_routeEnable_impl,fork(input(x),
          output(routeEnable))).
```

The specifications of the Timing block in HOL and MDG are shown below. The HOL specification of the Timing FSM is described using a state transition function and an output function. The HOL definition of the state transitions of the FSM in Fig. 5.4, written in terms of the table specification is given as [50]:

```
TABLE [anyActive;frameStart;timing_state](n_timing_state o NEXT)
      [[DONT_CARE;TABLE_VAL(TRANS T);TABLE_VAL(STATE run)];
      [DONT_CARE;TABLE_VAL(TRANS F);TABLE_VAL(STATE run)];
      [TABLE_VAL(TRANS T);TABLE_VAL(TRANS F);
       TABLE_VAL(STATE WAIT)];
      [DONT_CARE;TABLE_VAL(TRANS T);TABLE_VAL(STATE route)]]
      [waitSIG;runSIG;routeSIG;waitSIG] waitSIG
```

where TABLE_VAL is is defined as a new HOL type, DONT_CARE is defined for the *don't care* condition and runSIG, waitSIG and routeSIG are the lifted versions for the constants *run*, *wait* and *route* respectively. TRANS and STATE are defined in HOL as a common type for all the input variables and state variables respectively. NEXT states that n_timing_state is the next state of timing_state in this case.

The equivalent MDG table specification of the Timing FSM state transition is generated using MDG_SEQ_TAC as:

```
[[anyActive, frameStart, timing_state, n_timing_state],
[*,1,run, wait],
[*,0,run, run],
[1,0,wait, route],
[*,1,route, wait] | wait]
```

Once the specification and implementation written in HOL are translated, MDG_SEQ_TAC generates the required order file, algebraic specification file and invariant file and calls the MDG tool for equivalence checking. The succeeded result from MDG is imported into HOL as a theorem. And hence the verification of the Timing block is done.

5.1.3 Timing Block Verification

We have shown that:

$$Timing\_Imp \equiv Timing\_Spec \text{ (equivalence)} \tag{5.7}$$

We got the above result from MDG and it is imported into HOL [35] as:

$$\vdash Timing\_Imp \Longrightarrow Timing\_Spec \tag{5.8}$$

Using similar MDG proofs for the other sub-modules of the arbitration block, we can get:

$$\vdash Timing\_Spec \wedge Decoder\_Spec \wedge PFilter\_Spec \wedge Arbiters\_Spec \Longrightarrow Arb\_Spec \tag{5.9}$$

Hence proving the higher-level sub-goal for the whole arbitration block.

We showed using MDG that the structural description (i.e. implementation) is equivalent to a high-level specification, described in terms of tables. Writing the

MDG Nodes	CPU Time (sec.)	Memory (MB)
227	0.41	0.161

Table 5.1: MDG Equivalence Checking Results for Timing Block

high-level specification using the tables in MDG is far easy compared to writing it down in HOL. In HOL, the proof is interactive and is time-consuming [37].

Using our hybrid tool, the procedure is faster than proving in HOL that the implementation implies the high-level specification. Curzon [37] took several hours to do the proof of the Timing block in HOL whereas the verification is done in less than a second in MDG (see Table 5.1). It took six man hours to write the HOL specification and implementation files. The automatic translation to the MDG system proved effective in this case, reducing the specification and verification time significantly. The verification results obtained by means of equivalence checking can be formally related to higher levels of abstraction. Also, *equivalence* is a stronger result compared to *implication*.

5.2 Combinational Verification

As an example for the combinational verification we have considered the Acknowledgement block of the same ATM switch fabric which we used for sequential verification. This acknowledgement block is considered as a sub-module of the ATM

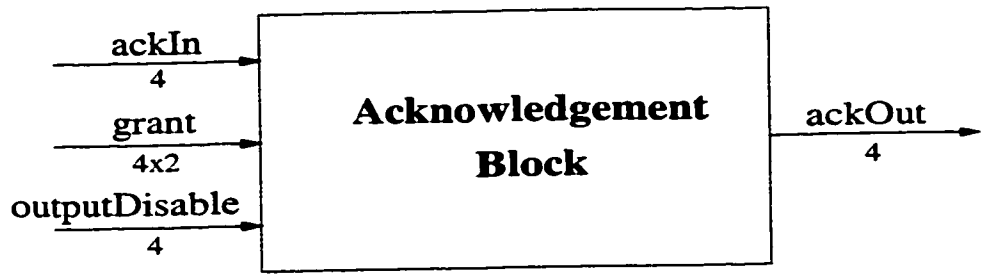

Figure 5.5: Block Diagram of the Acknowledgement Block

switch fabric when the hierarchical verification approach is used. The acknowledgement block produces the acknowledgement signals from the result of arbitration (see Fig. 5.1). The block diagram of the Acknowledgement block is shown in Fig. 5.5.

The Acknowledgement block takes as input acknowledgement signals $ackIn$, as from the output ports, signals indicating *grants* as to which request was granted for each output port and disable signals *outputDisable*, for each output port. It outputs acknowledgement signals *ackOut* for each input port.

The acknowledgement block gives the acknowledgement signal for a single input port on a single clock cycle. The input port receives a positive acknowledgement provided it has been selected by some output port, the disable signal is not asserted indicating that the grant is valid at that time and provided the acknowledgement signal from the granted output port is positive. It has two main units ACKGEN and ACKOR. The detailed block diagram is shown in Fig. 5.6.

ACKOR_N combines the acknowledgements for all the input ports giving one bit per input port. If any output acknowledges an input, then an acknowledgement signal is sent to that input. The implementation of ACKOR_N is shown in Fig. 5.6.

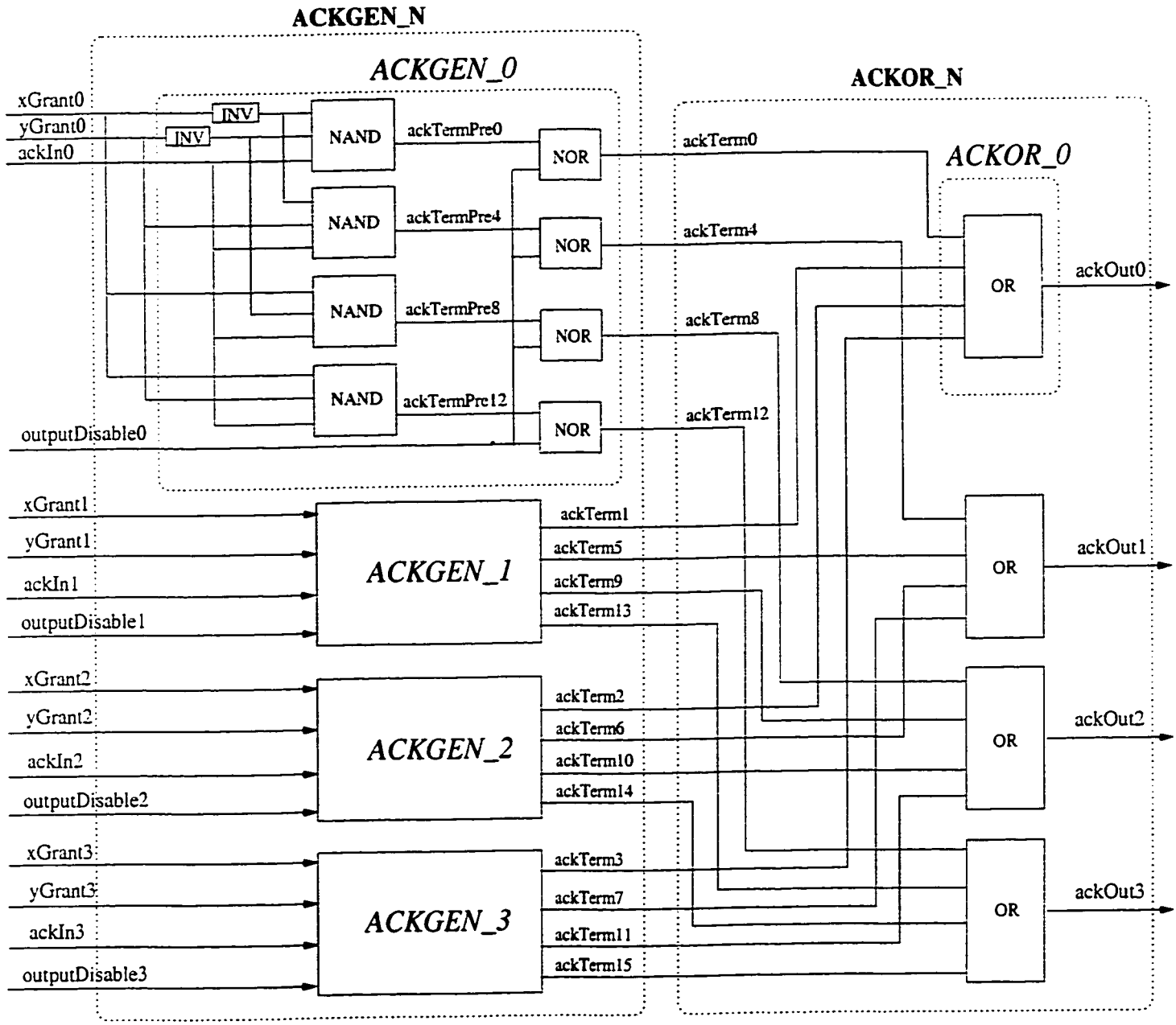

Figure 5.6: Implementation of the Acknowledgement Block

ACKGEN_N generates the acknowledgement signals for the output ports. The implementation of ACKGEN_N is shown in Fig. 5.6. If outputs are disabled for a port or the corresponding external port is sending a negative acknowledgement, then all inputs are sent a negative acknowledgement from that port. Otherwise the granted input is sent an acknowledgement and the others negative acknowledgements from that port. The separate signals for an input port are combined to give a single acknowledgement by a separate module ACKOR_N.

5.2.1 Acknowledgement Block Description

The behavioral specification for four output bits is given in four tables one for each bit. For the illustration purposes, we show specification in HOL for one output only which is given as:

For the output *ackOut*0:

```
TABLE [xGrant0; yGrant0; ackIn0; outputDisable0;
       xGrant1; yGrant1; ackIn1; outputDisable1;
       xGrant2; yGrant2; ackIn2; outputDisable2;
       xGrant3; yGrant3; ackIn3; outputDisable3 ](ackOut0)
[[TABLE_VAL F;TABLE_VAL F;TABLE_VAL T; TABLE_VAL F;
            DONT_CARE; DONT_CARE; DONT_CARE; DONT_CARE;
            DONT_CARE; DONT_CARE; DONT_CARE; DONT_CARE;
            DONT_CARE; DONT_CARE; DONT_CARE; DONT_CARE];
```

```
[DONT_CARE; DONT_CARE; DONT_CARE; DONT_CARE;
          TABLE_VAL F;TABLE_VAL F;TABLE_VAL T; TABLE_VAL F;
          DONT_CARE; DONT_CARE; DONT_CARE; DONT_CARE;
          DONT_CARE; DONT_CARE; DONT_CARE; DONT_CARE];
[DONT_CARE; DONT_CARE; DONT_CARE; DONT_CARE;
          DONT_CARE; DONT_CARE; DONT_CARE; DONT_CARE;
          TABLE_VAL F;TABLE_VAL F;TABLE_VAL T; TABLE_VAL F;
          DONT_CARE; DONT_CARE; DONT_CARE; DONT_CARE];
[DONT_CARE; DONT_CARE; DONT_CARE; DONT_CARE;
          DONT_CARE; DONT_CARE; DONT_CARE; DONT_CARE;
          DONT_CARE; DONT_CARE; DONT_CARE; DONT_CARE];
          TABLE_VAL F;TABLE_VAL F;TABLE_VAL T; TABLE_VAL F]];
[TSIG;TSIG;TSIG;TSIG] FSIG
```

where TABLE_VAL is defined as a new HOL type, and TSIG and FSIG are the lifted versions for the constants T and F which in turn are 1 and 0 respectively.

The equivalent MDG table specification of the Acknowledgement block generated using MDG_COMB_TAC for the output *ackOut*0 is as follows:

```
component(ackOut0 _spec, table([
  [ xGrant0 , yGrant0 , ackIn0 , outputDisable0 ,
      xGrant1 , yGrant1 , ackIn1 , outputDisable1 ,
      xGrant2 , yGrant2 , ackIn2 , outputDisable2 ,
      xGrant3 , yGrant3 , ackIn3 , outputDisable3 , ackOut0 ],
  [ 0, 0, 1, 0, *, *, *, *, *, *, *, *, *, *, *, *, 1 ],
  [ *, *, *, *, 0, 0, 1, 0, *, *, *, *, *, *, *, *, 1 ],
  [ *, *, *, *, *, *, *, *, 0, 0, 1, 0, *, *, *, *, 1 ],
  [ *, *, *, *, *, *, *, *, *, *, *, *, 0, 0, 1, 0, 1 ] | 0
  ])).
```

For understanding the specification, just the MDG tables (high-level specification) for the other three outputs ackOut1, ackOut2, ackOut3 are shown below.

For the output *ackOut*1:

```
[ 0, 1, 1, 0, *, *, *, *, *, *, *, *, *, *, *, *, 1 ],
[ *, *, *, *, 0, 1, 1, 0, *, *, *, *, *, *, *, *, 1 ],
[ *, *, *, *, *, *, *, *, 0, 1, 1, 0, *, *, *, *, 1 ],
[ *, *, *, *, *, *, *, *, *, *, *, *, 0, 1, 1, 0, 1 ] | 0
```

For the output *ackOut2*:

```
[ 1, 0, 1, 0, *, *, *, *, *, *, *, *, *, *, *, 1 ],
[ *, *, *, *, 1, 0, 1, 0, *, *, *, *, *, *, *, 1 ],
[ *, *, *, *, *, *, *, *, 1, 0, 1, 0, *, *, *, 1 ],
[ *, *, *, *, *, *, *, *, *, *, *, *, 1, 0, 1, 0, 1 ] | 0
```

For the output *ackOut3*:

```
[ 1, 1, 1, 0, *, *, *, *, *, *, *, *, *, *, *, 1 ],
[ *, *, *, *, 1, 1, 1, 0, *, *, *, *, *, *, *, 1 ],
[ *, *, *, *, *, *, *, *, 1, 1, 1, 0, *, *, *, 1 ],
[ *, *, *, *, *, *, *, *, *, *, *, *, 1, 1, 1, 0, 1 ] | 0
```

The implementation of the Acknowledgement block shown in Fig.5.6 described in HOL is trivial and is similar to the implementation in HOL of the Timing block which was shown in the sequential verification example. The resulting MDG-HDL implementation of the Acknowledgement block is generated by MDG_COMB_TAC.

Once the specification and implementation written in HOL are translated, MDG_COMB_TAC generates the required order file and algebraic specification file and calls the MDG tool for equivalence checking. The succeeded result from MDG is imported into HOL as a theorem. And hence the verification of the Acknowledgement block is done.

5.2.2 Acknowledgement Block Verification

We have shown that:

$$Ack\_Imp \equiv Ack\_Spec \text{ (equivalence)} \tag{5.10}$$

We got the above result from MDG and it is imported into HOL [35] as:

$$\vdash Ack\_Imp \Longrightarrow Ack\_Spec \tag{5.11}$$

We hence showed using MDG that the structural description (i.e. implementation) is equivalent to the high-level specification, described in terms of tables. Using our hybrid tool, the procedure is faster than proving in HOL that the implementation implies the high-level specification. The results for the combinational verification are shown in Table 5.2. The automatic translation to the MDG system proved effective in this case, reducing the specification and verification time significantly. Also, *equivalence* is a stronger result compared to *implication*.

MDG Nodes	CPU Time (sec.)	Memory (MB)
433,077	1426.51	163.98

Table 5.2: MDG Equivalence Checking Results for Acknowledgement Block

Chapter 6

Conclusions

Given that no single formal method is likely to be suitable for describing and analyzing every aspect of a complex system, a practical approach is to combine different methods. As explained in this thesis we combined theorem proving and equivalence checking methods.

To summarize the work, a linkage tool between HOL and MDG is built. It can be invoked by calling the functions MDG_COMB_TAC or MDG_SEQ_TAC from HOL. The tool uses the specification and implementation written in HOL in terms of MDG like tables and components respectively, generates all the required files (specification, implementation, algebraic specification, symbol order and invariant file) automatically, to be used in MDG. After the generation of the files, the MDG system is invoked where the MDG equivalence checking procedure is called and an appropriate theorem in HOL is generated in the positive case. By using this tool,

more complex circuits can be verified using the powerful induction and expressiveness of HOL and the automation of MDG with significant reduction of time avoiding cumbersome proof process of HOL as shown by our example. This is the main advantage of this hybrid tool. This hybrid approach is more effective in hierarchical verification. If the main module can be divided into smaller sub-modules, then certainly the use of this hybrid approach proves to be effective since there are less chances of *state explosion* problem and MDG can effectively handle smaller circuits.

The HOL-MDG interface implemented works for equivalence checking. The order, specification, implementation and the algebraic generators can be reused for extending this work to include model checking as part of the existing hybrid tool. The invariant file will be replaced by the property file and the MDG-HDL for the property file has to be generated.

Bibliography

[1] C. Seger, "An Introduction to Formal Hardware Verification," Tech. Rep. 92-13, Dept. of Computer Science, University of British Columbia, Vancouver, B.C., Canada, June 1992.

[2] A. Gupta, "Formal Hardware Verification Methods: A Survey," *Formal Methods in System Design*, vol. 1, no. 2/3, pp. 151–238, 1992.

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

[4] M. Gordon and T. Melham, *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, U.K., 1993.

[5] S. Owre, J. Rushby, and N. Shankar, "PVS: A Prototype Verification System," in *Proceedings of the 11th Conference on Automated Deduction (CADE'92)*, Lecture Notes in Computer Science 607, pp. 748–752, Springer Verlag, 1992.

[6] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas, "PVS: Combining Specification, Proof Checking, and Model Checking," in *Computer Aided Verification*, Lecture Notes in Computer Science 1102, (New Brunswick, NJ), pp. 411–414, Springer-Verlag, July 1996.

[7] R. Boyer and J. Moore, *A Computational Logic.* New York: Academic Press Inc., 1979.

[8] M. Kaufmann and J. Moore, "Design Goals for ACL2," Tech. Rep. 101, Computational Logic Inc., Austin, TX, 1994.

[9] M. Kaufmann and J. Moore, "ACL2: An Industrial Strength Version of Nqthm," in *Proceedings of the 11th Annual Conference on Computer Assurance (COMPASS'96, Gaithersburg, MD, June* (S. Faulk and C. Heitmayer, eds.), pp. 23–24, 1996.

[10] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking," in *Nato ASI*, vol. 152 of *F*, SpringerVerlag, 1996.

[11] E. Clarke, E. Emerson, and A. Sistla, "Automatic Verification of Finite-state Concurrent System Using Temporal Logic Specifications," *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 2, pp. 244–264, 1986.

[12] R. Bryant, "Graph-based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, pp. 677–691, August 1986.

[13] K. McMillan, *Symbolic Model Checking.* Boston, Massachusets: Kluwer Academic Publishers, 1993.

[14] L. Paulson, *ML for the Working Programmer.* U.K.: Cambridge University Press, 2nd ed., 1996.

[15] S. Rajan, N. Shankar, and M. Srivas, "An Integration of Model-checking with Automated Proof Checking," in *Computer Aided Verification* (P. Wolper, ed.), Lecture Notes in Computer Science 939, pp. 84–97, Springer Verlag, 1995.

[16] E. Emerson, *Temporal and Modal Logic, Handbook of Theoretical Computer Science.* Elsevier Sciences B.V. J. Van Leeuwn North Holland Edition, 1990.

[17] K. McMillan, *Getting Started with SMV; User's Manual.* Cadence Berkeley Laboratories, U.S.A., 1998.

[18] K. L. McMillan, "Verification of an Implementation of Tomasulo's Algorithm by Compositional Model Checking," in *Proc. Computer Aided Verification (CAV'98)*, (Vancouver, BC, Canada), June/July 1998.

[19] R. Brayton *et al*, "VIS," in *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)* (M. Srivas and A. Camilleri, eds.), Lecture Notes in Computer Science 1166, pp. 248–256, Springer Verlag, 1996.

[20] Cadence:, *Formal Verification Using Affirma FormalCheck; Manual, Version 2.3.* USA, October 1999.

[21] W. Thomas, *Automata on Infinite Objects*, vol. B of *Handbook of Theoretical Computer Science.* Amsterdam: Elsevier Science Publishers, 1990.

[22] F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny, "Multiway Decision Graphs for Automated Hardware Verification," *Formal Methods in System Design*, vol. 10, no. 1, pp. 7–46, 1997.

[23] W. Clocksin and C. Mellish, *Programming in Prolog.* Springer-Verlag, 1987. 3rd Edition.

[24] K. Schneider and T. Kropf, "Verifying Hardware Correctness by Combining Theorem Proving and Model Checking," Tech. Rep. SFB358-C2-5/95, University of Karlsruhe, Karlsruhe, Germany, December 1995.

[25] J. Joyce and C. Seger, "Linking BDD-based Symbolic Evaluation to Interactive Theorem Proving," in *Proceedings of the 30th International Conference on Design Automation (DAC'93)* (A. Dunlop, ed.), (Dallas, TX), pp. 469–474, ACM Press, June 1993.

[26] M. Gordon, "Combining Deductive Theorem Proving with Symbolic State Enumeration." 21 Years of Hardware Verification, December 1998. Royal Society Workshop to mark 21 years of BCS FACS.

[27] M. Aagaard, R. Jones, and C.-J. Seger, "Lifted-FL: A Pragmatic Implementation of Combined Model Checking and Theorem Proving," in *Theorem Proving in Higher Order Logics* (Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Theryvon, eds.), Lecture Notes in Computer Science 1690, pp. 323–340, Springer Verlag, September 1999.

[28] J. Hurd, "Integrating Gandalf and HOL," in *Theorem Proving in Higher Order Logics* (Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Theryvon, eds.), Lecture Notes in Computer Science 1690, pp. 311–321, Springer Verlag, September 1999.

[29] T. Tammet, "A Resolution Theorem Prover for Intuitionistic Logic," in *CADE 13*, Lecture Notes in Computer Science 1104, Springer Verlag, 1996.

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

[31] K. Schneider and D. Hoffmann, "A HOL Conversion for Translating Linear Time Temporal Logic to ω-Automata," in *Theorem Proving in Higher Order Logics* (Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Theryvon, eds.), Lecture Notes in Computer Science 1690, Springer Verlag, September 1999.

[32] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1991.

[33] V. Pisini, S. Tahar, O. Ait-Mohamed, P. Curzon, and X. Song, "An Approach to Link HOL and MDG for Hardware Verification." Proc. 1999 Micronet Annual Workshop, Ottawa, Canada, pp. 156-157, April 1999.

[34] V. Pisini, S. Tahar, P. Curzon, O. Ait-Mohamed, and X. Song, "Formal Hardware Verification by Integrating HOL and MDG," in *Proc. ACM 10th Great Lakes Symposium on VLSI (GLS-VLSI'00)*, (Chicago, Illinois, USA), pp. 23–28, ACM Publications, March 2000.

[35] H. Xiong, P. Curzon, and S. Tahar, "Importing MDG Results into HOL," in *Theorem Proving in Higher Order Logics* (Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Theryvon, eds.), Lecture Notes in Computer Science 1690, pp. 293–310, Springer Verlag, 1999.

[36] I. Leslie and D. McAuley, "Fairisle: An ATM Network for the Local Area," *ACM Communication Review*, vol. 19(4), pp. 327–336, 1991.

[37] P. Curzon, "The Formal Verification of the Fairisle ATM Switching Element," Technical Report 329, Computer Laboratory, University of Cambridge, U.K., March 1994.

[38] S. Tahar, X. Song, E. Cerny, Z. Zhou, M. Langevin, and O. Ait-Mohamed, "Modeling and Verification of the Fairisle ATM Switch Fabric using MDGs,"

IEEE Transactions on CAD of Integrated Circuits and Systems, vol. 18(7), pp. 956–972, July 1999.

[39] Y. Xu, E. Cerny, X. Song, F. Corella, and O. Ait-Mohamed, "Model Checking for a First-Order Temporal Logic using Multiway Decision Graphs," in *Computer Aided Verification* (A. Hu and M. Vardi, eds.), Lecture Notes in Computer Science 1427, pp. 219–231, Springer Verlag, 1998.

[40] M. Gordon, C. Wadsworth, and A. Milner, "Edinburgh LCF: A Mechanized Logic of Computation," Lecture Notes in Computer Science 78, Springer Verlag, 1979.

[41] S. Tahar and R. Kumar, "A Practical Methodology for the Formal Verification of RISC Processors," *Formal Methods in Systems Design*, vol. 13(2), pp. 159-225, September 1998. Kluwer Academic Publishers.

[42] P. Windley, "Formal Modeling and Verification of Microprocessors," in *IEEE Transactions on Computers*, vol. 44(1), pp. 54–72, January 1995.

[43] J. Joyce, *Multi-Level Verification of Microprocessor-Based Systems*. PhD thesis, Computer Laboratory, Cambridge University, U.K., December 1989.

[44] S. Tahar and P. Curzon, "Comparing HOL and MDG: A Case Study on the Verification of an ATM Switch Fabric," *Nordic Journal of Computing*, vol. 6, pp. 372–402, 1999.

[45] S. Tahar, P. Curzon, and J. Lu, "Three Approaches to Hardware Verification: HOL, MDG and VIS Compared," in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD'98)* (G. G. and W. P., eds.), Lecture Notes in Computer Science 1522, pp. 433–450, Springer Verlag, 1998.

[46] E. Cerny, F. Corella, M. Langevin, X. Song, S. Tahar, and Z. Zhou, "Automated Verification with Abstract State Machines Using Multiway Decision Graphs," in *Formal Hardware Verification: Methods and Systems in Comparison*, Lecture Notes in Computer Science 1287, State-of-the-Art Survey, pp. 79–113, Springer Verlag, 1997.

[47] Z. Zhou and N. Boulerice, *MDG Tools (V1.0) User's Manual.* Dept. of Computer Science, University of Montreal, Montreal, Canada, June 1996.

[48] Y. Xu, *MDG Model Checker User's Manual.* Dept. of Information and Operational Research, University of Montreal, Montreal, Canada, September 1999.

[49] Y. Xu, *Model Checking for a First-Order Temporal Logic Using Multiway Decision Graphs.* PhD thesis, Dept. of Information and Operational Research, University of Montreal, Montreal, April 1999.

[50] P. Curzon, S. Tahar, and O. Ait-Mohamed, "Verification of the MDG Components Library in HOL," in *Theorem Proving in Higher Order Logics: Emerging Trends* (J. Grundy and M. Newey, eds.), (Australian National University),

pp. 31–45, September 1998.

[51] M. Gordon, "Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware," in *Formal Aspects of VLSI Design: Proceedings of the 1985 Edinburgh Workshop on VLSI* (G. Milne and P. Subrahmanyam, eds.), pp. 153–177, North-Holland, 1986.