# A Hybrid Approach to Formal Verification Using HOL and MDG

V.K. Pisini[1], S. Tahar[1], P. Curzon[2], O. Ait-Mohamed[3] and X. Song[4]

**Technical Report**

[1]Dept. of Elec. & Comp. Eng., Concordia University, Montreal, Canada
E-mail: {pisini,tahar}@ece.concordia.ca

[2]School of Computing Science, Middlesex University, London, UK
E-mail: p.curzon@mdx.ac.uk

[3]Nortel Networks, Ottawa, Canada
E-mail: otmanem@nortelnetworks.com

[4]Dept. of IRO., University of Montreal, Montreal, Canada
E-mail: song@iro.umontreal.ca

November 1999

**Abstract**

*In order to overcome the limitations of automated tools and the cumbersome proof process of interactive theorem proving, we adopt a hybrid approach for formal hardware verification which uses the strengths of theorem proving (HOL) with powerful mathematical tools such as induction and abstraction, and the advantages of automated tools (MDG) which support equivalence checking and model checking. The MDG system is a decision diagram based verification tool, primarily designed for hardware verification. HOL is a theorem prover built on higher-order logic. The methodology used to link the tools and the functioning of the interface are described in detail. We use the timing block of the 4 by 4 Fairisle ATM switch fabric to illustrate the verification using this hybrid tool.*

# Contents

# 1 Introduction

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. Simulation, which is the state-o-the-art is often used as the main approach for verification, and despite the major simulation efforts, serious design errors often remain undetected which resulted in the evolution of applications such as formal methods in verifying the hardware design. There are several approaches to formal hardware verification: theorem-proving, model checking, equivalence checking, symbolic simulation to name a few [18]. Each of them has its own strengths and weaknesses. In this report, we present a methodology with an example as to how equivalence checking of the automated MDG system [4] supports the proof process of the HOL theorem prover [9]. HOL is an interactive system that is built on higher-order logic and developed at the University of Cambridge, U.K. Theorem proving 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. Integration of interactive and automated tools eases the verification complexity to a great extent.

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 which extend ROBDDs [2] with abstract sorts and uninterpreted function symbols. 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. Higher-order logic is very flexible and has a well-defined and well-understood semantics. It also allows us to use hierarchical verification wherein the modules are divided into sub-modules and even the sub-modules are divided until the lowest level (gate 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 MDG as illustrated by related work on microprocessor verification [19, 22, 11]. This is beyond the capabilities of the MDG on its own. 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 is very time-consuming. Combining both systems reaps the advantages of both. In our hybrid approach, we verify the sub-modules using the MDG system.

The remaining sections of this report contain the related work and describe the methodology we used in this hybrid approach. Section 3 describes the MDG and HOL systems. The MDG system is described in detail for the reader to understand the linking approach which is the main contribution of our work. In Section 4 we describe our hybrid approach and how it is embedded inside the logic of an interactive theorem-prover. In Section 5 we present an example we considered, the Timing block of the Fairisle ATM switch fabric, through which we illustrate the advantages of our approach. Section 6 finally concludes the paper.

# 2 Related Work

There exist a number of hybrid approaches such as combining theorem proving with model checking [8, 10, 15, 16, 17] and combining theorem proving and symbolic trajectory evaluation [1, 12]. Joyce and Seger [12] 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 which makes 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. The authors implemented a tactic called VOSS_TAC which calls the Voss system and 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 procedure.

Aagaard *et al* [1] constructed a system that integrates symbolic trajectroy evaluation based model checking with theorem proving in a higher-order classical logic. The approach is made possible by using the same programming language $fl$ as both the meta and object language of theorem proving. This is done by "lifting" $fl$, 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*". This approach is applicable to any dialect of the ML programming language and any model-checking algorithm that has inference rules for combining results.

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

Schneider *et al* [17] 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.

Hurd [10] described GANDALF_TAC, a HOL tactic that proves goals by calling Gandalf 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. This call can occur over a network, and a Gandalf server may be set up servicing multiple HOL clients. GANDALF_TAC is a Prosper plug-in [7] which 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.

Schneider and Hoffmann [16] described the embedding of linear time temporal logic LTL in HOL together with a translation of LTL formulae into equivalent $\omega$-automata [21]. The translation is implemented by HOL rules. 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 HOL users to directly verify temporal properties by means of HOL's induction rules.

Gordon [8] described the integration of the BuDDy BDD package in HOL. 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.

More work is still being done to integrate the formal verification tools. Since theorem proving approach has the flexibility to express the behavior of the circuit at different levels 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 technique is mostly used to integrate with theorem provers because the model checking tools offer complete automation.

In difference to related work, in this report we combine theorem proving with automated equivalence checking. We proposed a methodology as to how equivalence checking of the automated MDG system [4] supports the proof process of the HOL theorem prover [9]. 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 [23] to HOL.

The motivation to take up this work originated while looking into ways to integrate VHDL and formal verification. The work described in this report is part of a larger project to link VHDL, HOL and MDG as shown in Figure 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 the functions, MDG_COMB_TAC and MDG_SEQ_TAC, to generate the required files for the MDG system to complete the verification for combinational and sequential verifications respectively. In the case of property verification, an LTL property description (L_MDG) [24] 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.

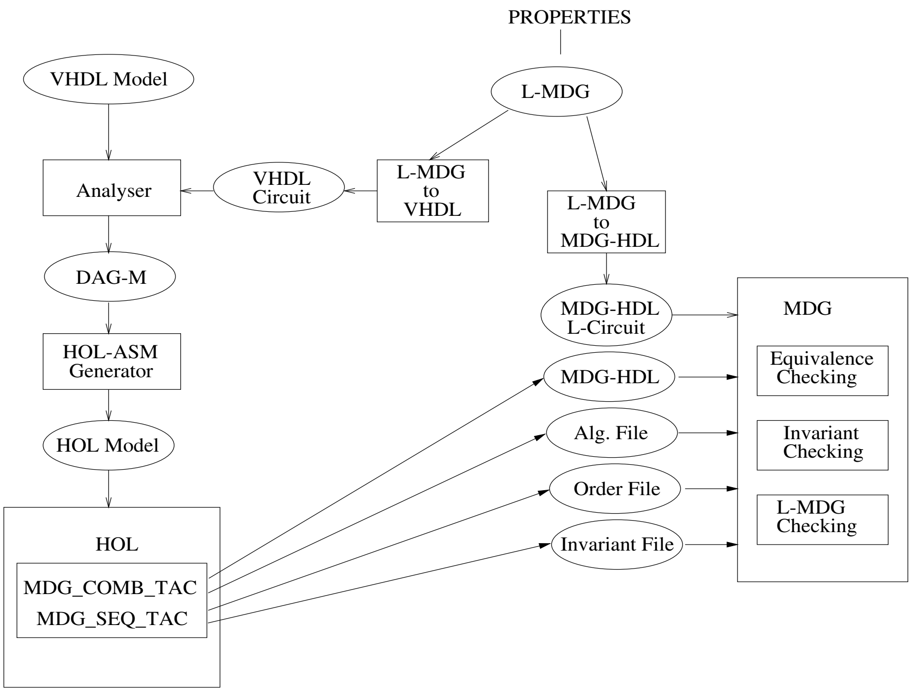


Figure 1: Intended VHDL-HOL-MDG Project Skeleton

# 3 HOL and MDG

## 3.1 HOL System

The HOL System is a theorem prover based on higher-order logic [9] 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. 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.

The basic interface to the system is a Standard ML (SML) 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. It is very flexible and supports forward and backward proof by creating theorems and applying inference rules to the already created theorems. In the backward proof, the user sets the desired theorem as a goal.

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. The 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 and an online help facility. The applications of the HOL system are hardware design and 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.

## 3.2 MDG System

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 [3]. A fundamental primitive of its hardware description language is the table which is the truth table representation of a relation between the values on variables. 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) [4].

Multiway Decision Graphs (MDGs) have been proposed [4] as a solution to the data width problem of ROBDD based verification tools. The MDG tool combines the advantages of representing a circuit at higher abstract levels as is possible in a theorem prover, and of the automation offered by ROBDD based tools. An MDG is a finite, directed acyclic graph (DAG). MDGs essentially represent relations rather than functions. MDGs can also represent sets of states. 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. The MDG tools package the basic MDG operators and verification procedures [25]. 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 [25].

MDG-HDL which is the input language for MDG, supports structural descriptions, behavioral ASM descriptions or a mixture of both. A structural description is usually a netlist of components connected by signals, and a behavioral description is given by a tabular representation of the transition/output relation or truth table. The MDG-HDL comes with a large library of predefined, commonly used, basic components (such as logic gates, multiplexers, registers, bus drivers, ROMs, etc.). 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 [0,1,2,3] as an enumeration, inputs: x1, x2, x3 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 *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 are: a behavioral specification file, a circuit description file, an algebraic file, a symbol order file, and an invariant file [25]. The behavioral specification file declares signals and specifies the behavior of the circuit using tables as dexcribed above. 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 circuit description file declares signals and their sort assignments, describes the circuit network and the mapping between state variables and the next state variables. The invariant file takes the corresponding circuit outputs from both behavioral specification file and circuit description file and joins them to enable the MDG system to check for equivalence between those outputs.

# 4 Linking Approach

## 4.1 Hierarchical Verification

In our hybrid approach, we follow a hierarchical hardware verification methodology. 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. 2. The sub-modules are repeatedly subdivided until eventually the logic gate level is reached.

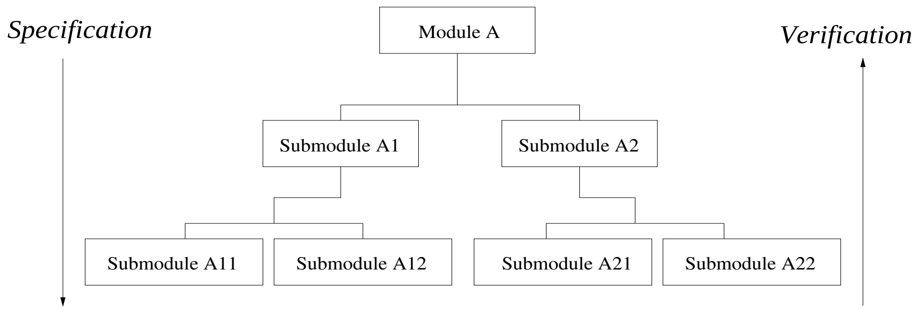


Figure 2: Hierarchical Verification

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

$$\vdash Imp\_A \Longrightarrow Spec\_A \quad (4.1)$$

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 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

$$\vdash Imp\_A = Imp\_A1 \wedge Imp\_A2 \quad (4.2)$$

Now (4.1) can be written as

$$\vdash Imp\_A1 \wedge Imp\_A2 \Longrightarrow Spec\_A \quad (4.3)$$

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

$$\vdash Imp\_A1 \Longrightarrow Spec\_A1 \quad (4.4)$$

$$\vdash Imp\_A2 \Longrightarrow Spec\_A2 \qquad (4.5)$$

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

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

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 our work is to identify and prove the correctness of certain sub-modules in an automatic fashion using the MDG system. In MDG, for each sub-module it will be proved by automatic verification that implementation is equivalent to specification and the result is imported into HOL. In our hybrid system, the sub-module is treated as a black-box.

## 4.2 Linking HOL and MDG

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 [5]. 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 MDG acceptable form using the description predicates. In case a sub-goal is not 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 the main goal is proved and hence the circuit is formally verified. The interface converts the HOL descriptions to equivalent MDG files and all required files for the MDG verification as specified in the following.

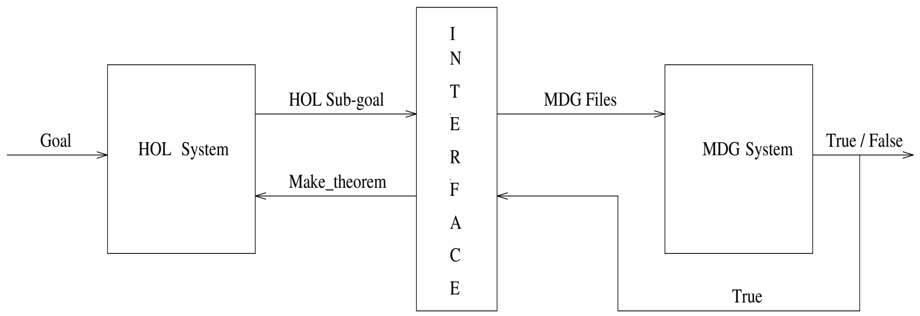


Figure 3: Block Diagram of the Hybrid System

The sub-goal specification and implementation which are in two different 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. If there are new user defined types, they are usually defined in the HOL specification file. From the given two HOL files, corresponding MDG circuit description file, MDG specification file, MDG algebraic specification file, MDG order file and MDG invariant file are created automatically. These MDG files are used for verification by the MDG system for equivalence checking. In the case where the equivalence checking has succeeded, MDG returns "true" and this result is imported into HOL in the form of a theorem (using the *make_theorem* in HOL) and the main proof procedure continues in HOL with the next sub-goal to be proved. Xiong *et al* [23] showed how the results of MDG can be imported into HOL. [23] provides a formal proof for the soundness of imported verification results from MDG to HOL. As part of the build-up of the mathematical interface between the two tools, the total MDG library was specified in HOL as predicates and Curzon *et al* [5] formally verified the MDG component library in HOL. The authors also showed how the MDG tables can be expressed in HOL.

We developed two SML functions (tactics), MDG_COMB_TAC (for combinational verification) or MDG_SEQ_TAC (for sequential verification) which link HOL and MDG for hybrid verification. The tasks of MDG_COMB_TAC/MDG_SEQ_TAC are shown as a flow-diagram in Fig. 4.

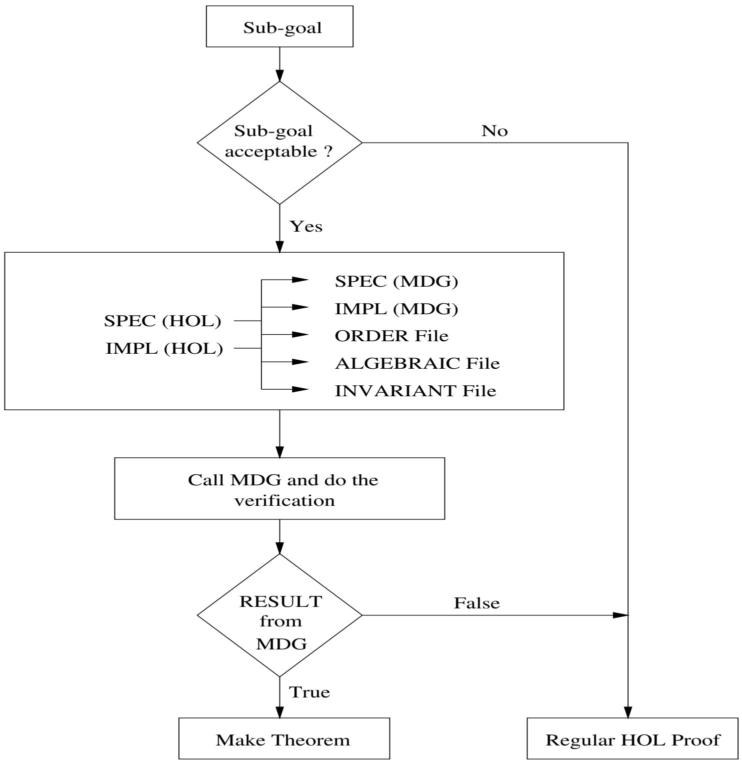


Figure 4: Task of MDG_COMB_TAC/MDG_SEQ_TAC

Fig. 5 explains the internal structure of the hybrid tool in detail. First, one of the tactics invokes the translation, and once the translation is done, the verification begins and the obtained validation result is imported into HOL for further verification proof process. All file generators (Algebraic, Order, Invariant, Specification, Implementation) shown in Fig. 5 are associated each to a specific generator in the translator code.

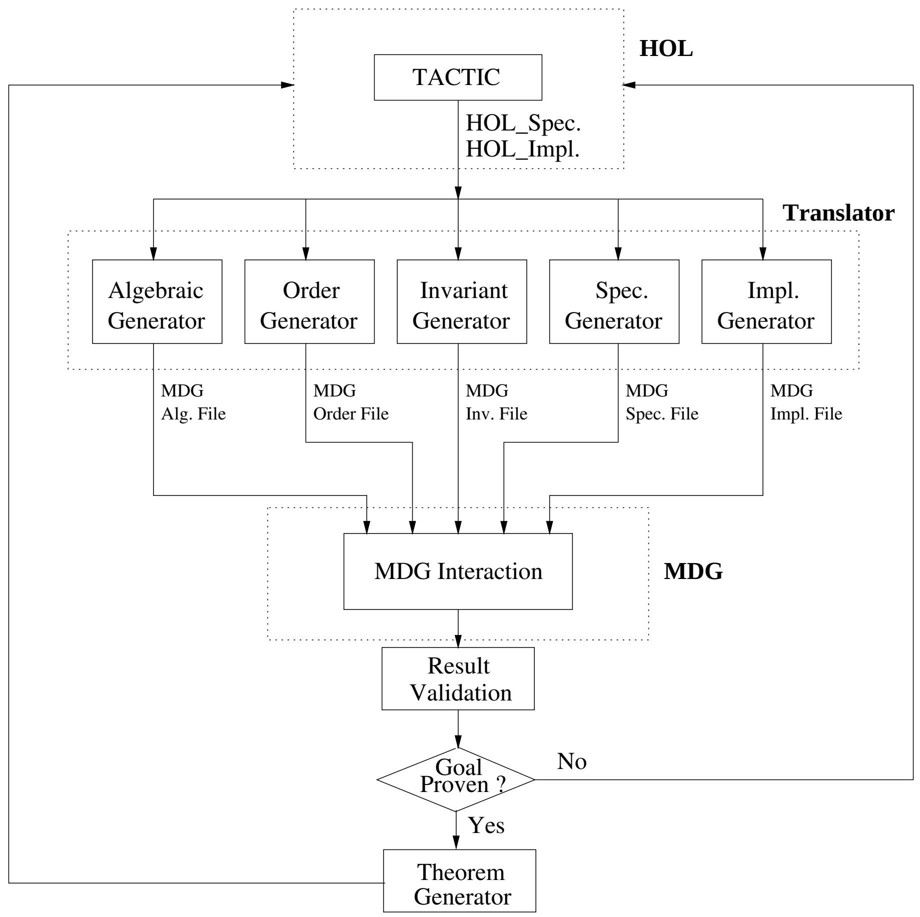


Figure 5: Internal Structure of the Hybrid Tool

# 5 Case Study

For illustration purposes, we show the verification of a sub-module of the Fairisle ATM switch fabric [13] (see Fig. 6) using our hybrid approach. Curzon [6] formally verified this ATM switching element using the theorem-prover HOL. Tahar *et al* [20] reverified it using MDG. 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. 6). In our example, we have taken the Timing block to be a sub-sub-module (one of the sub-goals) and used our hybrid tool to achieve the desired verification.

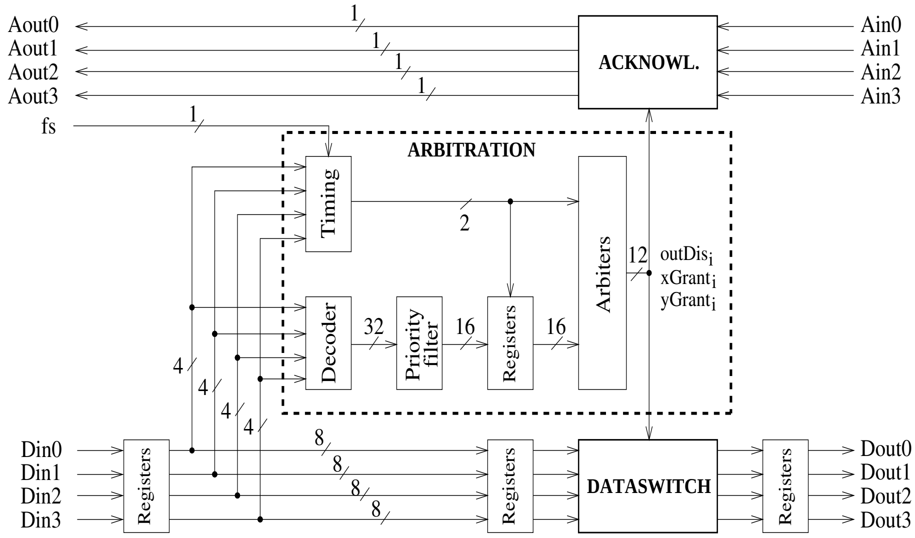


Figure 6: Fairisle ATM Switch Fabric

## 5.1 Proof Structure of the ATM 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. 7.

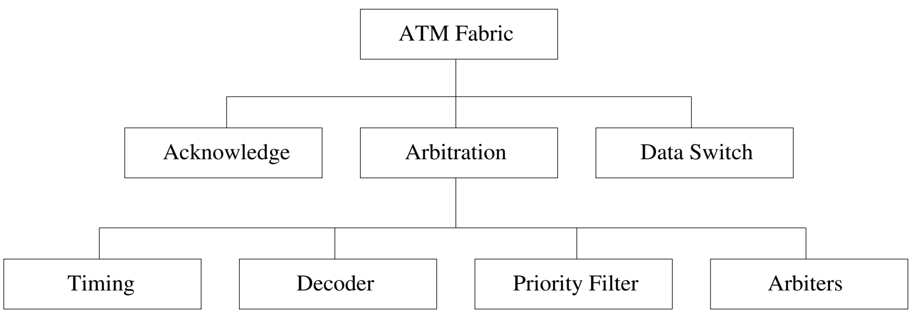


Figure 7: Hierarchical Verification of the Fairisle Switch Fabric

The goal is to prove that

$$\vdash Fabric\_Imp \Longrightarrow Fabric\_Spec \tag{5.1}$$

From Fig. 7 and the equations in Section 4.1, we have

$$\vdash Fabric\_Imp = Ack\_Imp \wedge Arb\_Imp \wedge DataSw\_Imp \tag{5.2}$$

as in (4.4) and (4.5), 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.

## 5.2 Timing Block Verification

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. 8 and the FSM representation is shown in Fig. 9.

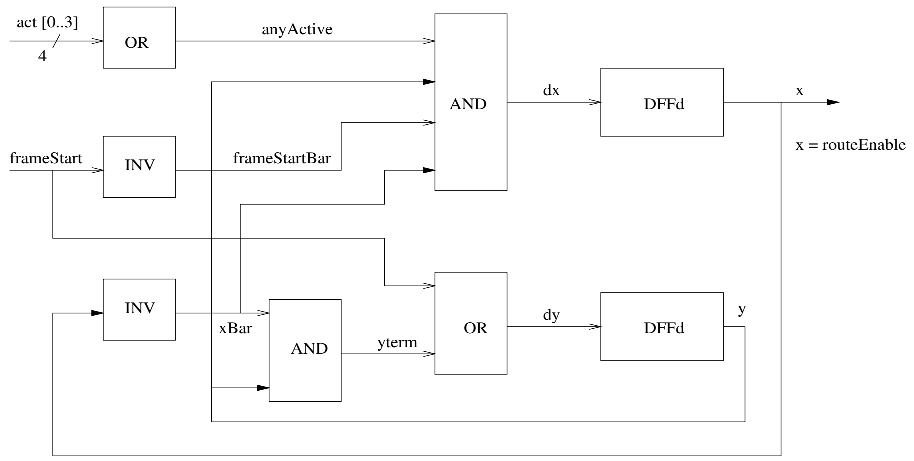


Figure 8: Implementation of the Timing Block

The implementation of the Timing block shown in Fig. 8 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)
```

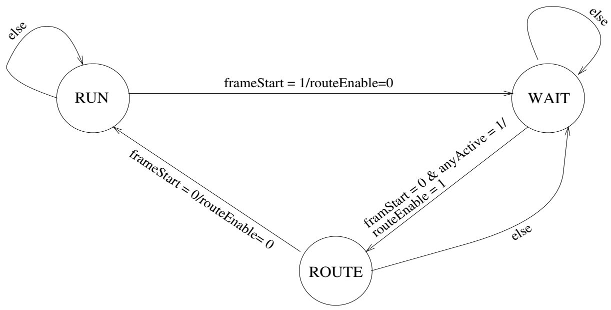


Figure 9: State Transitions of the Timing Block

The resulting MDG-HDL implementation of the Timing block 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))).
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 output function. The HOL definition of the state transitions of the FSM in Fig. 9 which is written in terms of the table specification is given as:

```
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
```

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 in HOL are translated to MDG-HDL, 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.

## 5.3 Verification Results

We have shown that:

$$Timing\_Imp \equiv Timing\_Spec \text{ (equivalence)} \qquad (6.1)$$

We got the above result from MDG and it is imported into HOL as [23]:

$$\vdash Timing\_Imp \Longrightarrow Timing\_Spec \qquad (6.2)$$

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

$$\vdash Timing\_Spec \wedge Decoder\_Spec \wedge PFilter\_Spec \wedge Arbiters\_Spec \Longrightarrow Arb\_Spec \quad (6.3)$$

Hence proving the higher-level subgoal for the whole Arbitration block.

We showed using MDG that the structural description (i.e. implementation) is equivalent to a high level specification specified using tables. Writing the high-level specification (behavioral 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 [6].

Using our hybrid tool, the procedure is faster than proving in HOL that the implementation implies the high-level specification. Curzon [6] took several hours to do the proof of the Timing block whereas the verification is done in less than a second in MDG (see Table 1). It took six man hours to write the specification and implementation files in HOL. The automatic translation to MDG proved effective in this case. 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*.

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

Table 1: MDG Equivalence Checking Results for Timing Block

# 6 Conclusions

To summarize our work, we have built a tool linking HOL and MDG. It can be invoked by calling the functions MDG_COMB_TAC or MDG_SEQ_TAC from HOL. A translator uses the specification and implementation written in HOL in terms of MDG like tables and components respectively, generates all the required files (specification and implementation files, algebraic specification file, symbol order file and invariant file automatically, to be used in MDG. After the generation of the files the MDG system is invoked for the verification. By using the automated MDG system, the total verification time is significantly reduced. This is the main advantage of this hybrid tool. The example we have chosen neatly fits into the proof structure.

By using this tool, more complex circuits can be verified using the powerful induction and expressiveness of HOL and the automation of MDG. 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 translator we have implemented works for equivalence checking of MDG. This work can be extended to use model checking as part of the hybrid tool.

# Acknowledgments

This work was partially supported by a Micronet research grant and a GRIAO student scholarship. Thanks are due to Dr. E. Cerny and Dr. A. Dekdouk at IRO, University of Montreal for initiating this work, and to Dr. I. Kort and Mr. M.H. Zobair at ECE, Concordia University who helped us out with HOL and MDG proofs.

# References

[1] M.D. Aagaard, R.B. Jones, and C-J.H. Seger. Lifted-FL: A Pragmatic Implementation of Combined Model Checking and Theorem Proving. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Theryvon, editors, *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science 1690, pages 323–340. Springer Verlag, September 1999.

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

[3] 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, pages 79–113. Springer Verlag, 1997.

[4] F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny. Multiway Decision Graphs for Automated Hardware Verification. *Formal Methods in System Design*, 10(1):7–46, 1997.

[5] P. Curzon, S. Tahar, and O. Ait-Mohamed. Verification of the MDG Components Library in HOL. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics: Emerging Trends*, pages 31–45, Australian National University, September 1998.

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

[7] L. Dennis, G. Collins, and M. Norrish. *The PROSPER Toolkit.* University of Glasgow, 1999-2000.

[8] M.J.C. 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.

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

[10] J. Hurd. Integrating Gandalf and HOL. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Theryvon, editors, *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science 1690, pages 311–321. Springer Verlag, September 1999.

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

[12] J.J. Joyce and C.J.H. Seger. Linking BDD-based Symbolic Evaluation to Interactive Theorem Proving. In A.E. Dunlop, editor, *Proceedings of the 30th International Conference on Design Automation (DAC'93)*, pages 469–474, Dallas, TX, June 1993.

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

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

[15] S. Rajan, N. Shankar, and M.K. Srivas. An Integration of Model-checking with Automated Proof Checking. In Pierre Wolper, editor, *Computer Aided Verification*, Lecture Notes in Computer Science 939, pages 84–97. Springer Verlag, 1995.

[16] K. Schneider and D.W. Hoffmann. A HOL Conversion for Translating Linear Time Temporal Logic to $\omega$-Automata. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Theryvon, editors, *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science 1690. Springer Verlag, September 1999.

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

[18] C.J. Seger. An Introduction to Formal Hardware Verification. Technical Report 92-13, Dept. of Computer Science, University of British Columbia, Vancouver, B.C., Canada, June 1992.

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

[20] S. Tahar, X. Song, E. Cerny, M. Langevin Z. Zhou, and O. Ait-Mohamed. Modeling and Verification of the Fairisle ATM Switch Fabric using MDGs. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 18(7):956–972, July 1999.

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

[22] P. Windley. Formal Modeling and Verification of Microprocessors. *IEEE Transactions on Computers*, 44(1), January 1995.

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

[24] 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 A. Hu and M. Vardi, editors, *Computer Aided Verification*, Lecture Notes in Computer Science 1427, pages 219–231. Springer Verlag, 1998.

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