# Verification of the MDG Components Library in HOL

Paul Curzon[1], Sofiène Tahar[2], and Otmane Aït Mohamed[3]

[1] School of Computing Science, Middlesex University, London, UK
p.curzon@mdx.ac.uk
[2] ECE Department, Concordia University, Montreal, Canada
tahar@ece.concordia.ca
[3] IRO Department, University of Montreal, Canada
ait@iro.umontreal.ca

**Abstract.** The MDG system is a decision diagram based verification tool, primarily designed for hardware verification. It is based on Multiway decision diagrams –an extension of the traditional ROBDD approach. In this paper we describe the formal verification of the component library of the MDG system, using HOL. The hardware component library, whilst relatively simple, has been a source of errors in an earlier developmental version of the MDG system. Thus verifying these aspects is of real utility towards the verification of a decision digram based verification system. This work demonstrates how machine assisted proof can be of practical utility when applied to a small focused problem.

## 1 Introduction

Verification systems can themselves contain errors. In the worst case this could result in a faulty application being certified correct. Ideally verification systems should themselves be formally verified: preferably using a verification system with a different architecture. In general, this is not practical, as verification systems are very large pieces of software. However, it can still be useful to verify aspects of the system, even if a full verification is not completed. In this paper we investigate the verification of the components library of a decision diagram based verification system using the HOL theorem prover [11]. The verification system under investigation is the MDG system [5]. This is a real hardware verification system that has been used in the verification of significant hardware examples [2]. It consists of a simple wide-spectrum hardware description language (MDG-HDL) in which both structural and behavioral hardware descriptions can be written. These descriptions are converted to an internal decision diagram representation, upon which the verification is performed. A fundamental primitive of the hardware description language is the table. In its simplest form this is just a 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) [5].

Tables are also used internally in the MDG implementation. They provide a simple and uniform means of implementing other primitive components. The current implementation of the MDG system provides a library of basic components in addition to the table with which hardware can be described. Examples include flip-flops and logic gates. Many of these primitives are implemented internally as tables. We have verified this library of components, proving that the table versions implemented in the MDG system are equivalent to the desired semantics of the components as specified in higher-order logic.

The library is only a small part of the MDG system. However, it is critically important that the components are correctly implemented. The MDG system provides a range of verification tools, including property checking, equivalence checking and reachability analysis. Each of these make use of the library primitives. For example, properties which abstractly can be thought of as temporal logic formulae are written in (or translated to) the HDL and thus make use of the library components.

One of the motivations for our work was an error in a table representation of one component, the *JK flip-flop with enable*. This error was discovered in a developmental version of the system, found during the actual verification of a hardware design [12]. The system was erroneously indicating there was an error in the design being verified. The erroneous component had only recently been added to the system specifically because it was needed for the verification of the hardware design [12] (only a JK flip-flop variant *without enable* was available within the library). This error was corrected in the system prior to our work. We have demonstrated that the new version is correct and that the other components implemented as tables are also correct. Furthermore, we have provided precise formal specifications of each library component. Finally we have provided simple parameterized HOL tactics which can be used to automatically verify future additions to the library.

## 2 Related Work

There has been a variety of techniques used to ensure the correctness of verification systems. In the LCF approach [9], also used in the HOL system, an abstract data type of theorem is used to ensure that only a core of functions corresponding to the primitive inference rules and axioms of the logic can compromise the system. All derived rules call these primitives to create theorems. Thus the validity of proved theorems is guaranteed by the type system of the implementation language, provided the primitives are correct.

A second approach experimented with in the HOL system by von Wright [13] and Wong [14], was that of independent proof checking. In this approach, the main verification system produces a log of the primitive inference steps used in a proof. This log can then be checked by an independent proof checker. Such a checker has to include only implementations of the primitive rules. It can thus be much simpler than a full theorem prover and is thus less likely to contain errors. Such a proof checker has been implemented for the HOL system by Wong [14].

Due to its simplicity, verifying such a proof checker is also more tractable. Von Wright demonstrated this by verifying the specification of a proof checker for the HOL system against a formal semantics of the HOL logic [13]. This specification was also used by Wong as the specification for his implementation, thus increasing the confidence in its correctness. A problem with this approach, however, is that the proof scripts generated are very large and the time taken to check a real proof may be intractable.

Other work on the verification of verification systems includes that of Homeier and Martin [8] who used the HOL system to verify a verification condition generator for a simple programming language. Chou and Peled [4] similarly used HOL to verify a partial-order reduction technique used to reduce the state-space exploration performed by model checkers. The technique examined is used in the SPIN system. This was a significant proof effort, resulting in almost 7500 lines of proof script and taking 10 weeks to complete.

## 3 MDG System

### 3.1 Multiway Decision Graphs

Multiway Decision Graphs (MDGs) have been proposed recently [5] 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. MDGs, a new class of decision graphs, comprises, but is much broader than, the class of ROBDDs [1]. It is based on a subset of many-sorted first-order logic, augmented with a distinction between abstract and concrete sorts. Concrete sorts have enumerations which are sets of individual constants, while abstract sorts do not. Variables of concrete sorts are used for representing control signals, and variables of abstract sorts are used for representing datapath signals. Data operations are represented by uninterpreted function symbols.

An MDG is a finite, directed acyclic graph (DAG). An internal node of an MDG can be a variable of a concrete sort with its edge labels the *individual constants* in the enumeration of the sort. It can also be a variable of abstract sort with its edges labeled by abstract terms of the same sort. Finally, it can be a *cross-term* (whose function symbol is a cross-operator). An MDG may only have one leaf node denoted as T, which means all paths in the MDG are true formulae. Thus, MDGs essentially represent relations rather than functions. MDGs incorporate variables of abstract type to denote data signals and uninterpreted function symbols to denote data operations. MDGs can also represent sets of states. They are thus much more compact than ROBDDs for designs containing a datapath. Furthermore, sequential circuits can be verified independently of the width of the datapath.

MDGs are used as the underlying representation for a set of hardware verification tools, providing both validity checking and verification based on state-space exploration. The MDG tools package the basic MDG operators and verification procedures [16]. The operators are *disjunction*, *relational product* (*conjunction* followed by *existential quantification*) and *pruning-by-subsumption*. 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 [16].

### 3.2 MDG-HDL

The MDG tools accept as hardware description a Prolog-style HDL, MDG-HDL [16], which allows the use of abstract variables and uninterpreted function symbols. The MDG-HDL description is then compiled into the internal MDG data structures. MDG-HDL supports structural descriptions, behavioral descriptions, or a mixture of structural and behavioral descriptions. A structural description is usually a netlist of components (predefined in MDG-HDL) connected by signals. A behavioral description is given by a tabular representation of the transition/output relation. The tabular constructor is similar to a truth table but allows first-order terms in rows. It allows the description of high-level constructs as ITE (If-Then-Else) formulas and CASE formulas.

A circuit description includes the definition of signals, components and the circuit outputs. Signals are declared along with their sorts, e.g. $signal(x, wordn)$, where $x$ is a signal of an abstract sort *wordn*. 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: $x0, x1, x2, x3$ of an abstract sort $\alpha$, and output: $y$ of the same abstract sort $\alpha$ is defined as:

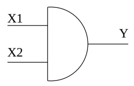
```
component(mux1,mux(sel(select),
                   inputs([(0,x0),(1,x1),(2,x2),(3,x3)]),
                   output(y))
```

Besides circuit descriptions, a variety of information, such as sort and function type definitions, symbol ordering and invariant specification, etc., have to be provided in order to use the applications outlined above.

As part of the MDG software package, the user is provided with a large set of predefined modules such as logic gates, multiplexers, registers, bus drivers, etc. Besides the logic gates which use Boolean signals, all other components allow signals with concrete as well as abstract types. 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 similar to the truth table, but it allows first-order terms in the rows. 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 remaining lists consist of the sets of values that the corresponding variables or cross-terms can take. The last element in the list of values could be a first-order term. This represents an assignment to the output variable. The other values must be either "don't cares" (represented by '*') or individual constants in the enumeration of their corresponding variable sort. The last element in a table is the default value. It is a term giving the value of the output variable when a set of values arises that is not explicitly given in the table. Fig. 1 illustrates different representation of an **and** gate with two inputs $x1, x2$ and one output $y$. Fig. 1(b) shows the MDG-HDL declaration of this gate using the primitive component **and**. The behavior of this gate can be described as a table (Fig. 1(c)) which can be written in MDG-HDL as follows:

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

This table description is further internally translated into an MDG (decision diagram) with the variable ordering $x1 < x2 < y$ (Fig. 1(d)).



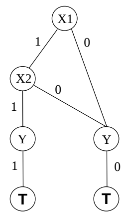

(a) Gate

```
component(
  and_gate,
  and (input(x1,x2),
       output(y))).
```

(b) MDG-HDL

| X1 | X2 | Y |
|---|---|---|
| 0 | * | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

(c) Table




(d) MDG

**Fig. 1.** Different representations of an **and** gate

A further example of the use of tables with abstract variables and functions is given by the following example:

```
table([[c,leq(x,y),n_y],[1,1,x]|y])
```

which defines the function

$$if\ (c = 1)\ and\ (leq(x, y) = 1)\ then\ n\_y = x\ else\ n\_y = y.$$

where $c$ is a concrete boolean variable, $x$ is an abstract input variable, $y$ is an abstract state variable, and $n\_y$ represents its next state. $leq$ represents a function symbol that means "less-or-equal". The term $y$ after symbol '|' in the table description is used as the default value.

## 4 Formalizing the MDG Library in HOL

The first step in the verification is to give formal specifications of the library components to be verified. This is a relatively simple task, since the components are mainly logic gates and flip-flops. Traditional relational hardware semantics in the style of Gordon [10] can by given. Signals are represented as functions from time (a natural number) to the value at that time. The semantics of a component is then a relation between the input signals and the output signals. For example, the **and** gate would be specified as:

```
AND x1 x2 y = ∀(t:num). y t = (x1 t) ∧ (x2 t)
```

Here $y$ is the output signal and $x_1$ and $x_2$ are the input signals. Similar specifications are given for each component in the library to be verified, as well as for tables. The definition for tables is more complex, requiring recursive definitions.

### 4.1 MDG-Tables

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.

Thus for example the **and** gate, specified above could be represented by the arguments:

```
([x1, x2], y, [[0,0],[0,1],[1,0],[1,1]], [0, 0, 0, 1], -)
```

The inputs are $x_1$ and $x_2$, the output is $y$, the possible values for the inputs are $(0,0),(0,1),(1,0)$ and $(1,1)$. The corresponding values on the output are $0,0,0$ and 1, respectively. Here no default value is needed as all cases are covered.
An alternative version, making use of the don't-care value (given by $*$), is

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

A more compact version still using the default value would be:

```
([x1, x2], y, [[1,1]], [1], 0)
```

If both inputs are 1 then so is the output, otherwise the output is 0. Similarly, the MDG system's implementation of a **JK** flip flop with enable is the table:

```
([e, j, k, q], nq,
     [[0,*,*,0],
      [0,*,*,1],
      [1,1,0,0],
      [1,1,0,1],
      [1,0,1,0],
      [1,0,1,1],
      [1,0,0,0],
      [1,0,0,1],
      [1,1,1,0],
      [1,1,1,1]],
     [0,1,1,1,0,0,0,1,1,0], -)
```

Here, $e$ is the enable signal, $q$ represents the last output and $nq$ the next output.

Our HOL specifications are based on the above representation. In fact the implementation which is in Prolog, uses a slightly different representation, taking a single list of list argument and a further default value. The first version of the **and** gate above actually appears in the implementation as the following (with no default specified).

```
[[x1, x2, y], [0,0,0],[0,1,0],[1,0,0],[1,1,1]]
```

Here, the inputs and output appear in a single list, with the latter distinguished by its position. In our description above and our HOL treatment we have separated out the components for clarity of definition. It should be noted that the above representation could not be used in our HOL treatment given below, as the lists have different types: values as opposed to traces of values (including don't-care) over time. It is thus possible that we could have made transcription mistakes from one form to the other. However, it would be relatively simple to modify our table definition to use two arguments: a variable list and a row list in the same order as in the MDG implementation. This would merely involve adding a wrapper function to the TABLE definition, which extracted the appropriate arguments.

### 4.2 Table Formalization in HOL

The first step in formalizing this definition is to define a type for table values. These can be either a normal value of arbitrary type or a don't-care value. This is defined as a new HOL type, with associated destructor function to access the value.

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

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

We next define the matching of input values to table values. A match occurs if either the table value is don't-care, or the value on the input is identical to the table value. This property must hold for each table entry. It is defined recursively by a function *table_match*.

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

(Table_match inputs (CONS v vs) t =
      ( ((HD(inputs) t) = TableVal_to_Val (v:'a Table_Val) ) ∨
         (v = DONT_CARE)) ∧
      (Table_match (TL inputs) vs t) )
```

If there is a match on a given row, the output has the corresponding value. Otherwise, we must check the next row. If there is no match, the output equals the default value. This is defined recursively on the input list as the relation table:

```
(table inps (out:num -> 'b) ([]:('a Table_Val list) list) V_out default t =
      (out t = default t) ) ∧

(table inps out (CONS v vs) V_out default t =
      ((Table_match inps v t) =>
       (out t = (HD V_out)t ) |
       (table inps out vs (TL V_out) default t)))
```

The above definitions refer to the time of interest, $t$. A given table will relate a given input to a given output, if the table relation is true at all times:

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

The above relation TABLE, thus defines the semantics of an MDG table. Using the HOL notation the Table for the AND component would be specified as:

```
AND_TABLE x1 x2 y =
      TABLE [(x1:num->bool);x2](y:num -> bool)
       [[TABLE_VAL F; TABLE_VAL F];
        [TABLE_VAL F; TABLE_VAL T];
        [TABLE_VAL T; TABLE_VAL F];
        [TABLE_VAL T; TABLE_VAL T]]
       [FSIG;FSIG;FSIG;TSIG] TSIG
```

We use the HOL booleans F and T for 0 and 1, respectively. Note that the values given in the input rows and default value are not values but signals: that is, functions from time to a value. The constant signals for 0 and 1 are thus represented by TSIG and FSIG which are just lifted versions of the constants.

```
FSIG = λ(t:num). F
TSIG = λ(t:num). T
```

The definition that we give is less flexible than the MDG system's tables since all the input values are restricted to be of the same type, whereas in the MDG system they can be of a variety of sorts. In the next subsection we present a way to deal with this problem.

### 4.3 Application of the Table Definition to Multisorts Inputs

In the above formalization of the MDG tables, it is assumed that the inputs of the table are of the same type. This is true for most components (gates) of the MDG-HDL library. In order to represent the MDG table of a more general component with inputs of different types in HOL, we need to extend our formalization to accommodate a list of inputs (the first argument of the table definition) with different types. As an example we present the formalization of the state transition diagram of the timing block of the Fairisle ATM switch fabric [6] in terms of an MDG table in HOL. Fig. 2 shows the finite state machine of the behavior of this timing block, which consists of three symbolic states (RUN,WAIT,ROUTE), and has two inputs (*frameStart* and *anyActive*) and one output *routeEnable*.

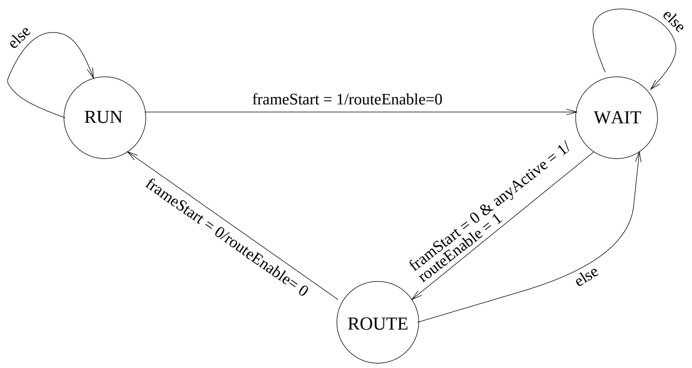



**Fig. 2.** State transitions of the Fairisle switch fabric timing block

The MDG table of the next state function of this state machine is:

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

While the inputs and the output are of boolean sort, *timing_state* and *n_timing_state* are of a concrete sort with the enumeration: RUN, WAIT, ROUTE. We hence need to create a common type for all the input variables as well as the state variable *timing_state* in order to use our definition of tables in HOL.
Let TIMING_TYPE_VAL be the states of our machine:

```
TIMING_TYPE_VAL = RUN | WAIT | ROUTE
```

The common type for all the input variables, TIMING_SPEC_TYPE, is defined as:

```
TIMING_SPEC_TYPE = TRANS of ‘a | STATE of TIMING_TYPE_VAL
```

Having these ingredients, we derive the HOL definition of the above table as:

```
TABLE [anyActive;frameStart;timing_state](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 F);TABLE_VAL(STATE ROUTE)];
  [DONT_CARE;TABLE_VAL(TRANS T);TABLE_VAL(STATE ROUTE)]]
  [WAITSIG;RUNSIG;ROUTESIG;RUNSIG;WAITSIG] WAITSIG
```

where RUNSIG,WAITSIG,ROUTESIG, are lifted versions of the constants RUN, WAIT and ROUTE.

```
RUNSIG = λ(t:num).(STATE:TIMING_TYPE_VAL -> bool TIMING_SPEC) RUN
WAITSIG = λ(t:num).(STATE:TIMING_TYPE_VAL -> bool TIMING_SPEC) WAIT
ROUTESIG = λ(t:num).(STATE:TIMING_TYPE_VAL -> bool TIMING_SPEC) ROUTE
```

### 4.4 Formal Verification of the Library Components

To verify a library component, we must prove that the semantics of the table used in the MDG implementation is equivalent to the semantics of the component. For example, for the **and** component we prove the theorem:

```
∀x1 x2 y. AND x1 x2 y = AND_TABLE x1 x2 y
```

This can be proved easily in HOL by first rewriting with the definitions and then applying the recently added, efficient tactic MESON_TAC. This was packaged into a simple tactic, COMB_MDG_TAC, that was then used to prove all combinational components in the library. For sequential components such as the RS flip-flop and JK flip-flops with and without enable, we use a different tactic, SEQ_MDG_TAC, based on rewriting and cases analysis, that we parameterize with respect to the input variables.

## 5 Use of Results

### 5.1 MDG Components Library

The main result of this work is that we have verified all components of the MDG component library except a few that are not implemented in terms of tables. This gives increased confidence in the MDG system. The work was originally motivated by an error found in a table in an early version of the system. This error was introduced because a new component (a JK flip-flop with enable) was added to the system on the fly. It is likely that new components will be added in the future. Tables provide a flexible and convenient way for this to be done. However, as they consist of tables of 1's and 0's it is easy to make mistakes. Our HOL theory and automatic proof tool, provide a simple, fast and convenient method for such future additions to be formally verified. As for the library components, this consists of giving the formal specification of the component in HOL, writing the Table definition in HOL, setting the goal and applying the tactic. The proof will of course only be automatic for simple components of the level of complexity found in the existing library. Users of the MDG system are liable to want to define their own similar primitive components. They can use the theory and proof tool in the same way. We have thus provided a toolkit (albeit limited) for both users and developers of the MDG system.

### 5.2 HOL Tables Theory

A definition and associated theory of tables is a useful addition to HOL in its own right, as tables provide a flexible means of giving definitions of logic functions. The definition we used in our proofs is not suitable directly as a general definition, as it has an explicit notion of time $t$ built in: this was most convenient for our application as we did wish to include a time component in our definitions. A more general definition of a table would have thus unnecessarily complicated the final definitions and proofs.

A more suitable definition for general use would be:

```
(Tab_match inputs []= T) ∧
(Tab_match inputs (CONS v vs) =
       (( HD inputs = TableVal_to_Val (v:'a Table_Val) ) ∨
       (v = DONT_CARE)) ∧ (Tab_match (TL inputs) vs) )

(TAB inps (out:'b) ([]:('a Table_Val list) list) V_out default =
(out = default) ) ∧
(TAB inps out (CONS v vs) V_out default =
       ((Tab_match inps v) =>
       (out = (HD V_out)) |
       (TAB inps out vs (TL V_out) default)))
```

### 5.3 Formal Verification of the MDG System

Some of the library components such as the multiplexer are implemented directly in terms of MDGs, rather than as a table that is then implemented as an MDG. The other such components are the register (with and without a control input), fork (equality of signals), transform (for uninterpreted functional blocks), and drivers (essentially a guarded command). However, in theory all the components could be implemented as tables. If this were done, those components could be verified in the same way as the ones we considered here. Then, the correctness of the library would depend only on the correctness of the translation of the tables into MDGs, rather than on the way a series of components were implemented as MDGs. It is also worth noting that tables have a fairly simple translation into a basic MDG.

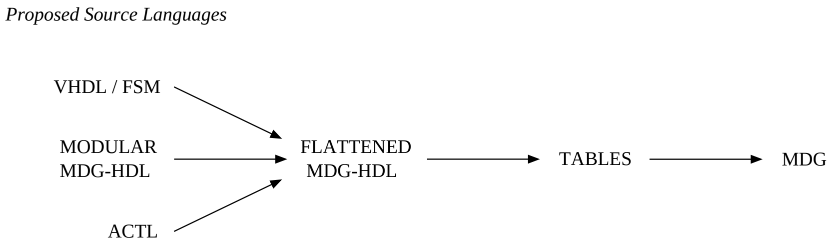



**Fig. 3.** Format translations within the MDG system

The above proofs correspond to the first step in a larger project to verify a formal specification of the MDG system [15]. The system can be considered as a series of translators, translating between different intermediate languages, as shown in Fig. 3. One step in that process is the translation from the MDG-HDL language to a subset of the language with only tables as components. This table subset is then translated into MDGs in a series of further translation steps. Currently structural, behavioral and property specifications are all given in this low level language. However, translators from specialist higher level languages are under development as shown in Fig. 3.

The correctness of a translator between two languages can be stated in terms of the semantics of the languages, as shown in Fig. 4. Essentially this states that the translation should preserve the semantics of the source language. This is the traditional form of compiler specification correctness used in the verification of compilers [3]. The same approach can be used to specify and verify a hardware verification system such as MDG. For the translation to tables the correctness theorem would have the form

$$\forall \mathtt{h}.\ S_h(\mathtt{h}) = S_t(T(\mathtt{h}))$$

where $h$ is a hardware description, $S_h$ is the semantics of the source language, $S_t$ is the semantics of the table subset and $T$ is a functional specification of the translation between the two. The proof of this theorem proceeds by structural induction on the source language. It requires lemmas stating that the translation of each kind of hardware component is correct. These lemmas are in fact the theorems that we have proved above.

### 5.4 MDG-HOL Hybrid System

The current work is also of relevance to a further project: namely that of combining the MDG and HOL systems, to give a hybrid hardware verification tool. The verification of a 16 by 16 switch fabric, already verified in the pure HOL system [7] is being used as a case study in this project. If results from the MDG system are to be imported into HOL, then the structural and behavioral specifications used must have HOL equivalents. The hardware semantics used in the work described here provides such a basis. Using the same semantics in the

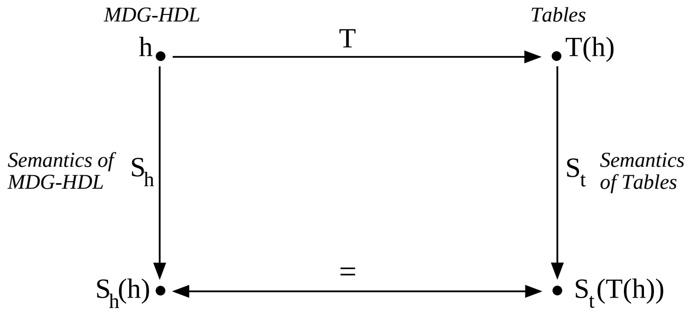



**Fig. 4.** Compilation correctness

verification of a specification of MDG and as the basis of the combination of the verification systems means that the translation correctness theorem provides a justification for the linkage. If different semantic foundations were used in the two approaches, there would be no guarantee of this.

As the two systems use different specification mechanisms, with the HOL approach allowing more abstract descriptions, the user of such a hybrid system would need to prove the correspondence of the HOL specification and the form needed for input to the MDG system. In such a proof, the MDG-HDL specification could be considered as a concrete implementation for which the HOL specification was the behavioral specification. This is precisely what we have done for the MDG primitives. The tables provide a concrete MDG-HDL description, the hardware semantic functions a more intuitive HOL one. Having proved the equivalence, further HOL proofs would use the latter rather than the table description.

## 6 Summary and Conclusions

We have formally specified the semantics of the MDG component library using HOL. This includes a formalization of the Table construct that forms the heart of the MDG wide-spectrum hardware description language. We then formally described the table implementations of each of the hardware components that are implemented in terms of tables in the MDG system. We verified the correctness of each table implementation against the formal specification of the component. This was done using two simple automated tactics in HOL, one for combinational library components and one for sequential ones. These tactics were sufficiently flexible and powerful to verify all table-based components of the MDG library.

We have thus proved the correctness of one small but crucial part of the MDG system, thus increasing the confidence of users of MDG have of the system. Whilst the table implementations we verified are a relatively simple part of the system, errors have previously been uncovered in such table definitions. Our verification is thus of practical utility.

We have demonstrated how a theorem prover can be of utility on real and highly complex software, if a small and well-defined problem is tackled. Furthermore, by verifying a decision diagram system using a verification system implemented on a different paradigm, we have reduced the possibility that the verification is flawed due to an error in the verification system used.

We have also given formal specifications of the library components, which will help ensure users have an accurate understanding of those components and so use them correctly. The automated HOL proof tool can be used by system designers to ensure that new components added to the MDG system are correctly implemented. Similarly users of MDG who need to define their own basic components in terms of tables can use the HOL proof tool to ensure the correctness of those tables.

Finally, the work done forms the first step of a larger project to verify a formal specification of the MDG system. The theorems proved are the main lemmas that would be needed in verifying one stage of such a formal specification. Similarly, the hardware semantics given for the components are an essential step in the ongoing project to combine the HOL and MDG systems. By using the same semantics for both these projects we open the way for linking the correctness proof of MDG with hybrid proofs of hardware using the combined system. We have also paved the way towards providing a toolkit that can be used by both the users of the combined MDG system and developers of that system.

## References

1. R. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
2. E. Cerny, F. Corella, M. Langevin, X. Song, S. Tahar, and Z. Zhou. Automated Verification with Abstract State Machines Using Multiway Decision Graphs. In T. Kropf, editor, *Formal Hardware Verification: Methods and Systems in Comparison*, State-of-the-Art Survey. Lecture Notes in Computer Science, Vol. 1287, pages 79–113. Springer-Verlag, 1997.
3. L.M. Chirica and D.F. Martin. Toward Compiler Implementation Correctness Proofs. *ACM Transactions on Programming Languages and Systems*, 8(2):185–214, 1986.
4. C.-T. Chou and D. Peled. Formal Verification of a Partial-Order Reduction Technique for Model Checking. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, Vol. 1055, pages 241–257. Springer-Verlag, 1996.
5. F. Corella, Z. Zhou, X. Song, M. Langevin and E. Cerny. Multiway Decision Graphs for Automated Hardware Verification. *Formal Methods in System Design*, 10(1):7–46, 1997.
6. P. Curzon. The Formal Verification of the Fairisle ATM Switching Element: an Overview. Technical Report no. 328, University of Cambridge, Computer Laboratory, March 1994.
7. P. Curzon and I. M. Leslie. A Case Study on Design for Provability. In *Proceedings of the International Conference on Engineering of Complex Computer Systems*, pages 59–62. IEEE Computer Society Press, November 1995.

8. P. V. Homeier and D. F. Martin. A Verified Verification Condition Generator. *The Computer Journal*, 38(2):131–141, 1995
9. M. J .C. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanical Logic of Computation.* Lecture Notes in Computer Science, Vol. 78. Springer-Verlag, 1979.
10. M. J. C. Gordon. HOL: A Proof Generating System for Higher-order Logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, 1988.
11. M. J. C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-order Logic.* Cambridge University Press, 1993.
12. S. Tahar, Z. Zhou, X. Song, E. Cerny, and M. Langevin. Formal Verification of an ATM Switch Fabric using Multiway Decision Graphs. *Proc. IEEE Sixth Great Lakes Symposium on VLSI*, Ames, Iowa, USA, pages 106–111. IEEE Computer Society Press, March 1996.
13. J. von Wright. Representing Higher-Order Logic Proofs in HOL. *The Computer Journal*, 38(2):171–179, 1995
14. W. Wong. Recording and Checking HOL Proofs. In E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors, *Higher-Order Logic Theorem Proving and Its Applications.* Lecture Notes in Computer Science, Vol. 971, pages 353–368. Springer-Verlag, 1995.
15. H. Xiong and P. Curzon. The Verification of a Translator for MDG's Components in HOL. To be presented at *MUCORT'98, Computers and Engineering in the Millennium*, Middlesex University, April 1998.
16. Z. Zhou and N. Boulerice. MDG Tools (V1.0) User's Manual. University of Montreal, Dept. D'IRO, 1996.