Hybrid Verification Integrating HOL Theorem Proving with MDG Model Checking

Rabeb Mizouni [a], Sofiène Tahar [a], and Paul Curzon[b]

[a]Department of Electrical & Computer Engineering,
Concordia University, Montreal, Quebec,
H3G 1M8, Canada
{mizouni,tahar}@ece.concordia.ca

[b]Department of Computer Science,
Queen Mary University of London, Mile End, London,
E1 4NS, UK
pc@dcs.qmul.ac.uk

In this paper, we describe a hybrid tool for hardware formal verification that links the HOL (Higher-Order-Logic) theorem prover and the MDG (Multiway Decision Graphs) model checker. Our tool supports abstract datatypes and uninterpreted function symbols available in MDG, allowing the verification of high level specifications. The hybrid tool, HOL-MDG, is based on an embedding in HOL of the grammar of the hardware modeling language, MDG-HDL, as well as an embedding of the first-order temporal logic $\mathcal{L}_{mdg}$ used to express properties for the MDG model checker. Verification with the hybrid tool is faster and more tractable than using either tools separately. We hence obtain the advantages of both verification paradigms.

1. Introduction

Hybrid verification approaches that link interactive proof tools with automated (e.g. BDD based) proof tools are now common. Such links gain the automation of the BDD tools instead of, for example, using the interactive tool to manage the proof. Whilst abstraction can be dealt with by the interactive tool, it is advantageous if it could also be dealt with by the automated tool. In this paper, we describe a hybrid tool that does this. It combines the HOL theorem prover [13] and the MDG model checker [20]. HOL (Higher-Order Logic) is an interactive theorem prover based on higher-order logic. The MDG (Multiway Decision Graphs) system is a decision diagram based verification tool for Abstract State Machines (ASM) verification encoded by multiway decision graphs [7]. The latter extend Reduced-Ordered Binary Decision Diagrams (ROBDD) [4] with abstract datatypes and uninterpreted function symbols. It is this feature that allows abstract designs to be verified automatically using MDG, rather than needing to do such proof wholly in the theorem prover HOL. The down side of this abstraction facility is that in some cases the state reachability algorithm may not terminate [2]. This is due to the fact that edges may be labelled by terms that are arbitrary large and hence arbitrarily many. In a pure system for this rare case, the user would have to use one of many heuristics provided in [2,22]. The proposed hybrid tool gives ways to overcome the problem.

There has been a great deal of effort combining model checking tools with proof systems. Similar work to ours, though based on binary decision diagrams rather than multiway ones, includes Rajan *et al.*'s [16] integration of a propositional μ-calculus model checker with PVS, and Schneider and Hoffmann [15] who linked the CTL model checker SMV to HOL. Gordon [8] took a different approach with the BuDDy BDD package, providing a secure and general programming infrastructure to allow users to implement their own BDD-based verification algorithms integrated within

the HOL system rather than tools being linked externally. Sugar2.0 [3] has also been embedded in HOL in order to prove meta-theorems. Sugar provides ways to specify properties for both simulation and formal verification, providing the users with an interface to combine both theorem proving and model checking, with simulation techniques. Forte [12], based on the work of Aagarad *et al.* [1] is one of the maturest formal verification environments based on tool integration including simulation. It has been used in large-scale industrial verification projects at Intel. Its power comes from the very tight integration of the two provers, using a single functional language, as both the theorem prover's meta-language and its object language.

The tool described here extends the capabilities of an earlier HOL-MDG tool and methodology [17,11] for hierarchical hardware verification. The main contribution of the current work is that our hybrid tool supports the *abstract* datatypes of MDG in addition to concrete (enumeration/Boolean) sorts in [11,17]. This allows abstract designs to be passed from HOL to MDG for verification. This allows, for example, larger data paths to be dealt with automatically than with a BDD based linkage. In particular, we extended a previous HOL formalization of the MDG modeling language, MDG-HDL [14]. We also implemented an interface that automatically supports the communication between the MDG and HOL tools. It generates the necessary MDG files from the HOL files, passing them to the model checker, takes back the MDG results, interprets them, and finally submits them to HOL in an appropriate form (see Figure 1).

The tool supports both equivalence checking and *model checking* of abstract designs: a further extension of the original hybrid tool. This involved embedding the MDG temporal property specification language, $\mathcal{L}_{mdg}$ in HOL. An additional novel aspect is the explicit support of model reduction in HOL based on the natural design hierarchy and the specification being verified.

The rest of the paper is organized as follows. Section 2 describes the embedding of MDG-HDL language and the $\mathcal{L}_{mdg}$. In Section 3, we present the proposed hybrid verification procedure. Section 4 describes the internal structure of the hybrid tool. In Section 5, we display some sample experimental results. Finally, Section 6 concludes the paper.

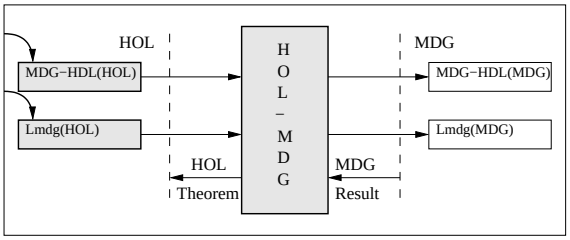

Figure 1. The Hybrid Tool Overview

2. Embedding MDG Specification Languages in HOL

2.1. MDG-HDL

The MDG tools accepts model descriptions in a Prolog-style HDL (Hardware Description Language) called MDG-HDL [21]. MDG-HDL models are then compiled into Abstract State Machines (ASM), which are encoded using internal MDG data structures.

The syntax used in MDG-HDL is based on an ordinary many-sorted first order logic. The vocabulary consists of sorts, constants, variables and function symbols, with a distinction between *abstract* and *concrete* sorts. Concrete sorts have an *enumeration* while abstract sorts do not. This enumeration represents a set of distinct constants of one defined sort. These constants are referred to as *individual constants*. It is possible to define a constant for an abstract sort, referred to as *generic constants*. The distinction between abstract and concrete sorts leads to a distinction between three kinds of function symbols. Let f be a function symbol of type $\alpha_1 \times \alpha_2 \times \ldots \times \alpha_n \rightarrow \alpha_{n+1}$. If α_{n+1} is an abstract sort, then f is an *abstract function symbol*. If all the $\alpha_1 \ldots \alpha_{n+1}$ are concrete, then f is a concrete function symbol. If α_{n+1} is concrete while at least one of the $\alpha_1 \ldots \alpha_n$ is abstract, then f is referred to as a *cross-operator*. Concrete function symbols must

have an explicit definition, since they are eliminated before computing the MDG, while abstract function symbols and cross-operators are *uninterpreted*. This means implementation models can include abstract features such as n-bit words, and abstract functions.

MDG-HDL supports structural descriptions, behavioral ASM descriptions, or a mixture of both. As part of the MDG software package, the user is provided with a large set of pre-defined modules such as logic gates, multiplexers, registers, bus drivers, etc. Besides the logic gates which only use Boolean signals, the other components allow signals with both concrete and abstract types. Moreover, a special *table* structure is defined. Tables can be used to describe functional blocks in both implementations and specifications. A table is similar to a truth table. It has as entry values first-order terms in the rows. It is composed of a list of rows which define is a list of inputs values and their corresponding output. A default value of the output is defined if the inputs sequence given does not fit the defined rows.

The table structure as well as the MDG components library have been embedded previously in HOL [5]. Since the grammar of the language itself was not embedded, the differentiation between various terms (abstract and concrete) was not previously possible. We overcome this limitation in the current work.

Embedding :

To embed the grammar of the MDG-HDL language in HOL, it is necessary to cover the syntax of the subset of many sorted first-order logic used by MDG. In HOL, we define an abstract sort to be of type α to *string* as seen in the definition below. The second parameter in this definition is specified mainly to permit the user to impose a specific abstract sort like *word5* or *word10*, rather than the default abstract MDG sort *wordn* (used for n-bit words).

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

Predicates that specify which kind of sort we are dealing with are also defined .

Functions, *MDG_Fun*, are specified by their input list and their output. For MDG, a function has a unique output.

```
MDG_Fun = MDG_FUN of  string =>
      ('a MDG_VAR) list =>  ('a MDG_VAR)
```

Since the domain of the function is a list of variables, to determine if the function is abstract, we test if both inputs and output are of abstract sort. So, we define a predicate to determine recursively if the list is composed of abstract variables. The test is first done on *h*, the head of the list, and it is repeated recursively on *tl*, the tail of the list, until reaching the empty list.

```
⊢def  AbstractVarList(h::tl) =
        ((IsAbstractVar h)        ∧
         (AbstractVarList tl))    ∧
         (AbstractVarList  [ ] = T )
```

Thereafter, a function is abstract if both its domain and range are abstract:

```
⊢def  AbstractFunc (MDG_FUN nm InputVarList OutVar)=
     (AbstractVarList InputVarList) ∧
     (IsAbstractVariable OutVar)
```

After defining all the different elements of the MDG vocabulary, we can define the different kinds of MDG terms. An *MDG_term* is either:

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

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

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

Based on the embedding of the MDG-HDL grammar, an MDG *table entry*, called *Table_Val* is defined as follows:

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

A function that returns back the value of a table entry is also defined:

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

The above HOL definition specifies a new HOL datatype *Table_Val*, which has two constructors : *TABLE_VAL* and *DONT_CARE*. The latter can take any type. Curzon *et al.* [5] defined the matching of input values to table values. A match occurs if either the table value is don't-care, or the value on the input is identical to the table value. This property must hold for each table entry. It is defined recursively by the function *table_match*.

```
⊢def (Table_match inputs [ ] (t:num) = T)
   ∧ (Table_match inputs (CONS v vs) t) =
    (((HD(inputs) t) =
      TableVal_to_Val (v:'a Table_Val))
   ∨ (v = DONT_CARE))
   ∧ (Table_match (TL inputs) vs t)
```

Next, we give the definition *table* stating that the *Table_match* test is first done on the first element in the input list. If there is a match on a given row, the output has the corresponding value. Otherwise it is repeated on the rest of the list until reaching the empty list. If there is no match, the output of the considered entry will be assigned the default value.

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

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

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

Finally, note that the outputs of the table are always considered as signals, which explains their definition according to the time *t*.

In summary, we have semantically embedded the full version of the MDG hardware description language, MDG-HDL, supporting abstract variables and uninterpreted functions in HOL. All redefined modules, such as logic gates, registers, multiplexers, etc., have been defined in HOL and verified against behavioral specifications in terms of tables. This provides the basis of a trusted integration of HOL and MDG. MDG hardware descriptions can be written directly in HOL via the developed embedding.

2.2. $\mathcal{L}_{mdg}$

$\mathcal{L}_{mdg}$ [19] is the properties specification language of the MDG model checker. It is a subset of first-order linear time logic, which supports abstract data sorts and uninterpreted functions.

The properties allowed in $\mathcal{L}_{mdg}$ can have the following templates:

Property :
- Next_let_formula
- | G(Next_let_formula)
- | F(Next_let_formula)
- | (Next_let_formula)U(Next_let_formula)
- | G((Next_let_formula)→(F(Next_let_formula)))
- | G((Next_let_formula) →
  ((Next_let_formula) U (Next_let_formula)))

G, F, and U are the standard linear time logic operators: for all time, at some time, and until, respectively. A *Next_Let_Formula* is defined as:

- each atomic formula is a Next_Let_Formula,
- if p and q are Next_Let_Formulas, then so are: $!p$ (*not p*), $p\&q$ (*p and q*), $p|q$ (*p or q*), $p \rightarrow q$ (*p implies q*), Xp (p holds in the next state), and $LET\ (v = t)\ IN\ p$ where t is an ASM_variable (input, state or output variable) and v an ordinary variable.

A path π is a sequence of states. We use π_i to denote a path starting from s_i, where s_i denotes the i^{th} state in π. All formulas in $\mathcal{L}_{mdg}$ are *path formulas*. We write $(\pi,\ \sigma)\ \models\ p$ to mean that a path formula p is true at path π under a ψ-compatible assignment σ to the ordinary variables. We use $Val_{\pi_0 \cup \sigma}(v)$ to denote the value of term v under a ψ-compatible assignment s to state variables, input variables, and output variables, and a ψ-compatible assignment σ to the ordinary variables. The $\models$ is inductively defined as follows [19]:

$\pi,\ \sigma \models\ v_1 = v_2$ iff $Val_{\pi_0 \cup \sigma}(v_1) = Val_{\pi_0 \cup \sigma}(v_2)$.
$\pi,\ \sigma \models$ LET $(v_1 = v_2)$ IN p iff $\pi,\ \sigma' \models\ p$ where $\sigma' = \{(v_1, \sigma(v_1))\} \cup \{(v_1, Val_{\pi_0 \cup \sigma}(v_2))\}$.
$\pi,\ \sigma \models\ !p$ iff it is not the case that $\pi,\ \sigma\ \models\ p$.
$\pi,\ \sigma \models\ p\&q$ iff $\pi,\ \sigma\ \models\ p$ and $\pi,\ \sigma\ \models q$.
$\pi,\ \sigma \models\ p|q$ iff $\pi,\ \sigma\ \models\ p$ or $\pi,\ \sigma\ \models q$.
$\pi,\ \sigma \models\ p \rightarrow q$ iff $\pi,\ \sigma\ \models\ !p$ or $\pi,\ \sigma\ \models q$.
$\pi,\ \sigma \models\ Gp$ iff $\pi_j,\ \sigma\ \models\ p$ for all $j \geq 0$.
$\pi,\ \sigma \models\ Fp$ iff $\pi_j,\ \sigma \models\ p$ for some $j \geq 0$.
$\pi,\ \sigma \models\ Xp$ iff $\pi_1,\ \sigma \models\ p$.
$\pi,\ \sigma \models\ qUp$ iff for some $k \geq 0,\ \pi_k,\ \sigma \models\ q$, and $\pi_j,\ \sigma\ \models\ p$ for all j $(0 \leq j \leq k)$.

Embedding :

In our HOL embedding of $\mathcal{L}_{mdg}$, we consider that each logical proposition (property) p is a function of the path, expressed here by s, and the current state. The path can be formulated as a history function keeping trace of the states, where the property holds. For instance, the HOL definition of the G operator is defined as follows:

```
⊢def  LMDG_G p s = ∀t. p s t
```

That is, for all time t, property p holds of path s at that time. Note that we do not need to quantify over the history function s, while we have to verify that the property p holds over the different states of a given path. So, *LMDG_G (p s)* holds if for all states, *p(s(t))* holds.

A similar HOL definition is provided for each operator of $\mathcal{L}_{mdg}$.

```
⊢def  LMDG_F p s = ∃t. p s t
⊢def  LMDG_X p s t = p s (t+1)
⊢def  LMDG_U p q s =
      ∃t. (p s t  ∧ (∀t1. t1 < t →  q s t1)
```

In addition, let, negation, disjunction, conjunction, and implication of predicates are defined as functions of path formulas p and q, as follows:

```
⊢def  LMDG_LET (v1,v2) p s t =
     (λ v1. p s t ) ⟹  (λ v2. p s t )
⊢def  LMDG_NOT p s t = ¬ p s t
⊢def  LMDG_AND p q s t =  p s t ∧  q s t
⊢def  LMDG_OR p q s t =  p s t ∨  q s t
⊢def  LMDG_IMP p q s t = ¬(p s t) ∨  q s t
```

In summary, we have semantically embedded the property specification language of MDG in HOL. $\mathcal{L}_{mdg}$ specifications can be written directly in the theorem prover using the embedding. This opens the way for writing MDG style model checking goals in HOL, proving them using HOL or MDG.

3. Hybrid Verification with HOL-MDG

The hybrid tool developed consists of an interface integrating the HOL theorem prover and the MDG model checker. During the verification procedure, the user deals mainly with HOL. As shown in Figure 2, the user starts by giving the HOL design model, property specification, and the goal to be proven. The respective MDG files (property specification, design model, symbol order, algebraic specification, and fairness constraints) are generated automatically and sent to the MDG tool for model checking. If the property holds, a HOL theorem is created. This

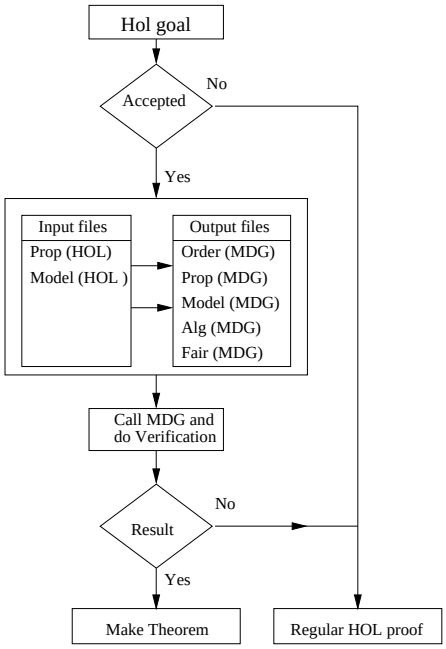

Figure 2. Verification Procedure with the Hybrid Tool

could be used in higher HOL proofs, for example proving theorems about the consequences of the properties. If the verification within the MDG tool fails (due to the property checking to false, non-termination or state explosion), we have to perform the proof interactively using the theorem prover.

The tool does not accept any arbitrary HOL specification: only MDG-style models and properties using the embedded HOL theories presented. The HOL goal should also be an implication:

$$\vdash Model \supset Property$$

Since the verification is done in MDG, we need to formalize the (MDG) result in HOL. Therefore, we convert the MDG results into a form that can be used [18]:

$$\vdash FormalizedMDGresult \supset Model \supset Property$$

A formalized version of this general conversion theorem into HOL has been proved in HOL [18]. The proved theorem can be instantiated for any design and any property under consideration.

MDG model checking result is converted to a form that can be used in HOL to infer the properties from the design model [18].

Our hybrid tool also supports hierarchical verification, where it is able to extract in HOL the block about which we want to check a property, then generating files of the specific block only. This is achieved by defining the structure "block" in a recursive manner. So, for each block, we are able to determine its subblocks (see Figure 3). Hence, the model checker deals with the verification of the considered block only, not the whole design. As a result, we save on model size without constraining the user to write another specification for the appropriate block. This idea of program slicing is well-known in the model checking literature [6]. The difference in our work is the fact that the "slices" are extracted while expanding the proof goal by the theorem prover HOL, and based on the definition of the design block. In

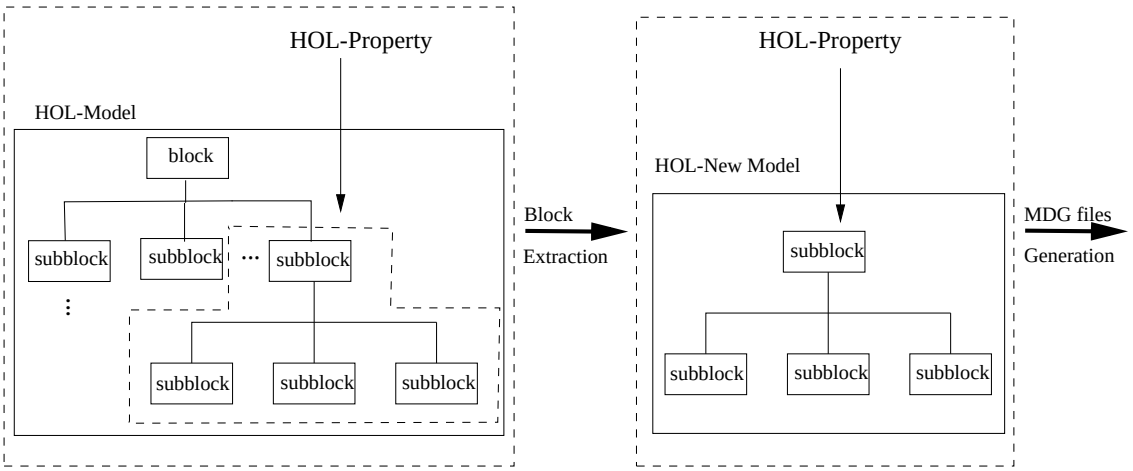

Figure 3. Block Extraction

our approach, it is therefore done formally within HOL rather than informally outside the tool.

4. HOL-MDG Hybrid Tool Structure

Our hybrid tool is written in SML. It is composed of five main modules: the *Hybrid Tool Interface*, the *Property Module*, the *Description File Module*, the *HOL Goal Parser Module* and the *MDG Interaction Module* (Figure 4). The user's interface [9] to the hybrid tool is a Java GUI. It is responsible for:

1. getting the HOL goal, the property file and the model description file,
2. passing the files to HOL,
3. loading the $\mathcal{L}_{mdg}$ and MDG-HDL theories, and
4. communicating the result to the user at the end of the verification process.

The user thus sees the hybrid tool as an integrated system but one that is more powerful than MDG alone. In the second module, the *Property Parser* generates as output a data structure from which the *MDG File Generator* produces the MDG property file, and the *Property Type Generator* provides the property type. The latter contains information about the type of property submitted to the tool, according to which, it calls the appropriate property checking algorithm. The *Description File Module* flattens the specification by removing hierarchy.

When parsing the goal, we obtain the name of the property and the block to check. The latter can be either the main module in the model description or one of its submodules. If the specification is written in a hierarchical way, it is possible to extract the target module, and its submodules, discarding the others. The *Block Extraction Module* achieves this task. In the next step, the corresponding MDG files are generated, including:

- *MDG model* and *MDG property* files,
- an *algebraic* file containing sorts, functions, and rewriting rules,
- an *order* file, giving a total order of variables and function symbols, and eventually
- *fairness* files, each describing an imposed fairness constraint.

The MDG file generation is done automatically. The HOL specification file contains two

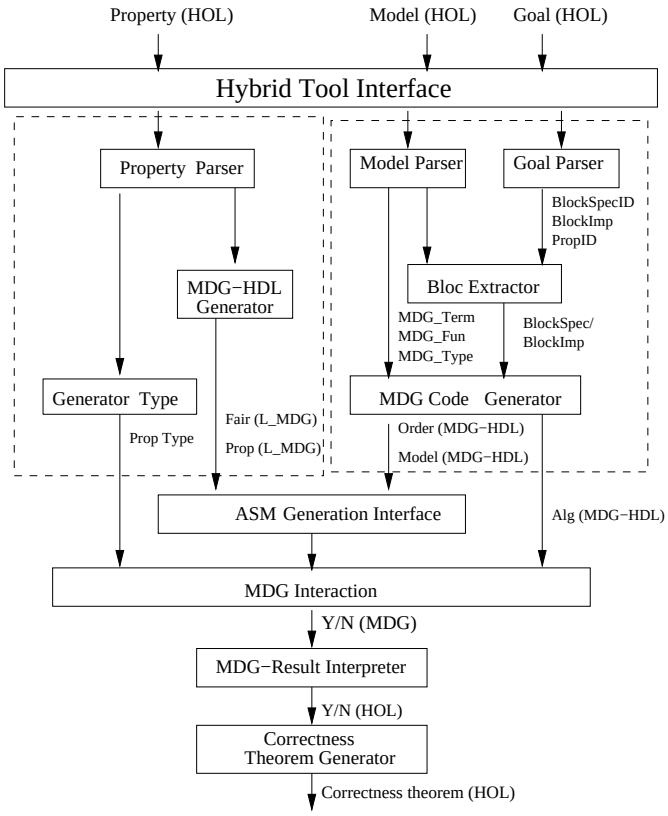

Figure 4. Hybrid Tool Structure

main parts. The first is dedicated to the definition of the different sorts, functions, and MDG terms used. The second is dedicated to the tables definitions. Using a syntactical analysis of the submitted HOL files, our tool extracts the useful information from them to generate the MDG files in the appropriate MDG-HDL syntax.

Before proceeding with the model checking operation, the MDG tool has to encode the MDG-HDL syntax to generate ASMs. Since we wanted the communication between the linked tools to be automatic, we implemented a special module, called *ASM Generation Interface* that implicitly executes the appropriate MDG instructions. The *MDG Interaction Module* does the communication with MDG. It takes all the generated MDG files, the property type and the fairness number. The latter are provided by the property parser module. They indicate respectively the number of fairness constraints in the HOL property, if they exist, and their temporal type. All these files are supplied to the MDG tool, which performs the verification process and passes the result to HOL through the *MDG Result Interpreter Module*. If the property holds, a theorem is generated in HOL.

5. Experimental Results

We have experimented with our hybrid tool using a number of benchmark designs including the Island Tunnel Controller (ITC) [14] (Figure 5), which experimental results we report here. The ITC controls the traffic lights at both ends of

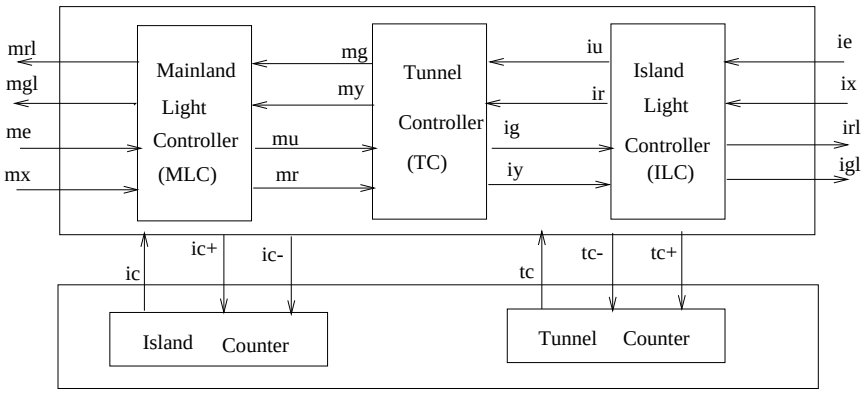

Figure 5. ITC Structure

a tunnel connecting a mainland and island. It was chosen for two reasons. First, its specification contains abstract sorts and functions. It was not possible to express the specification of this example in the tool in [11]. Second, the same example was verified in [22], where the authors faced a problem of non-termination in the Island Counter module. The hybrid tool offers the solution of doing a hybrid verification, such that the subblocks causing the non-termination problem are verified within the HOL theorem prover interactively, while those which do not are verified within the MDG model checker.

The input specifications for the ITC were written in HOL, using the HOL MDG-HDL theory [14]. It is composed of a term declaration of the MDG part, the different table specifications and the main modules. The specification is written in a hierarchical way. Each component is represented by the conjunction of its tables. The whole system therefore is the conjunction of the five mentioned blocks.

Experimental results on the verification of a set of properties are given in Table 1. It gives CPU time, verification memory usage and number of MDG nodes generated as well as the number of components and signals of the reduced (extracted) design model effectively used for model checking in MDG. It is clear that verification is much faster than doing the proof interactively with HOL. At the bottom of Table 1, we give the example experimental results of checking Property 1 and Property 3 without block extraction done in the theorem prover side, i.e., on the whole model. We can clearly see that the CPU time and memory consumption were decreased by more than half in the former case, which is due to the block extraction. The results here are similar to those in [20], where only the MDG tool is used on the full model. This fact proves that our hybrid tool achieves the verification without obstructing the model checker.

6. Conclusions

In this paper, we presented a hybrid verification approach and tool integrating the HOL theorem prover and the MDG model checker. In an earlier HOL-MDG tool, where HOL and the MDG equivalence checker were linked, neither abstract data sorts nor abstract functions were supported. The main contribution of our work is the extension of this tool to handle these main features of MDG compared to BDD based model checkers as with other tools. For this purpose, we embedded in HOL the grammar of the MDG input languages $\mathcal{L}_{mdg}$ and MDG-HDL. Next, we provided a new link between HOL and the MDG model checker. Our system handles abstraction for model checking and equivalence checking. Furthermore, it directly supports hierarchical proof to be conducted

Table 1
Experimental Results on the ITC

Property	$CPU_{(s)}$	$Memory_{(MB)}$	MDG Nodes	#Components	#Signals
Property1	0.32	0.66	318	18	32
Property2	0.36	0.77	313	13	31
Property3	0.41	0.73	401	16	34
Property4	1.12	1.91	1266	13	29
Property5	0.91	1.26	1027	10	26
Property6	0.93	1.77	1166	13	29
Property7	1.15	1.39	11002	16	33
Property8	1.15	1.39	11002	16	33
Property1(*)	0.74	1.38	870	26	62
Property3(*)	0.87	1.46	1027	26	62

saving verification time and memory usage. It also provides a way of overcoming the non termination problem of MDG. The tool has been tested on several examples, including the Island Tunnel Controller reported here. In a future work, we intend to apply our tool on more complex designs as well as looking into ways to render the MDG-HOL specification templates more user-friendly.

REFERENCES

1. M.D. Aagaard, R. Jones, and C. Seger. Lifted-FL: A Pragmatic Implementation of Combined Model Checking and Theorem Proving. In *Theorem Proving in Higher Order Logics*, LNCS 1690, pages 323-340, Springer Verlag, 1999.
2. O. Ait Mohamed, X. Song, and E. Cerny. On the Non-termination of MDG-based Abstract State Enumeration. In *Theoretical Computer Science*, 300: 161-179, 2003.
3. I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The Temporal Logic Sugar. In *Computer Aided Verification*, LNCS 2102, pages 363-367, Springer Verlag, 2001.
4. R. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. In *ACM Computing Surveys*, 24(3):293-318, September 1992.
5. P. Curzon, S. Tahar, and O. Ait-Mohamed. Verification of the MDG Components Library in HOL. *Supplementary Proc. International Conference on Theorem Proving in Higher-Order Logics*, pages 31-45, Canberra, Australia, September 1998.
6. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
7. F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny. Multiway Decision Graphs for Automated Hardware Verification. In *Formal Methods in System Design*, 10(1):7-46, 1997.
8. M. Gordon. Combining Deductive Theorem Proving with Symbolic State Enumeration. 21 Years of Hardware Formal Verification, Royal Society Workshop to mark 21 years of BCS FACS, U.K., December 1998.
9. R. Hum, H. Yip, H. Li, R. Mizouni, and S. Tahar. A GUI for linking HOL to MDG. Technical report, ECE Dept., Concordia University, June 2002.
10. J. Joyce and C. Seger. The HOL-Voss System: Model-Checking inside a General Purpose Theorem-Prover. In *Higher Order Logic Theorem Proving and Its Applications*, LNCS 780, pages 185-198, Springer Verlag, 1994.
11. I. Kort, S. Tahar, and P. Curzon. Hierarchical Formal Verification Using a Hybrid Tool. *Software Tools for Technology Transfer*, 4(3):313-322, May 2003, Springer Verlag.
12. T. Melham. Integrating Model Checking and Theorem Proving in a Reflective Functional Language. In *Integrated Formal Methods*, LNCS 2999, pages 36-39, Springer Verlag,

2004.
13. T. Melham and M. Gordon. *Introduction to Higher Order Logic, Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
14. R. Mizouni. Linking HOL Theorem Proving and MDG Model Checking. Master's thesis, Electrical and Computer Engineering Dept., Concordia University, 2003.
15. K. Schneider and D. Hoffmann. A HOL Conversion for Translating Linear Time Temporal Logic to ω-Automata. In *Theorem Proving in Higher Order Logics*, LNCS 1690, pages 255-272, Springer Verlag, 1999.
16. S. Rajan, N.Shankar, and M.Srivas. An Integration of Model-Checking with Automated Proof Checking. In *Computer Aided Verification*, LNCS 939, pages 84-97, Springer Verlag, 1995.
17. V.K. Pisini, S. Tahar, O. Ait-Mohamed, P. Curzon, and X. Song. Formal Hardware Verification by Integrating HOL and MDG. In *ACM 10th Great Lakes Symposium on VLSI*, pages 23-28, Chicago, Illinois, USA, 2000.
18. H. Xiong, P. Curzon, and S. Tahar. Importing MDG Verification Results into HOL. *In Theorem Proving in Higher Order Logics*, LNCS 1690, pages 293-310, Springer Verlag, 1999.
19. Y. Xu and E.Cerny and X. Song and F. Corella and and O. Ait-Mohamed. Model checking for a first-order temproal logic using Multiway Decision Graphs. In *Computer Aided Verification*, LNCS 1427, pages 219-231, Springer Verlag, 1998.
20. Y. Xu. Model Checking for a First-Order Temporal Logic Using Multiway Decision Graphs. PhD Thesis, University of Montreal, Canada, April 1999.
21. Z. Zhou and N. Boulerice. *MDG Tools(V1.0) User's Manual*. University of Montreal, Dept. D'IRO, 1996.
22. Z. Zhou, X. Song, S. Tahar, E. Cerny, F. Corella, and M. Langevin. Formal Verification of the Island Tunnel Controller Using Multiway Decision Graphs. In *Formal Methods in Computer-Aided Design*, LNCS 1166, pages 233-247, Springer Verlag, 1996.