# On the Embedding of the MDG Specification Languages in HOL

**Rabeb Mizouni**
ECE Dept, Concordia University.
Montreal, Quebec, Canada
mizouni@ece.concordia.ca

**Sofiène Tahar**
ECE Dept.Concordia University,
Montreal, Quebec, Canada
tahar@ece.concordia.ca

**Paul Curzon**
School of Computing Science,
Middlesex University,
London, UK.
*P.Curzon@mdx.ac.uk*

**Abstract**

*In this paper, we propose an embedding of the MDG input languages in HOL. The MDG (Multiway Decision Graph) system is a tool for equivalence and model checking. It is based on multiway decision graphs that extend Reduced-Ordered Binary Decision Diagrams with abstract sorts and uninterpreted functions, prime feature of the MDG. The HOL system is a higher-order logic theorem prover. It has an open user-extensible architecture, giving the possibility of adding expressiveness power to the theorem prover by embedding new theories. We have embedded in HOL the grammar of the MDG hardware description language, MDG-HDL, and the first-order temporal logic, $L_{MDG}$, used to specify properties for the MDG model checker. A Hybrid tool for verification, linking HOL with the MDG model checker, is proposed as an application of the developed embeddings.*

**Keywords**

Formal Hardware Verification, Theorem Proving, Model Checking, Hardware Description Languages, Temporal Logic, Language Embedding.

## 1. INTRODUCTION

Formal Verification [10] has been proposed as a way to overcome the limitation of simulation. It tries to mathematically prove that an implementation of a system fully satisfies its specification. There exists today several formal verification approaches like theorem proving, model checking, equivalence checking, etc. Each one has its advantages and drawbacks.

Theorem proving has the potential of dealing with large designs since it supports both abstraction and hierarchical design verification. However, its interactivity and need for user expertise make the verification a cumbersome process. Decision diagram based verification is fully automated but suffers from the state explosion problem. Hence, the integration of these two approaches is expected to reduce the verification efforts required since it benefits from the high expressiveness and scalability of the theorem prover, and the automation of the model checker. The contribution of our work is to embed the specification language of a decision diagram based tool, MDG [3], in a theorem prover environment, HOL [13].

The HOL system [13] is based on higher-order logic, and was originally intended for hardware verification. Thanks to its generality, HOL has been used in many application areas. The basic interface to the system is the functional language ML. The main advantage of theorem proving is the ability to deal with large-scale design. Despite the expressive power of higher-order logic and the many features offered by the HOL system, the verification process is still a cumbersome task since it needs a very deep understanding of the design structure and proof expertise of the user, which make the verification time-consuming.

The MDG system [3] is a decision diagram based verification tool, primarily designed for hardware verification and offering a black-box verification approach. It provides four kinds of verification procedures: combinational equivalence checking, sequential equivalence checking, invariant checking, and model checking. Multiway decision graphs represent a new class of directed acyclic graphs, proposed to overcome the limitation of the ROBDD-based methods. The underlying logic is 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 enumeration while abstract sorts do not. This enumeration represents a set of distinct constants of defined sort. These constants are referred to as individual constants. This distinction between abstract and concrete sorts leads to a distinction between three kinds of function symbols. If the output is from an abstract sort, so the function is defined to be an *abstract function.* If all the inputs and the output are from concrete sort, so the function is defined to be *concrete*. And finally, if the output of the function is concrete, and at least one of its inputs is abstract, the function is defined to be *cross-function.* Concrete function symbols must have explicit definition, while abstract function symbols and cross-operators are *uninterpreted.*

In [9] a hybrid tool and a methodology tailored to perform hierarchical hardware verification have been developed at the Hardware Verification Group of Concordia University. They integrate the HOL theorem prover to the MDG equivalence checker. Usually to reduce the verification complexity, abstraction techniques and hierarchical

verification are used. However, the tool in [9], although efficient in many ways, remains limited to concrete data sorts.

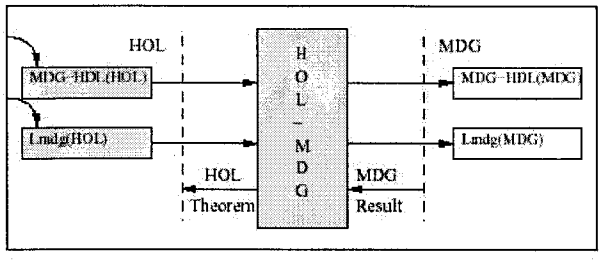



**Figure1: HOL and MDG Model Checker Interface**

The primary contribution of our work is to extend the capabilities of the previous tool to support abstract data types and thus high-level specification. Furthermore, we aim at allowing the user to express and verify properties within the theorem prover rather than the whole behavior. For these purposes, we propose to formalise in HOL the MDG input language for model description MDG-HDL [18], as well as the MDG input language of properties, $L_{mdg}$ [17]. These formalisations are introduced in HOL as new theories. The proposed embedding of both languages in HOL will allow us to interface two tools as shown Figure 1.

The remaining sections of this paper are organised as follows. Section 2 contains related work. In Section 3, we present the proposed formalisation of the MDG-HDL language in HOL. Section 4 describes the embedding of the $L_{mdg}$ temporal logic in HOL. Section 5 proposes an application of the defined embeddings. Section 6 finally concludes the paper.

## 2. RELATED WORK

Many projects have attempted to integrate model checking and theorem proving in order to introduce some automation to the latter. Moreover, they tried to integrate a new expressiveness within the theorem provers. For these purposed, they proposed various ways of embedding external tools specification languages in the logic of the theorem prover.

Rajan *et al.* [14] proposed an approach for the integration of a propositional μ-calculus model checker, based on BDDs, within an automated proof system PVS [4]. They used μ-calculus as a medium for communicating between PVS and the model checker. The μ-calculus was formalised by using the higher-order logic of PVS. The temporal operators that apply to arbitrary state spaces are given the customary fixpoint definitions using the μ-calculus. These expressions were translated to the form required by the model checker. The obtained form was then used to verify the subgoals generated within PVS.

Scheinder and Hoffmann [15] linked SMV [12] to HOL using PROSPER [6]. They embedded the linear time temporal logic LTL in HOL and translated LTL formulae into equivalent ω-Automata, a form that can be reasoned about within SMV. On successful model checking, the results are returned to HOL and turned to theorems. This hybrid tool allows SMV to be used as a HOL decision procedure. The deep embedding of the SMV specification language in HOL allows LTL specifications to be manipulated in HOL.

Other related projects aimed at embedding the semantic of hardware description language in higher-order logic. Kropf and Reetz [11] defined the semantics of a significant subset of VHDL in HOL to formalise a compiler generator, providing a general framework for various verification techniques such as model checking or first-order logic theorem proving. In [7], Gordon too defined three different semantics for a subset of Verilog for use in different applications. One is based on event semantics; the second is based on the trace semantics while the third is based on cycle semantics. In the same range of idea, Boulton et al. [2] presented the semantics embedding of three hardware description languages in higher order logic: HOL-ELLA, HOL-SILAGE, and HOL-VHDL. In the two first languages they used a shallow embedding while in the third one they used deep embedding. A comparison of these approaches is also given.

More recently, Gordon [8] presented an embedding of the Sugar2.0 [1] semantics in higher-order logic supported by HOL. The motivation of such work is mainly proving meta-theorems with a theorem prover to provide a deeper kind of sanity checking, hence developing a machine-readable semantics. In addition, Sugar provides ways to write properties in both simulation and formal verification, providing the users with an interface to combine formal verification techniques, both theorem proving and model checking, and simulation techniques.

## 3. MDG-HDL GRAMMAR EMBEDDING IN HOL

### 3.1 MDG-HDL

The MDG tools accept a Prolog-style hardware description language specification language, called MDG-HDL [18], which allows the use of abstract variables for representing data signals. This MDG-HDL description is then compiled into internal MDG data structures. MDG-HDL supports structural descriptions, behavioral 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 that only use Boolean signals, all other

components allow signals with concrete as well as abstract. Moreover, a special structure is defined called tables. Tables can be used to describe functional blocks in both structural and behavioral descriptions. A table is similar to the truth table; it has as entry values first-order terms in the rows. It is composed of a list of rows. Each row is a list of input values and their corresponding output. A default value of the output is defined if the input sequence given does not fit the defined rows. The table structure as well as the MDG components library has been embedded previously in HOL [5]. Since the grammar of the language itself was not embedded, the differentiation between different terms (abstract and concrete) was not possible. A

### 3.2 Grammar Embedding

To embed the abstraction in the hybrid tool presented in [9], two alternatives were possible:

- specify different new HOL datatypes depending on the example at hand; or
- define the full grammar of MDG-HDL in HOL.

In the first approach, no embedding of the grammar is required. It relies on predefined HOL types. For each application, the user has to implement the appropriate types, which is not practical and requires a certain user expertise with HOL. Furthermore, the complexity of the HOL specification will increase when the design grows. The fact of homogenising types for MDG table entries will also introduce ambiguities. In the second approach, we need to embed in HOL the full MDG-HDL grammar. This, however, offers a format on the way to write the MDG types, variables, functions and terms. We experimented both alternatives, and realized that the absence of formatting in the first approach will lead to cumbersome specifications. We hence adopted the second alternative.

To embed the grammar of the MDG-HDL language in HOL, it is necessary to cover the syntax of many-sorted first-order logic. An MDG sort can be either abstract or concrete. In HOL, we define an abstract sort to be of type $\alpha$ to string. The second parameter in this definition is specified mainly to permit the user to impose a specific MDG sort. A concrete sort (Boolean sort included) is defined by the list of its enumerated values.

The HOL definition of the MDG sort is:

$$\vdash_{def} MDG\_sort = ABSTRACT\ of\ 'a \quad string \mid CONCRETE\ of\ string \quad string\ list$$

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

$$\vdash_{def} (IsConcreteSort\ (ABSTRACT\ Abs\ MDG\_name) = F) \wedge (IsConcreteSort\ (CONCRETE\ Conc\ val\_list) = T)$$

*IsConcreteSort* returns true if the type is of concrete sort. Similarly, we define a predicate to test if the type is of the abstract sort. The variables are defined according to their sorts. A function is defined by its domain, which is a list of concrete variables, abstract variables or a mixture of both, and its range, which is a unique output. The type of the function is determined according to its domain and range. If the output is from an abstract sort, so the function is defined to be an abstract function. If all the inputs and the output are from concrete sort, so the function is defined to be concrete. And finally, if the output of the function is concrete, and at least one of its inputs is abstract, the function is defined to be cross-function.

$$\vdash_{def} MDG\_Fun = MDG\_FUN\ of\ string \quad MDG\_VAR\ list \quad MDG\_VAR$$

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

$$\vdash_{def} ConcreteVarList(h::tl) = ((IsConcreteVar\ h) \wedge (ConcreteVarList\ tl)) \wedge (ConcreteVarList\ [\,] = T)$$

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

$$\vdash_{def} concreteFunc\ (MDG\_FUN\ name\ InputVarList\ OutputVar) = (ConcreteVarList\ InputVarList) \wedge (IsConcreteVariable\ OutputVar)$$

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

- a concrete constant, *CONC_Const*, one of the concrete sort enumeration;
- a generic constant, *GE_Const*, a constant defined for an abstract sort;
- a variable, *VAR_Term*, either from a concrete sort or an abstract sort; or
- a composed term.

The latter is done using the constructor TERM. It takes as argument a defined MDG Term and returns an MDG Term The obtained HOL definition is:

$\vdash_{def}$ *MDG_term = GE_Const of 'a*
*| CONC_Const of string*
*| VAR_Term of MDG_VAR*
*| FN_Term of MDG_Fun*
*| TERM of MDG_term MDG_term*

For the MDG tables, we kept the same defined structure as in [5], where we impose that the table entry should be either a don't care value or an MDG term.

$\vdash_{def}$ *Table_Val = TABLE_VAL of 'a MDG_term*
*| DONT_CARE*

In addition to the embedding of the grammar, we added to the theory developed in [5] some abstract components that previously were defined only for the Boolean type, such as the multiplexer or the register. These components have abstract inputs and abstract output. For example, the multiplexer component is defined as follow:

$\vdash_{def}$ *mdg_mux* $x_1$ $x_2$ *(y:num bool) z =*
$\forall t.\ (z\ t) = if\ (y\ t)\ then\ (x_2\ t)\ else\ (x_1 t)$

Another example is the one of the MDG *transform* component. An MDG *transform* is a black-box component that implements any function.

$\vdash_{def}$ *mdg_transform* $x_1\ x_2 = \ \forall t.\ \exists y.\ (x_2\ t) = y\ (x_1\ t)$

As an application of our embedding, we propose in the next section the example of an abstract counter. Its specification and implementation are written in HOL according to the embedded theory. A proof to verify the equivalence of both is conducted in HOL.

### 3.3 Abstract-Counter Example

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

$$R \equiv [(y = \underline{0}) \wedge count' = inc\ (count)] \vee$$
$$[(y = \underline{1}) \wedge eqz\ (count) = \underline{0} \wedge count' = dec(count)] \vee$$
$$[(y = \underline{1}) \wedge eqz\ (count) = \underline{1} \wedge count' = count]$$

*The HOL Specification*
In this example, we have an abstract sort, two abstract functions and a cross-function. First, we define an abstract sort 'abs'. Second, we specify the variables derived from it, and finally we declare the different functions. Furthermore, we add the definition of the generic constants derived from the abstract sort 'abs': *abs_val*. In HOL, the output of the tables should be defined as signals. Hence, we define them as functions of time[1], giving the series of values of the signal over time:

$\vdash_{def}$ *absSIG* $= \lambda\ (t:num).\ abs\_val$
$\vdash_{def}$ *decSIG* $= \lambda\ (t:num).\ dec\_abs\_val$
$\vdash_{def}$ *incSIG* $= .\ \lambda\ (t:num).\ inc\_abs\_val$
$\vdash_{def}$ *dec_incSIG* $= \lambda\ (t:num).\ abs\_val$

In terms of a truth table, the counter behavior is given in Table 1; where *n_count* represents *count'*, and *eqz* represents *eqz(count)*:

**Table 1. Abstract-Counter Behavior**

| count | eqz | y | n_count |
|---|---|---|---|
| abs_val | * | F | inc_abs_val |
| inc_abs_val | F | T | abs_val |
| abs_val | T | T | abs_val |
| abs_val | F | T | dec_abs_val |

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

$\vdash_{def}$ *bool_to_MDGTerm:( bool string MDG_term) b =*
*if (b = T) then (CONC_Const "T")*
*else (CONC_Const "F")*

In HOL terms, the counter specification is equivalent to:

$\vdash_{def}$ *COUNTER_TAB (count) (v:(num bool))*
*(y:(num bool))(n_count) =*

*[TABLE ; bool_to_MDGTerm o v; bool_to_MDGTerm o y]*
*(n_state o SUC)*

*[ [ TABLE_VAL (abs_val);DONT_CARE;*
*TABLE_VAL (CONC_Const "F" ) ];*

*[TABLE_VAL(inc_abs_val);TABLE_VAL(CONC_Const"F*
*");TABLE_VAL (CONC_Const"T")];*

*[ TABLE_VAL (abs_val);TABLE_VAL(CONC_Const"F");*
*TABLE_VAL(CONC_Const"T") ];*

[1] $\lambda$ t. x means that x is function of t.

*[ TABLE_VAL (abs_val);TABLE_VAL(CONC_Const"T");*
*TABLE_VAL(CONC_Const"T") ] ]*

*[ incSIG;dec_incSIG;decSIG;absSIG ] absSIG*

The first four rows represent the possible inputs combination. The fourth list represents the output for each row respectively. And finally, the *absSIG* is the default value of the output if the input sequence is different from what is specified.

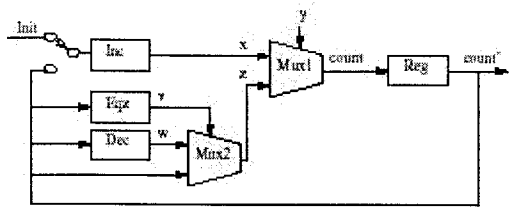



**Figure2 : Abstract Counter Implementation**

*The HOL Implementation*
The implementation of the abstract counter is given in Figure 2. It is composed of two multiplexers, one register for the next state *n_count*, two black-box (uninterpreted) functions *Dec* and *Inc*, and finally one MDG transform block for the cross function *eqz*. We initialised the *count* value using an initial predicate that sets the variable count at *abs_val* (c.f Figure2).

*⊢ def COUNTER_IMP (v) (y:num bool) (n_count) =*
*∃x z w (count:num string MDG_term) count1.*

*(Reg count1 n_count) ∧*
*(Mux1 x z (y:num bool) count1 ) ∧*
*(Inc n_count x) ∧*
*(Mux2 n_count w (v) z) ∧*
*(Dec ( n_count) (w)) ∧*
*(Eqz ( n_count) (v)) ∧*
*(Initial count)*

The next objective is to verify in HOL that the implementation of the counter implies its specification. The proof should be done for any generic constant our counter takes. The goal has an implication pattern specifying that the implementation we propose implies the specification:

*⊢ def ∀ count v y n_count abs_val .*
*COUNTER_IMP v y n_count ⇒ COUNTER_TAB count v y n_count*

The proof was conducted such that all definitions are first rewritten, and then using a combination of two predefined HOL tactics: ARITH_TAC and PROVE_TAC. The First one is used to split the goal to several subgoals. After that each subgoal is proven individually. The original goal is proven when all subgoals are proven.

## 4. $L_{MDG}$ EMBEDDING IN HOL

### 4.1 $L_{MDG}$

$L_{MDG}$ [17] is the properties input language of the MDG model checker. It represents a subset of the Abstract CTL* language [17]. This logic is used to specify properties for the computation model. Only universal path quantification is possible with the current version of MDG model checker. The properties allowed in $L_{MDG}$ can have the following templates:

*Property :*
***A** (Next_let_formula)*
*/**AG** (Next_let_formula)*
*/**AF** (Next_let_formula)*
*/**A** ((Next_let_formula) **U** (Next_let_formula))*
*/**AG** ((Next_let_formula) ⇒ **F** (Next_let_formula))*
*/**AG** ((Next_let_formula) ⇒ ((Next_let_formula) **U***
*(Next_let_formula)))*

where the *Next_Let_Formula* is defined to be a nesting formula, or a basic formula. The latter represents the equality between the different kind of state variables, conjunction and disjunction of terms. We present in the next section the embedding of this language in HOL allowing us to express temporal properties in the theorem prover.

### 4.2 $L_{MDG}$ Embedding

In order to embed $L_{MDG}$ in HOL, it is important to respect the semantics of the original language. Generally, properties are defined according to two notions: *path* and *state*. A *path* is a sequence of states, while a *state* is an assignment to the set of state, input and output variables. A full path starting from a state $s_i$ is denoted by:

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

All formulae in $L_{MDG}$ are path formulae. Hence, given a property in $L_{MDG}$ for a model under a given interpretation $\psi$, the property holds on the model if and only if the property is true for all paths starting from each initial state.
We write $\pi_i,\sigma \models p$ to means that a path formula $p$ is true at path $\pi_i$ according to a model $\sigma$. Hence, the semantics of the **F** operator will be:

$$(\pi,\sigma) \models F_p \text{ iff } (\pi_j, \sigma) \models p \text{ for some } j \geq i$$

Since $L_{MDG}$ is an Abstract CTL* like language [17], we divide the properties in two classes: the first is the CTL* [10] like properties and the second is the LTL [10] like properties. For the latter ones, we define a property according to the predicate we want to verify. Each logical proposition is a function of the path, expressed here by *s*

which can be formulated as a history function keeping a trace of the states on the path.
Since $L_{MDG}$ is an Abstract CTL* like language [17], we divide the properties in two classes: the first is the CTL* [10] like properties and the second is the LTL [10] like properties. For the latter ones, we define a property according to the predicate we want to verify. Each logical proposition is a function of the path, expressed here by *s* which can be formulated as a history function keeping a trace of the states on the path.

$$\vdash_{def} LMDG\_F\ p\ s = \exists\ p\ s\ t$$

For each temporal operator, a HOL definition is set. Besides, the conjunction, disjunction, and implication of predicates are defined as functions of the proposition.

$$\vdash_{def} LMDG\_CONJ\ p1\ p2\ s\ t = (p1\ s\ t) \vee (p2\ s\ t)$$
$$\vdash_{def} LMDG\_DISJ\ p1\ p2\ s\ t = (p1\ s\ t) \wedge (p2\ s\ t)$$
$$\vdash_{def} LMDG\_IMP\ p1\ p2\ s\ t = \neg(p1\ s\ t) \vee (p2\ s\ t)$$

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

$$\vdash_{def} LMDG\_AG\ R\ p = \forall\ ((R\ s) \wedge (\forall\ (p\ s\ t)))$$
$$\vdash_{def} LMDG\_AF\ R\ p = \forall\ ((R\ s) \wedge (\forall\ (p\ s\ t)))$$

Introducing the notion of circuit in the property denotes the fact that each path considered belongs to the design description we have. In addition, when verifying a hierarchical design, this feature raises the possibility of checking a property of a specific sub-block since the property itself is defined according to this sub-block. We specify a HOL definition for each CTL* operator in the $L_{MDG}$ language. To get the appropriate template for the MDG tool, the combination of these formulae is done by the user. Templates that do not exist in the $L_{MDG}$ language hence can be expressed. As en example, the property:

**AF** ((Next_let_formula) **U** (Next_let_formula))

can be defined in HOL while it does not belong to the syntax of $L_{MDG}$. Therefore, the embedding we propose is more expressive than the original one.

### 4.3 Abstract Counter Example

To illustrate the proposed $L_{MDG}$ embedding, we present here a liveness property of the abstract counter example used in section 3.3.
According to the specification of the counter, when the control signal *y* is equal to *false*, independently from the value of the cross-function *eqz*, the counter must increment the input. Using the MDG syntax, this property is expressed as follows:

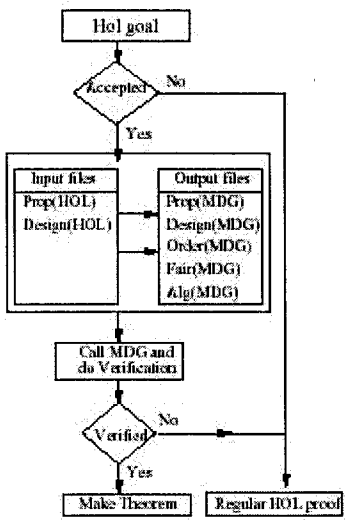
$$AG((y = 0) \Rightarrow (X(count = inc(count))))$$

Where X represents the next temporal operator.
In terms of the embedded theory in HOL, the property corresponds to:

$$\vdash_{def} LMDG\_AG\ (COUNTER\_TAB\ count) \wedge$$
$$(LMDG\_IMP$$
$$(LMDG\_CONJ\ (\lambda count\ (t: num).\ y = false)$$
$$(\lambda count\ (t: num).\ count\ t = abs\_val))$$
$$(LMDG\_X\ (\lambda count\ t).\ n\_count\ t = inc\_abs\_val))$$

## 5. APPLICATION: HOL-MDG HYBRID TOOL

Having the embedding of the input languages of MDG in HOL raises the possibility of passing HOL goals out to MDG. To this end, we developed a hybrid tool, HOL-MDG, around the embedding. Its verification procedure is depicted in Figure 3. During the verification procedure, the user starts by giving the HOL specification, the HOL property and the goal he wants to prove. If this goal fits the required pattern, the respective MDG files are generated. The latter are sent to MDG for the model checking verification. If the property holds, a HOL theorem is set. However, if the verification within MDG tool fails, we have to develop the proof interactively within HOL.




**Figue3 : HOL-MDG Verification Procedure**

Obviously, the tool would not accept any arbitrary HOL specification. It will only accept MDG-styles specifications and properties. We make use of the embedded HOL theories to express both the description model, and the properties.

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

$$/\text{-}_{\text{def}}\ Model \Rightarrow Property$$

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 [16]:

$$/\text{-}_{\text{def}}\ FormalisedMDGresult \Rightarrow Model \Rightarrow Property$$

The general conversion theorem into HOL has been proved in [16]. So, the theorem can be instantiated for any design and any property under consideration.

## 6. CONCLUSION

We have described in this paper an embedding of the MDG input languages in HOL: the MDG-HDL grammar and the $L_{MDG}$ temporal logic. The embedding of the former is achieved by covering the syntax of the many-sorted first-order logic. It allowed us to extend the tool in [9] with abstract types providing a way to write higher-level specification and pass them to the MDG model checker. While the embedding of $L_{MDG}$ is achieved by doing a mapping one to one to all the temporal operators, both LTL and CTL*, used in the language. This allowed us to define $L_{MDG}$ like properties in HOL. As an application of these embeddings, we presented a hybrid tool linking HOL and MDG model checker. The hybrid tool takes as input specifications and properties written in MDG-HDL and $L_{MDG}$ languages, respectively based on the developed embeddings in HOL.


## ACKNOWLEDGEMENTS

We would like to express our gratitude to Dr. Ait-Mohamed from Concordia University for his precious hints and discussions during this project.


## REFERENCES


[1] 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.

[2] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. Van Tassel. Experience with Embedding Hardware Description Languages in HOL. In Theorem Provers in Circuit Design, pages 129-156, North-Holland, 1992.

[3] 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.

[4] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A Tutorial: Introduction to PVS. SRI International, April 1995.

[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, Canberra, Australia, pages 31-45, September 1998.

[6] L. A. Dennis, G. Collins, M. Norrish, R, Boulton, K. Slind, G. Robinson, M. Gordon, and T. Melham. The PROSPER Toolkit. In proceedings of the sixth International Conference on Tools and Algorithms for the Construction and Analysis of Systems, LNCS 1785, pages 78-92, Springer Verlag, 2000.

[7] M. Gordon. The Semantic Challenge of Verilog HDL. In the 10th Annual IEEE Symposium on Logic in Computer Science, San Diego, California, pages 26-29, June 1995.

[8] M. Gordon. Using HOL to study Sugar 2.0 Semantics. NASA Conference Proceedings CP-2002-211736, 2002.

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

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

[11] T. Kropf and R. Reetz. Simplifying Deep Embedding: A Formalised Code Generator. In Higher Order Logic Theorem Proving and its Applications, LNCS 859, pages 378-390, Springer Verlag, 1995.

[12] K. L. McMilian. Symbolic Model Checking. Kluwer, 1993.

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

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

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

[16] H. Xiong, P. Curzon, and S. Tahar. Importing MDG Verification Results into HOL Theorem Proving in Higher Order Logics, LNCS 1690, pages 293-310, Springer Ve rlag, 1999.

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

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