Interfacing ASM with the MDG Tool

Amjad Gawanmeh[1], Sofiène Tahar[1], and Kirsten Winter[2]

[1] Concordia University, Montreal, Quebec, H3G 1M8 Canada
Email:{amjad,tahar}@ece.concordia.ca
[2] SVRC, University of Queensland, Australia
Email:kirsten@svrc.uq.edu.au

Abstract. In this paper we describe an approach to interface Abstract State Machines (ASM) with Multiway Decision Graphs (MDG) to enable tool support for the formal verification of ASM descriptions. ASM is a specification method for software and hardware providing a powerful means of modeling various kinds of systems. MDGs are decision diagrams based on abstract representation of data and are used primarily for modeling hardware systems. The notions of ASM and MDG are hence closely related to each other, making it appealing to link these two concepts. The proposed interface between ASM and MDG uses two steps: first, the ASM model is transformed into a flat, simple transition system as an intermediate model. Second, this intermediate model is transformed into the syntax of the input language of the MDG tool, MDG-HDL. We have successfully applied this transformation scheme on a case study, the Island Tunnel Controller, where we automatically generated the corresponding MDG-HDL models from ASM specifications.

1 Introduction

With the increasing reliance on digital systems, errors in their design can cause failures, resulting in the loss of time, money, and a long design cycle. Large amounts of effort are required to correct an error, especially when the error is discovered late in the design process. For these reasons, we need approaches that enable us to discover errors and validate designs as early as possible. Conventionally, simulation has been the main debugging technique. However, due to the increasing complexity of digital systems, it is becoming impossible to simulate large designs adequately. Therefore, there has been a recent surge of interest in formal verification and tool support for this task, such as theorem proving, combinational and sequential equivalence checking, and model checking [11]. These approaches vary in the degree of interaction that is required by the user. Of particular interest are those tools that run automatically and do not require any special knowledge about the formal techniques that are applied. Equivalence and model checkers belong to this category.

ASM (Abstract State Machines) [8] is a specification method for software and hardware modeling and provide a powerful means of modeling various kinds of systems. MDGs (Multiway Decision Graphs) [3] are decision diagrams based on abstract representation of data and are used for modeling hardware systems in first place. The MDG tool provides equivalence checking and model checking applications based on MDG. The given modeling language is the hardware description language MDG-HDL [20]. In this work, we propose to build an interface from ASM to the MDG tool in order to enable the formal verification of ASM descriptions using automated tools. This interface serves two purposes, depending on the perspective: firstly, it provides tool support for automated verification of ASM models. Secondly, it provides the MDG tool with a high-level modeling language, namely ASM, which allows us to model a wide range of applications in an understandable and succinct manner.

Our work is based on earlier work on an interface from ASM to the model checker SMV [6, 17] and also from ASM to MDG [17]. In contrast to the SMV model checker, the MDG tool provides a useful means for representing abstract models containing uninterpreted functions, where SMV supports neither abstract data types nor uninterpreted functions. This allows model checking on an abstract level at which the state explosion problem can in some cases be avoided. Although the latter approach in [17] already provides support for abstraction by exploiting the MDG data structure, it does not provide an interface to the actual MDG tool. Our work provides a tool that transforms ASM-SL, the modeling language of the ASM Workbench (ASM-WB) [5], into MDG-HDL code which can be input into the given MDG tool.

Other work on model checking ASM specifications include the complexity analysis of by Spielmann [14] and the work on real-time systems by Beauquier and Slissenko [2], which provide very useful theoretical results. Moreover, an interface to the model checker SVE is suggested by Plonka [13]. These results are complemented by our work since the MDG tool facilitates the handling of functions over abstract domains and ranges. ¿From a more general perspective, the work described by Shankar [15] and Katz and Grumberg [9] is also related in that they provide a very general tool framework comprising a general intermediate language which allows one to interface a high-level modeling language with a variety of tools. In [10], Kort *et al.* propose a hybrid verification tool linking MDG and the HOL theorem prover, making it possible to delegate the verification of HOL subgoals to the MDG tool for automatic proof. Our work results in an ASM-specific solution which extends the interface framework of [6, 17] as well as [10] with another tool interface.

2 Abstract State Machines

Abstract State Machines (ASM) [8] is a specification method for software and hardware modeling. The system is modeled by a set of states and transition rules which specifies the behavior of the system. Transition rules specify possible state changes according to a certain condition. The notation of ASM is efficient for modeling a wide range of systems and algorithms as the number of case studies demonstrates [8].

An ASM model consists of states and transition rules. States are given as many sorted first-order structures, and are usually described in terms of functions. The ASM notation includes *static functions* and *dynamic functions.* Static functions have the same interpretation in every state, while dynamic functions may change their interpretation during a run. There are also *external functions* which cannot be changed by the system itself, but by the outer environment.

Dynamic as well as static components of the system can be described within the same ASM model. We can also have behavioral (specification) as well as a structural description (implementation) for the same system. This is the typical way for modeling hardware designs. A behavioral description is a higher level model of the system, we use *if-then-else* rules and *dynamic functions* to describe the system behavior. On the other hand, a structural description is a lower-level model in which we use *static functions* to define hardware primitives, such as logic gates. From these primitives we build a hierarchical or modular structure of the system.

The ASM Workbench (ASM-WB) [5] provides a number of basic functions including: parsing, type checking, pretty printing and evaluation of terms and rules. It supports computer aided specification, analysis and validation of ASM models. The ASM-WB supports the ASM specification language (ASM-SL) which is a strongly typed version of ASM. The ASM-WB is designed as an extensible tool, to which different tool interfaces can be added. One interface with the SMV model checker was suggested and implemented by [6]. Since our work is built on preliminary work [17], we can furthermore exploit the notion of *abstract types.* This concept provides a simple means for abstraction in order to abstract from infinite data types. This feature can be essential when applying automated verification techniques like model checking.

3 Multiway Decision Graphs

Multiway Decision Graphs (MDGs) [4] is a relatively new class of decision diagrams which subsumes the traditional ROBDDs (Reduced Order Binary Decision Diagrams) [11] while allowing abstract data sorts and uninterpreted function symbols. MDGs are based on a subset of many-sorted first-order logic, with a distinction between *abstract* and *concrete* sorts (including the Boolean sort). Concrete sorts have *enumeration* while abstract sorts do not. The enumeration of a concrete sort is a set of distinct constants occurring in the enumeration and referred to as *individual constants.* Constants of abstract sort are called *generic constants* and could be viewed as 0-ary function symbols. 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 $\alpha_1 \ldots \alpha_n$ 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 explicit definition; they can be eliminated and do not appear in MDG. Abstract function symbols and cross-operators are *uninterpreted.*

An MDG is a finite, directed acyclic graph (DAG). An internal node of an MDG can be a variable of concrete sort with its edge labels being the individual constants in the enumeration of the sort; or it can be a variable of abstract sort and its edges are labelled by abstract terms of the same sort; or 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 an MDG are true formulae. Thus, MDGs essentially represent relations rather than functions. MDGs can also represent sets of states.

3.1 System Modeling using MDG

Logic gates can be represented by MDGs similarly to ROBDDs, because all inputs and outputs are of Boolean type. Figure 1 shows the MDG for an AND gate for a given variable order. A design description on the register transfer level (RTL), however, involves the use of more complex functions and data inputs that go beyond the capacity of ROBDDs. For example, Figure 2 shows the MDG of an arithmetic logic unit (ALU), where *op* is a concrete variable with enumeration sort {0,1,3,4}, *x1, x2* and *y* are abstract variables, *zero* is a generic (abstract) constant of the same sort, and *sub, add* and *inc* are uninterpreted functions.

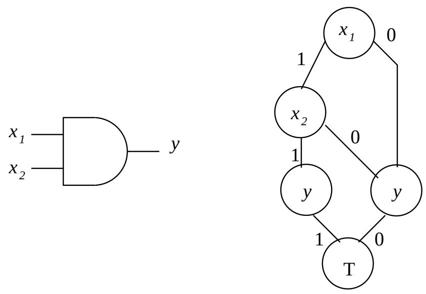

Fig. 1. MDG for an AND gate

To describe hardware circuits, the MDG tool provides a Prolog-style hardware description language called MDG-HDL [20]. It allows the use of abstract as well as concrete variables for representing data operations. A circuit can be described on the structural (implementation) or behavioral (specification) level. Often models on both levels of abstraction are given and shown to have equivalent behavior (e.g., by means of equivalence checking). A structural description is a collection of components connected by signals that can be of abstract or concrete type. MDG-HDL includes a library of predefined components that represent logic gates such as AND, OR, multiplexers, registers, drivers, etc. There is also a component that represents functions as a black box called *transform.* It is used for uninterpreted functions, or cross-terms.

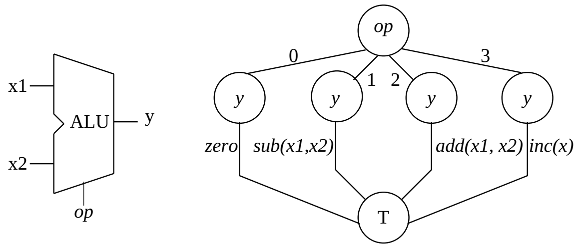

Fig. 2. MDG for an ALU

The behavioral description is represented by means of abstract descriptions of state machines [3] defined in MDG-HDL in terms of *tables*. An MDG table is similar to a truth table, but it allows first-order terms as entries in addition to concrete variables. Tables usually describe the transition, the output relation, or the combinational functionality of the system. They contain a list of rows where the first row contains variables and cross-terms. Variables must be concrete except for the last element which can be abstract. This last element provides the resulting value of the function or transition. All other rows except the last one must contain individual constants in the enumeration of their corresponding variable sort, or the "*" which symbolizes a "don't care" value. The last element can be a constant value or a first-order term. Figure 3 shows a tabular description of a simple state machine, with its MDG representation, where x is a Boolean input, a is an abstract state variable and a' is its next state variable. It performs *inc* operation when $x = 1$, where *inc* is an uninterpreted function symbol.

x	a'
0	*a*
1	*inc (a)*

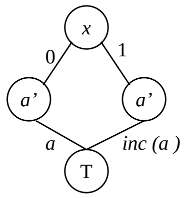

Fig. 3. Table and MDG for a simple behavioral state machine

3.2 Verification using the MDG Tool

For the formal verification of finite state systems (machines), the MDG tool provides a set of application procedures for combinational and sequential equiv-

[3] In the MDG literature [3] such a finite state machine (FSM) is called abstract state machine (ASM) which is obtained by letting some data input, state or output variables of an FSM be of abstract sort, and the data operations be uninterpreted function symbols.

alence checking [4], invariant checking [4] and model checking [18]. The MDG tool has been used to verify a number of non-trivial systems such as communication switches and protocols [1, 3, 16, 19, 21].

Figure 4 summarizes the MDG tool applications. In order to verify designs with this tool, we need first to specify the design in MDG-HDL in terms of a behavioral and/or a structural description (design specification and design implementation in Figure 4). Moreover, an algebraic specification is to be given to declare sorts, function types, and generic constants that are used in the MDG-HDL description. Rewrite rules that are needed to interpret function symbols should be provided here as well. Like for ROBDDs, a symbol order according to which the MDG is built should be provided by the user. This symbol order can affect critically the size of the generated MDG. While the current version of MDG uses manual static ordering, a newer version will be released soon including automatic dynamic ordering [7].

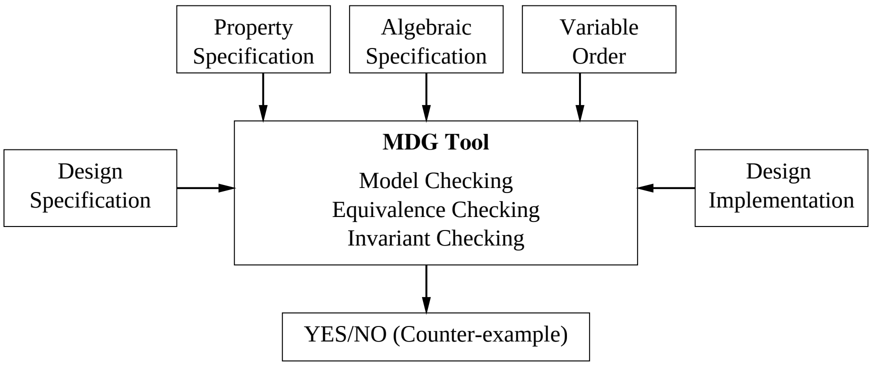

Fig. 4. MDG verification tool

4 Interfacing ASM with the MDG Tool

In order to provide a generic interface for the ASM-WB with different tools, ASM models are automatically translated into the ASM-IL [17]. An ASM-IL representation is a flat, simple transition system, which means that all nested rules in the ASM model are flattened and all complex data structures are unfolded. Based on this ASM-IL, we build an interface to the MDG tool. The disadvantage of a flattened representation in ASM-IL, however, is the fact that it does not preserve the structure of the original ASM model, because it provides no modular or hierarchical descriptions.

In ASM-IL, (*locations* are identified with state variables by mapping each location to a unique variable name. (*Guards* are mapped into simple Boolean terms. Thus, an ASM model is represented by a set of guarded updates in a triplet form (*loc, guard, val*). All nested rules are flattened then mapped into simple guarded updates using a simplification function. Each term that occurs in an ASM rule is simplified until the result contains only constants, locations and variables. Abstract functions and cross-operators are left uninterpreted in ASM-IL. Only cross-operators that match one of the standard relational operators are mapped into a cross term.

Starting from the ASM-IL language, we built our interface, as shown in Figure 5, in four steps. Each step automatically generates one of the tool inputs: MDG-HDL design behavior and/or MDG-HDL design structure, algebraic specification, and symbol (variable) order. These steps are described in the subsections to follow. The interface was implemented and then tested with a case study.

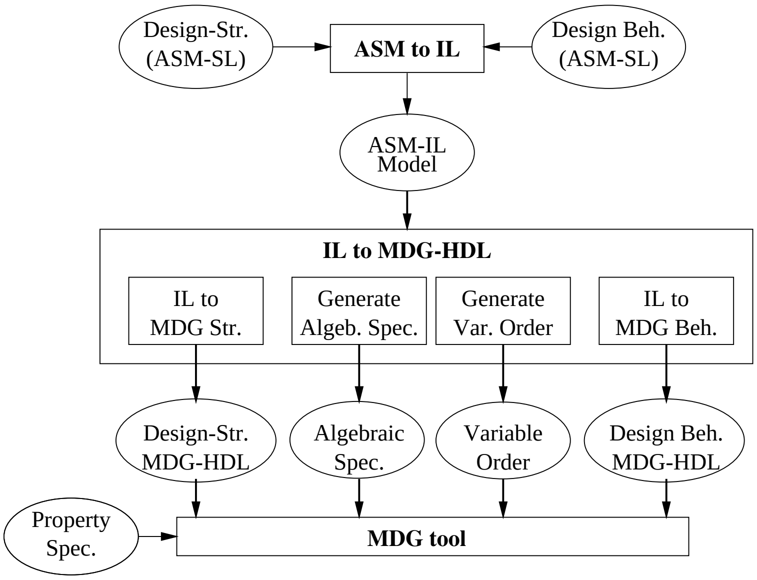

Fig. 5. ASM-MDG interface

4.1 MDG-HDL Structural Description

To build an MDG-HDL structural description from an ASM-IL model, we map locations, guards, and values into MDG-HDL components preserving the functionality of the model. Uninterpreted functions and abstract sorts in the original model are left unchanged.

Locations. In an ASM-IL representation each location is associated with a set of guarded updates, each consisting of a Boolean guard and an update value. The whole expression evaluates into one value, $value_i$, which is the next state value of the dynamic location, if there is at least one guard satisfied. Otherwise, the value of the location will be the same in the next state. Figure 6 shows the interpretation of guarded updates.

(*loc*, ($guard_0$, $value_0$) ($guard_1$, $value_1$) .... ($guard_n$, $value_n$))	if $guard_0$ then $loc =$ $value_0$ if $guard_1$ then $loc =$ $value_1$ .... if $guard_n$ then $loc =$ $value_n$

Fig. 6. ASM-IL guarded updates

First, each location is mapped into a state component in MDG-HDL. Since locations are considered state variables, we represent locations as registers. A register has two signals, an input and an output signal. Each location name is mapped into a signal that is connected to the register's output. The result value of the location is mapped to a signal that is connected to the register's input. This will generate a state machine (sequential circuit) in which the number of state variables is equal to the number of updated locations in the model.

For guards and values, we build a set of MDG-HDL components that are interconnected with signals that evaluate to the next state value of the location: Each pair (guard, value) is mapped into a multiplexer where the guard is the control and the value is one input. We connect these multiplexers together in a hierarchical way as shown in Figure 7. This output is connected into the input of the state element representing the location. The location is fed back into the last multiplexer in the hierarchy to represent the case in which no guard is satisfied.

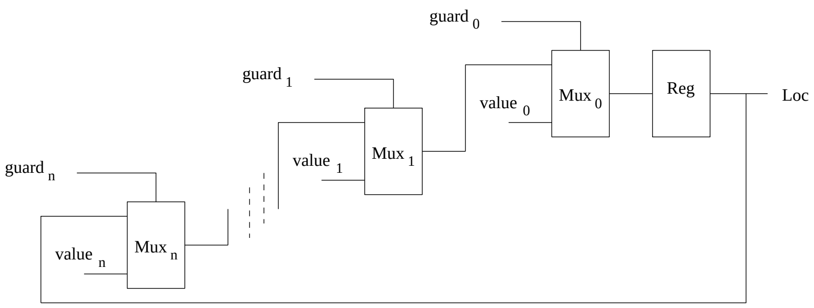

Fig. 7. Mapping ASM-IL expression for one location into MDG-HDL

Values. Values can be locations, constants, or any variable. If a value is a location or a variable, we map it directly into a signal with the same type. Constants are mapped into an MDG-HDL component called *constant_signal*, that is a component with an individual value from the enumeration of the concrete data sort or a generic constant in case of an abstract type. For Boolean types we use two default components that have always the values of 0 and 1.

Guards. A guard is a Boolean ASM-IL expression. It might contain concrete functions, uninterpreted functions, or cross-terms. Concrete functions can be default Boolean operators or any other function. We map these operators into MDG-HDL components that perform the same functionality. We apply the mapping function on each rule by creating an MDG-HDL component, then mapping the same function again on its parameters until we get a constant value, or a variable.

All default binary operators are mapped into MDG-HDL logic gates. An equality expression for a variable and the value **True** are simply mapped into a signal with the variable name. Equality expressions for a variable and the value **False** are mapped into the corresponding negated MDG-HDL component, **NOT**. Relational operators, such as $>$, $>=$, $<$, $<=$, etc., are mapped into a *transform* component that can be viewed as a black-box. All other cross-terms, abstract functions, and uninterpreted functions are also mapped into *transform* components.

4.2 MDG-HDL Behavioral Description

MDG-HDL behavioral descriptions are represented by a *table* similar to a truth table. For each location in the ASM model, we generate one table. The first row of the table contains all variables in the model and any cross-term or function that occur in the ASM-IL guarded update expression of that location. The last element is the location itself. Then we treat the list of (*guard, value*) pairs one by one (see Figure 8). An expression with one variable in the guard is mapped into one row with all other variables are set to the "don't care" ("*") symbol. A conjunction is mapped into one row with each variable or cross-term is assigned its value (val_i), or "don't care" if it does not occur in the expressions. The result *value* is assigned to the last element in the row, which gives the valuation of the location. A disjunction is mapped into as many rows as the number of variables and cross-terms in the expression. In each row, a value is assigned to the corresponding variable, all others are "don't care" values. The last element of each of these rows contains the value of the location as shown in Figure 8.

4.3 Algebraic Specification

We have to declare all sorts and functions before we use them in our MDG-HDL design descriptions. In the MDG tool, there is a default abstract sort *wordn* (for n-bit words) and a default concrete sort *bool* with the enumeration of [0,1]. Any other sorts must be explicitly declared. An ASM-IL representation preserves the enumeration for each variable. Based on this, we declare a concrete sort for each

(("=", var_i , val_i), val)	[*, *, val_i , *, val]
conj (("=", var_1 , val_1), ("=", var_2 , val_2), **...** , val)	[val_1 , val_2 , ... , val]
disj (("=", var_1 , val_1), ("=", var_2 , val_2), **...** , val)	[val_1 , *, *, ... , val], [*, val_2 , *, ... , val], ... ,

Fig. 8. Creating MDG tables rows from guarded updates

different enumeration. Since different abstract sorts can be distinguished in the ASM-SL model, we declare different abstract sorts for each one.

All functions and cross-terms are also declared in the algebraic specification in the same way. This includes uninterpreted functions and cross-terms. We declare any function that occurs in the ASM-IL expressions in the algebraic specification according to its arguments and domain sorts. These sorts are derived from the expressions where the function occurs. Figure 9 below illustrates the mapping function that we use to generate the MDG algebraic specifications.

[var , enum (abstract ($sort$))]	abs_sort($sort$)
[var , enum (constant (val_1 , val_2 , ... , val_n))]	conc_sort (var_sort [val_1 , var_2 , ... , var_n])
[cross_oper (var_1 , var_2)]	function([$sort_1$, $sort_2$] , bool)
[fun (var_1 , var_2 , ... , var_n)]	function([$sort_1$, $sort_2$, ... , $sort_n$] , bool)

Fig. 9. Declarations of functions and sorts in the algebraic specifications

4.4 Variable Order

MDGs have some restrictions on the order of abstract variables and cross-operators. In order to obey these restrictions, we explore all functions and cross-operators in the ASM-IL expressions and order the variables according to the dependencies between abstract variables themselves and also between abstract variables and cross-terms or functions. If a variable *var1* depends on another variable (or function) *var2*, then *var1* is sorted above *var2* in the order file. Also if a cross-term *f* depends on a variable *var1*, then *var1* should appear above *f*.

5 Case Study: Island Tunnel Controller

In this section, we provide a case study based on the Island Tunnel Controller (ITC) example [19] in order to illustrate the proposed ASM-MDG interface. The ITC is used to control two traffic lights for a tunnel that connects an island to the mainland as shown in Figure 10. There are four sensors to detect vehicles at both sides of the tunnel: *ie* at the entrance of the island (*carEntering* in ASM), *ix* at the exit of the island (*carExiting* in ASM), *me* at the entrance of the mainland and *mx* at the exit of the mainland. There are four output signals to control the traffic lights at both sides: *irl* for island red light, *igl* for island green light, *mrl* for mainland red light and *mgl* for mainland green light. The ITC is specified using three communicating controllers and two counters as shown in Figure 10.

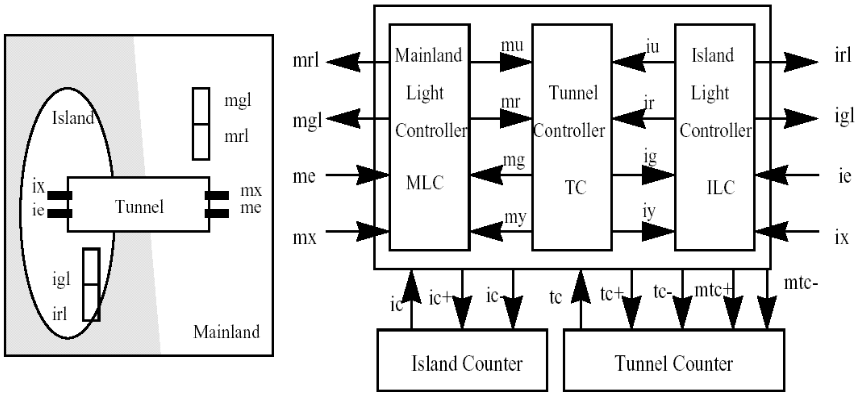

Fig. 10. Island Tunnel Controller

The tunnel allows a maximum number of cars to be in the island at one time. In ASM this number can be taken as a parameter of an abstract type that represents any natural number. We can then define a cross-operator for the operation "$islandCounter < n$". This allows the modeling of the controller for any number of cars. The advantage of modeling designs in ASM over MDG-HDL might not be clear in this hardware example since MDG-HDL is a hardware oriented language. However, from our experience it is easier and more convenient to model the controller behavior in ASM rather than to produce the MDG-HDL table representation, especially for large systems. Moreover, this example nicely illustrates the advantage of using abstract types that are supported by our framework. In contrast to other modeling and verification techniques, we are able to verify this system for any arbitrary counter size. We choose the Island Light Controller (ILC) as part of the design to develop two models in ASM: specification and implementation[4]. Figure 11 shows the state transition diagram for the ILC. We model this in ASM by defining a free type that represents the states of the ILC as follows:

[4] The full specification models in ASM as well as the generated MDG-HDL models can be obtained at: http://hvg.ece.concordia.ca/Tools/ASMMDG/ITC/

```
freetype IS_SORT == {green, red, exiting, entering}
```

Increment and decrement operations on the counters are generally infinite mappings over integers. In our model, we specify those as abstract static functions, which maps between abstract values. These functions are left uninterpreted in our transformation.

External functions are used to represent environment operations for detecting vehicles at the entrance or exit in addition to signals from the Tunnel Controller, e.g. `external function carEntering: BOOL`

We describe the dynamic control of the controller states using the dynamic function: *islandState*.

```
dynamic function islandState: IS_SORT initially red
```

All Boolean outputs of this controller are also described by *dynamic functions*, e.g. `dynamic function itc_min: BOOL initially undef` which declares the island tunnel counter. We then describe the behavior of the system using *if-then-else* rules. One example is shown below for the *entering* state. Note that the value of *carEntering* is indicated by the sensor *ie* (see Figures 10 and 11).

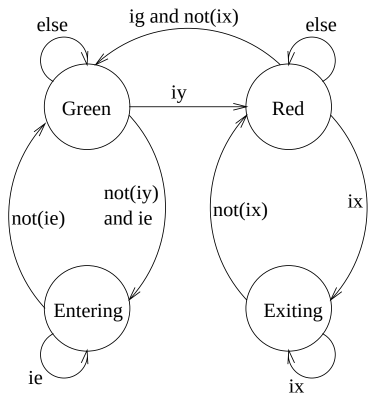

Fig. 11. State Transition for the ILC

The implementation of the ILC is shown in Figure 12. We use *static functions* to define primitive gates (AND, OR, NAND, etc.). An example is shown below for an OR gate with two inputs.

```
static function or2 (in0,in1) ==
if in0 = true  or  in1 = true
then true
else false
endif
```

We describe the behavior of output signals in addition to internal signals in one transition ITC_IMPL. The time delay between signals is modeled using a transition DFF that represents a flip flop component.

```
transition DFF ==
block
x := Dx
y := Dy
z := Dz
endblock
```

We applied our interface tool on both of these models, generating the ASM-IL models for each one, then we built our target syntax MDG-HDL structural and behavioral models in addition to the algebraic specification and the variable order. The algebraic specification contain the declaration of all abstract and concrete sorts used in the modeling, e.g., the declaration of the concrete sort is generated as:

```
conc_sort(islandState_type,[entering_,exiting_,green_,red_]).
```

Once the required files generated, the application of the formal verification procedures of the MDG tool (e.g., equivalence checking or model checking) on the design models is straightforward. This is done like any MDG tool verification session, which compiles the MDG-HDL model files, runs the corresponding verification procedure, and either reports success or failure. In the latter case a counter-example is generated which contains a trace leading from the initial state to the state where the checking failed.

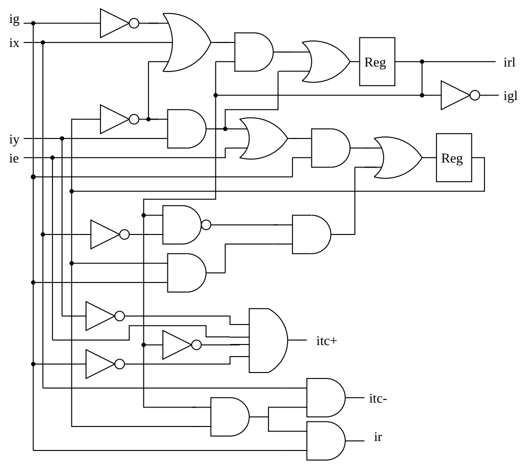

Fig. 12. ILC Implementation

6 Conclusions

In this paper, we introduced a new interface from the ASM Workbench to the MDG tool. This enables the user of ASM to exploit the fully automated verification techniques that are provided by the MDG tool, namely equivalence checking and model checking. In contrast to other model checkers, the MDG tool provides a very simple means for abstracting an ASM model from infinite domains in order to render it feasible for the automated checking approaches. The interface has been implemented and tested on a case study. It automatically transforms ASM-SL models into MDG-HDL code, which is the input language of the MDG tool. The first step ASM to ASM-IL is tailored to the ASM Workbench, however, if any other ASM tool would provide the same step into some kind of ASM-IL, we certainly could reuse our mapping into MDG.

The approach is built upon existing work that transforms ASM to an intermediate language. Since no structure from the original ASM model is preserved in this step, structural information needed to be regained in the second step that transforms the intermediate language to the different parts of the MDG-HDL code. Apart from this code, the interface produces a static variable ordering, necessary for the checking procedures, that satisfies the restrictions given by the MDG approach.

Although the case study, the Island Tunnel Controller, is a hardware example and could have also been modeled in MDG-HDL, the benefits of extending the MDG tool with a general high-level modeling language like ASM are easy to realize once the user focuses on non-hardware problems. Furthermore, the case study nicely demonstrates the benefits of the MDG tool over ordinary ROBDD-based tools, like SMV: Parameterized models can be checked without instantiating the parameters. In the case of the Island Tunnel Controller, the model could be checked for an arbitrary number of allowed cars in the tunnel.

References

1. S. Balakrishnan A Hierarchical Approach to the Formal Verification of Embedded Systems Using MDGs. Master's Thesis, Concordia University, Department of Electrical and Computer Engineering, November 1999.
2. D. Beauquier and A. Slissenko. The Railroad Crossing Problem: Towards Semantics of Timed Algorithms and their Model-Checking in High-Level Languages. In M. Bidoit and M. Dauchet (eds.), Proceedings of TAPSOFT'97, LNCS 1214, Springer-Verlag, 1997.
3. E. Cerny, F. Corella, M. Langevin, X. Song, S. Tahar, Z. Zhou. Automated Verification with Abstract State Machines Using Multiway Decision Graphs. In T. Kropf (ed.), Formal Hardware Verification: Methods and Systems in Comparison, LNCS 1287, State-of-the-Art Survey, Springer-Verlag, 1997, pp. 79-113.
4. F. Corella, Z. Zhou, X. Song, M. Langevin and E. Cerny. Multiway Decision Graphs for Automated Hardware Verification. Formal Methods in System Design, Vol. 10, February 1997, pp. 7-46.

5. G. Del Castillo. The ASM Workbench: A Tool Environment for Computer-Aided Analysis and Validation of Abstract State Machine Models. Ph.D. Thesis, Heinz Nixdorf Institute, Paderborn, Germany, 2000.
6. G. Del Castillo and K. Winter. Model Checking Support for the ASM High-level Specification Language. In S. Graf and M. Schwartzbach (eds.), Tools and Algorithms for the Construction and Analysis of Systems. LNCS 1785, Springer-Verlag, pp. 331- 346, 2000.
7. Y. Feng, E. Cerny. Term Ordering Problem on MDG. In Proc. ACM 12th Great Lakes Symposium on VLSI, New York City, New York, USA, April 2002, pp. 160-165.
8. J.K. Huggins. Abstract State Machines home page. EECS Department, University of Michigan. http://www.eecs.umich.edu/gasm/.
9. S. Katz and O. Grumberg. A Framework for Translating Models and Specification. In K. Sere and M. Butler (eds.), Integrated Formal Methods. LNCS 2335, Springer-Verlag, 2002.
10. S. Kort, S. Tahar, and P. Curzon. Hierarchical Verification Using an MDG-HOL Hybrid Tool. In: T. Margaria and T. Melham (eds.), Correct Hardware Design and Verification Methods, LNCS 2144, Springer Verlag, 2001, pp. 244-258.
11. T. Kropf. Introduction to Formal Hardware Verification, Springer-Verlag, 1999.
12. Y. Mokhtari, M. Shirazipour, and S. Tahar. A Case Study on Model Checking and Refinement of Abstract State Machines. Proc. Eight International Conference on Computer Aided Systems Theory, Las Palmas de Gran Canaria, Canary Islands, Spain, February 2001, pp. 239 242.
13. C. Plonka. Model checking for the design with abstract state machines. Master's thesis, University of Ulm, Department of Applied Information Processing, 2000.
14. M. Spielmann. Automatic verification of abstract state machines. In N. Halbwachs and D. Peled (eds.), Computer Aided Verification, LNCS 1633, pages 431-442. Springer- Verlag, 1999.
15. N. Shankar. Symbolic Analysis of Transition Systems. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele (eds.), Abstract State Machines, Theory and Applications, ASM 2000, LNCS 1912, pp. 287-302, Springer-Verlag, 2000.
16. S. Tahar, X. Song, E. Cerny, Z. Zhou, M. Langevin and O. Ait-Mohamed. Modeling and Verification of the Fairisle ATM Switch Fabric using MDGs. IEEE Transactions on CAD of Integrated Circuits and Systems, Vol. 18, No. 7, July 1999, pp. 956-972.
17. K. Winter. Model Checking Abstract State Machines, Ph.D. thesis, Technical University of Berlin 2001.
18. Y. Xu, E. Cerny, X. Song, F. Corella, O. Mohamed. Model Checking for First-Order Temporal Logic using Multiway Decision Graphs. In Computer Aided Verification, LNCS 1427, Springer Verlag, 1998, pp. 219-231.
19. Z. Zhou, X. Song, S. Tahar, E. Cerny, F. Corella, M. Langevin. Formal Verification of the Island Tunnel Controller using Multiway Decision Graphs. In Formal Methods in Computer-Aided Design, LNCS 1166, Springer Verlag, 1996, pp. 233-246.
20. Z. Zhou and N. Boulerice. MDG Tools (v1.0) User's Manual. Dept. of Information and Operation Research, University of Montreal, Canada, 1996.
21. M.H. Zobair. Modeling and Formal Verification of a Telecom System Block using MDGs. M.A.Sc. Thesis, Concordia University, Department of Electrical and Computer Engineering, Montreal, Canada, April 2001.