Formal Verification of ASM Designs using the MDG Tool

Amjad Gawanmeh, Sofiène Tahar
Concordia University
Montreal, Quebec
H3G 1M8 Canada
{amjad,tahar}@ece.concordia.ca

Kirsten Winter
SVRC
University of Queensland
Australia
kirsten@svrc.uq.edu.au

Abstract

In this paper, we present a formal hardware verification framework linking ASM with MDG. ASM (Abstract State Machine) is a state based language for describing transition systems. MDG (Multiway Decision Graphs) provides symbolic representation of transition systems with support of abstract sorts and functions. We implemented a transformation tool that automatically generates MDG models from ASM specifications, then formal verification techniques provided by the MDG tool, such as model checking or equivalence checking, can be applied on the generated models. We support this work with a case study of an Island Tunnel Controller, which behavior and structure were specified in ASM then using our ASM-MDG tool successfully verified within the MDG tool.

1 Introduction

There has been a recent surge of interest in formal verification and tool support recently, this is because of the increasing complexity of digital hardware systems, and as a result, it is becoming impossible to simulate large designs adequately.

ASM (Abstract State Machines) [10] is a formal specification method for software and hardware systems that has become successful for specifying and verifying complex systems [4]. It provides powerful means for *abstraction* and *uninterpreted function symbols* in order to fit larger models into the validation and verification process, which are not available in other hardware modeling languages like VHDL and Verilog, in addition, ASM is not limited to the hardware domain, but also includes software modeling, or mixture of both. An ASM model describes the state space of the system by means of universes or functions, and the state transitions by means of transition rules. ASM is used as a modeling language in a variety of domains as it has been used both in academic and industry contexts [4, 11]. The wide group of ASM users shows that there is interest in the language and, consequently, there is an interest in tool support. ASM model transition systems in a simple and uniform fashion and give these transition systems an operational semantics. Many verification tools that are available are based on transition systems. A transformation from ASM into these tools' languages can be done without losing properties of the original model [18].

MDGs (Multiway Decision Graphs) [6] 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 [22]. The MDG tool can support verification of larger systems as shown by different case studies [2, 5, 17, 21, 23]. However, the main problem of verification with the MDG tool, is the lack of support by any high level language for specification or a standard hardware description languages such as VHDL or Verilog, instead MDG-HDL is used, which contains no support for advanced modeling features like modularity and hierarchy.

In this paper, we introduce a formal hardware verification framework linking ASM with MDG as shown in Figure 1. We chose to interface ASM with the MDG tool for three reasons: first, both notions, ASM and MDGs, are closely related to each other since they are both based on a subset of many-sorted first order logic and the abstract representation of data. They also both support uninterpreted functions which is not available in many hardware modeling languages. In fact the transformation is easier and more concise than the treatment of the syntax of another input language would be. Second, MDGs as data structure for representing transition systems provide a powerful means for abstraction in order to fit large models into the model checking process. Finally, the need to provide the MDG tool with a high-level modeling language, namely ASM, would allow MDG users to model a wide range of applications in a more elegant and succinct manner.

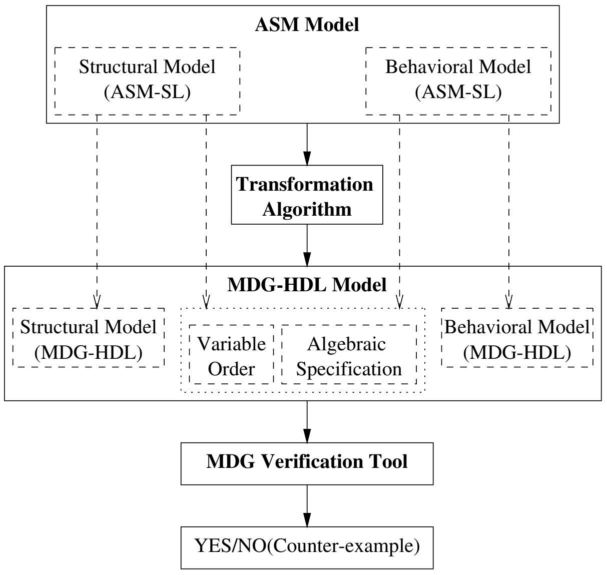


Figure 1. ASM-MDG verification procedure

This work is a major extension of initial ideas presented in [8], where a tool interface between ASM and MDG was proposed through the usage of an intermediate specification language, ASM-IL [18], which is a flat architectural model. In fact, experiments have shown that the generated MDG-HDL code for structural models (design implementations) cannot even be compiled by the MDG tool for a medium range circuit due to its large size. To overcome this problem, we propose in this paper a totally different transformation approach based on syntactic analysis and direct mapping of design structures. Moreover, we propose a complete framework, which enables the verification of translated ASM models in the MDG tool by extracting and generating required data structures and files for the verification procedure. We illustrate the proposed approach and tool interface through a comprehensive case study of an Island Tunnel Controller.

The work introduced in [18] about interfacing ASM and MDG is closely related to our work, however the purpose of that work was to represent ASM models using the MDG data structures. Doing so, will not enable us to use the MDG tool as a black–box tool. In other words, the work there provided an interface to the MDG internal data structures, rather than to the MDG tool [18]. It also provided an interface for ASM models with the SMV model checker [14], however, 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.

Other related work in the open literature about verification of ASM models include the work of Spielmann [15], who investigated the problem of verifying a class of restricted abstract state machine programs (called nullary programs) automatically. In the work on real-time systems by Beauquier and Slissenko [3], ASMs are represented by an extension of the theory of real addition and then the verification problem is discussed. 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 [16] and Katz and Grumberg [12] are 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 [13], Kort *et al.* describe a hybrid formal hardware verification tool linking MDG and the HOL theorem prover obtaining the advantages of both verification paradigms.

2 Abstract State Machines

Abstract State Machines (ASM) [10, 11] is a specification method for software and hardware modeling. It is efficient for modeling a wide range of systems and algorithms as the number of case studies demonstrates [11]. The system is modeled by a set of states and transition rules.

States are given as many sorted first-order structures, and are usually described in terms of functions. A structure is given with respect to a signature. A signature is a finite collection of function names, each of a fixed arity. The given structure fixes the syntax by naming sorts and functions, and provides carrier sets and a suitable symbol interpretation on the carrier sets, which assigns a meaning to the signature. So a state can be defined as an algebra for a given signature with *universes* (*domains or carrier sets*) and an interpretation for each function symbol. States are usually described in terms of functions. A *location* of a state is a pair of a dynamic function symbol and a tuple of elements in the domain of the function. For changing values of locations the notion of an *update* is used. An *update* of state is a pair of location and value. To fire an update at the state, the update value is set to the new value of the location and the dynamic function is redefined to map the location into the value. This redefinition causes the state transition. The resulting state is a successor state of the current state with respect to the *update*. All other locations in the next state are unaffected and keep their value as in the current state.

Transition rules define the changes over time of the states of ASMs. While terms denote values, transition rules denote *update sets*, and are used to define the dynamic behavior of an ASM. ASM runs starting in a given initial state are determined by a closed transition rule declared to be the *program*. Each next state is obtained by firing the update sets at the current state. Basic transition rules are *skip*, *up-*

date, *block*, and *conditional rules*.

The notion of ASM includes *static functions*, *dynamic functions* and *external functions*. ***Static functions*** have a fixed interpretation in each computation state: that is, static functions never change during a run. They represent primitive operations of the system, such as operations of abstract data types (in software specifications) or combinational logic blocks (in hardware specifications). ***Dynamic functions*** which interpretation can be changed by the transition occurring in a given computation step, that is, dynamic functions change during a run as a result of the specified system's behavior. They represent the internal state of the system. ***External functions*** which interpretation is determined in each state by the environment. Changes in external functions which take place during a run are not controlled by the system, rather they reflect environmental changes which are considered uncontrollable for the system [18].

The ASM Specification Language (ASM-SL) [7] is the language used for specifying ASM models. In case of a hardware model we can have behavioral (specification) as well as a structural description (implementation) for the same system. A behavioral description is a higher-level model of the system, we can 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 our primitives, such as operations on abstract data types or combinatorial logic blocks. From these primitives we build a hierarchical or modular structure of the system.

In order to exploit the support of abstract data types provided by MDGs, a syntactic feature to label any sort as being an *abstract sort* was introduced [18]. Functions over abstract sorts do not have a fixed interpretation. They generally substitute infinite sorts, and functions over them, since these cannot be exhaustively explored. They allow for any interpretation that matches their signature. Abstracting from sorts is a means of lifting a "concrete" ASM model into an "abstract" ASM model whose instances comprise concrete models for all possible interpretations of the abstract sorts and functions.

3 Multiway Decision Graphs

MDG [6] is a relatively new class of decision diagrams which subsumes the traditional ROBDDs 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 α is a set of distinct constants of sort α. The constants occurring in the enumeration are referred to as *individual constants*, and other constants as *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 labeled 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. In MDG, a data value can be represented by a single variable of abstract type rather than by concrete (e.g., 32 bits) boolean variables. Variables of concrete sorts are used for representing control signals. Using MDGs, a data operation is represented by an uninterpreted function symbol. As a special case of uninterpreted functions, cross-operators are useful for modeling feedback from the datapath to the control circuitry.

Using abstract sorts and uninterpreted functions reduces the size of the model represented by the MDG, and thus makes reachability analysis and equivalence checking feasible for lager systems. It allows the user to model on a higher level of abstraction and to hide design details of the lower level. In terms of hardware systems, for instance, the user can model at the register transfer level (RTL) rather than the logic gate level. MDGs hence allow a direct representation of the high level descriptions without additional encoding into Booleans (which is necessary when using ROBDDs).

The MDG tool accepts hardware description in a Prolog-style called MDG-HDL, which allows the use of abstract variables for representing data signals. MDG-HDL supports structural descriptions, behavioral descriptions, or a mixture of both. Often models on both levels of abstraction are given and shown to have equivalent behavior (e.g., by means of sequential equivalence checking).

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, black box components, etc. Moreover, a special structure is defined called *tables*, which can be used to describe functional blocks. A table is similar to truth table, but allows first-order terms in the rows.

MDG tool supports model checking, equivalence checking and invariant checking. Model checking is an algorithm that can be used to determine the validity of formulas (properties) written in some temporal logic with respect to a behavioral model of a system. Equivalence checking is used to prove functional equivalence of two design representations modeled at different levels of abstraction. Equivalence checking can be divided into two categories: combinational equivalence checking and sequential equivalence checking. In combinational equivalence checking, the functions of the two circuits to be compared are converted into canonical representations of Boolean functions, typically Binary Decision Diagrams (BDDs) or their derivatives, which are then structurally compared. Sequential equivalence checking only considers the behavior of the two designs while ignoring their implementation details. Invariant checking is used to show that a class of CTL formulas is *invariant*, i.e., formulas of the form $\mathbf{AG}\alpha$, where α is propositional formula. The semantics of this kind of formulas is that α is true in all reachable states. The properties in MDG are represented in a universally quantified first-order branching time temporal logic, called $\mathcal{L}_{\mathcal{MDG}}$ [20], which is a CTL* like language where all formulas in $\mathcal{L}_{\mathcal{MDG}}$ are path formulas. When any of the verification procedures fails, a counter-example is generated. Figure 2 summarizes the MDG tool applications.

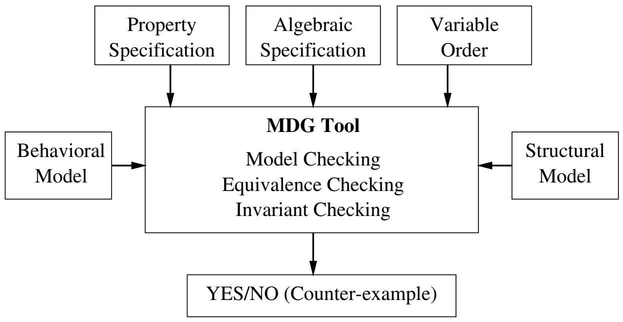


Figure 2. MDG verification tool

The MDG tool has some significant practical limitations: For instance, due to the non-interpretation of data operators, the reachability analysis of abstract states may not terminate [1]. Another practical drawback of the MDG tool with respect to an industrial setting is that they do not accept VHDL or Verilog HDL as input language.

4 ASM-MDG Tool

The interface was motivated by the fact that ASM as a modeling language is very close to MDG, specially when it is used to model hardware systems, since they both provide powerful means for *abstraction* and *uninterpreted function symbols* in order to fit larger models into the validation and verification process. ASM uses the notion of many-sorted first-order structures to describe states of a system and adds transition rules for modeling the system behavior during a run. The MDG approach uses so called “Abstract State Machines” too in order to identify the system that is to be analyzed. In ASM, we treat specific sorts as *abstract sorts* and thus every function that is applied to parameters of these sorts is either a cross–operator or an abstract function and has to be left uninterpreted. MDG is able to handle these abstract sorts, cross–operators, and uninterpreted functions since they can be part of the MDG graph structure as well as the MDG-HDL syntax [5]. In a previous work in [8], we introduced an implementation for the ASM-MDG interface to generate MDG-HDL components from ASM models.

The interface works in two directions: generates behavioral and structural MDG-HDL models. We failed to input generated MDG-HDL structural models into our MDG tool for large designs since they contained a huge number of components beyond the capacity of the tool. This is because we built our interface to the MDG tool based on ASM-IL [18], which is a flattened representation of ASM models. The disadvantage of ASM-IL is the fact that it does not preserve the structure of the original ASM model because it provides no means for modular or hierarchical descriptions. When an ASM model is translated into the ASM-IL rules, all structured functions are flattened into the primitive ones. These rules are used to build the MDG-HDL structural model, which is a set of components interconnected by internal signals. Since MDG-HDL supports neither modularity nor hierarchy, the resulting MDG-HDL structural model will be very large as only the predefined MDG-HDL components are used. A large number of components results also in a large number of variables which makes it very hard to generate a good variable order. The work we present here solves the above problems by introducing a direct interface from ASM to MDG (i.e, without going through the ASM-IL).

The proposed ASM-MDG tool consists of two complementary parts: the first part generates MDG-HDL structural models from ASM specifications, while the second part generates MDG-HDL behavioral models. So we develop two ASM models which are separately transformed into the corresponding MDG-HDL models: ASM behavioral model and an ASM structural model. One models the behavior in terms of transition rules, the other models the structure of the design in terms of static functions.

4.1 Transformation of Structural Models

Figure 3 shows the proposed ASM-MDG direct interface for structural designs. In the first part, ASM universes including all type declarations, ASM functions including static, dynamic and external functions, and transition rules that describe the structure of the model are collected and then used to construct design components, variables, functions and sorts that represent the design. Finally, MDG-HDL models are produced based on the information collected in the previous step. Algebraic specifications are produced based on the generic constants, concrete sorts, abstract sorts, and uninterpreted functions. Variable ordering in turn is generated according to the relationship between variables and functions in the design such that the order obeys the restrictions imposed by the MDG tool. It includes all variables and internal signals used in the model.

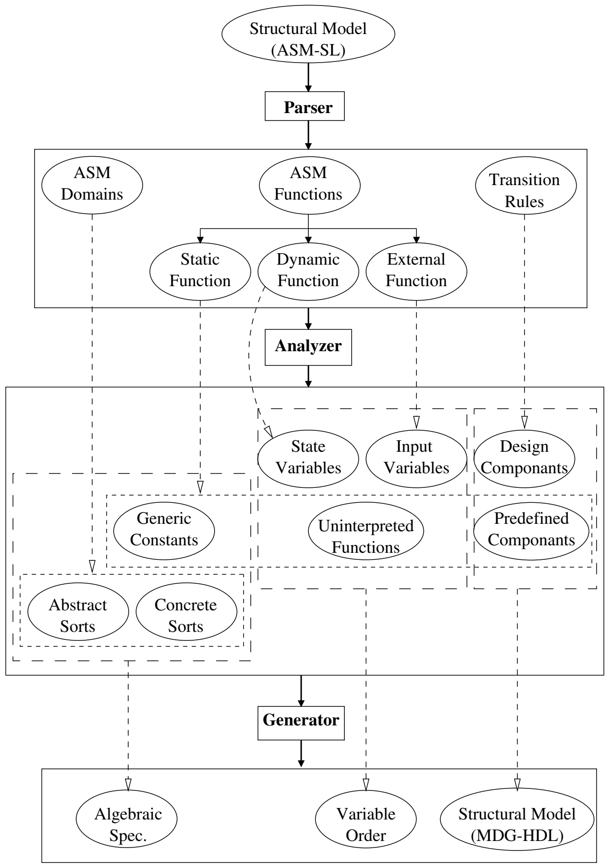
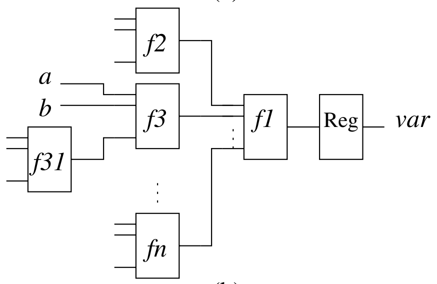


Figure 3. ASM-MDG interface for structural models

The MDG-HDL structural model is a circuit description given as a netlist of components interconnected with signals. This is generated by a one-to-one mapping from an ASM model of static functions to MDG-HDL library of

$\boxed{var := f1(\ f2(...)\ ,\ f3(\ f31(...),\ a,b),\ \ ...,\ \ fn(...)\ )}$

(a)



(b)

component(reg(S1, *var*))
component(*f1*((S11, S12, ..., S1n), S1))
component(*f2*((S21, S22, ..., S2k),S11))
component(*f3*((S31, a, b), S2k))
component(*f31*((S311, S312, ..., S31m),S31))
component(*fn*((Sn1, Sn2, ..., Snj), S1n))

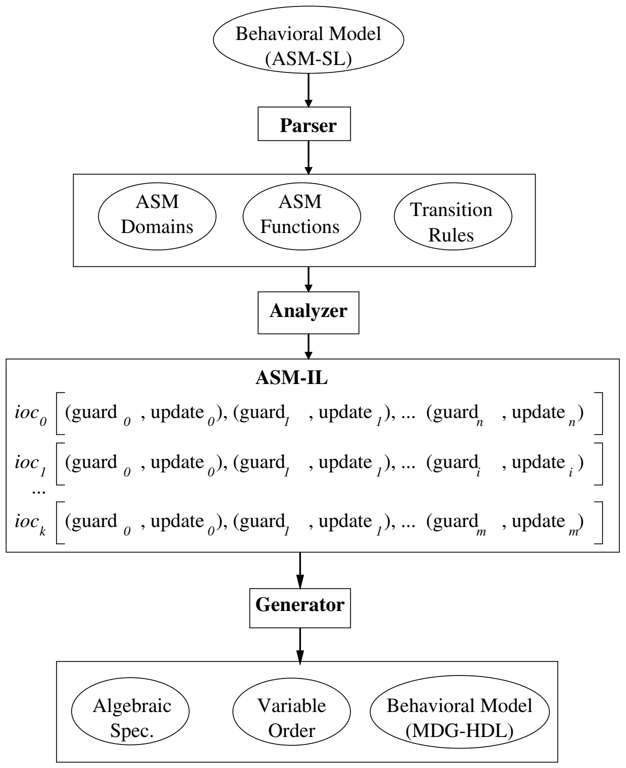
(c)

Figure 4. Mapping structural ASM-SL into MDG-HDL components

components. The current implementation of the tool supports only a set of ASM functions that can be mapped directly to MDG-HDL, in addition to uninterpreted functions and cross–operators. These functions include: AND, OR, NOR, NAND, with up to *n* inputs, inverters and multiplexers. Figure 4 shows a structural modeling of an ASM dynamic function (a), its mapping into MDG-HDL components (b), and the generated MDG-HDL components (c), where *f1*, through *fn* can be any of the library functions above, an uninterpreted function or a cross operator, *var* is the state variable, *Sjk*s are internal signals, and finally *a* and *b* are ASM functions (i.e, variables). All functions are declared as $function((inputs), output)$. This structure is recursively treated until a predefined function is found, which is mapped into the corresponding MDH-HDL library components without exploring its semantical meaning.

4.2 Transformation of Behavioral Models

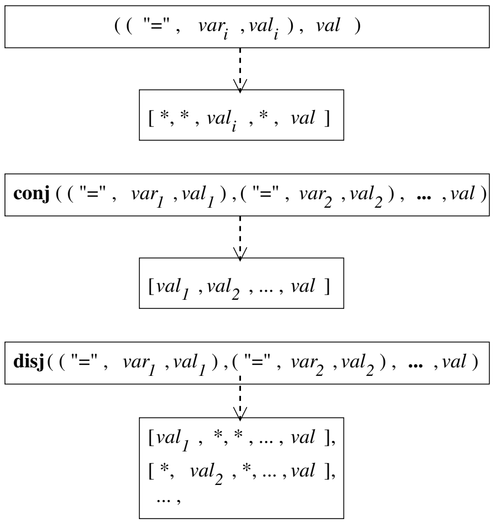
To treat behavioral ASM-SL specifications, the transformation is done via the general intermediate representation in ASM-IL. The transformation is shown in Figure 5. The model is first parsed for syntax check and collection of ASM universes, functions, and transition rules. Then an analyzer generates the ASM-IL representation in which all nested transition rules are flattened and all complex data structures are unfolded [18]. The behavior of the model is described as a set of guards and updates for each state variable. The state variable evaluates in the next state to the given value if the corresponding guard is satisfied, otherwise it keeps the same value as in the previous state. Based on this ASM-IL model, MDG-HDL behavioral description is generated in terms of tabular representations similar to truth tables. In addition, a variable order and algebraic specifications are produced [9].



Behavioral Model (ASM-SL)
Parser
ASM Domains
ASM Functions
Transition Rules
Analyzer
ASM-IL
ioc_0 [(guard$_0$, update$_0$), (guard$_1$, update$_1$), ... (guard$_n$, update$_n$)]
ioc_1 [(guard$_0$, update$_0$), (guard$_1$, update$_1$), ... (guard$_i$, update$_i$)]
...
ioc_k [(guard$_0$, update$_0$), (guard$_1$, update$_1$), ... (guard$_m$, update$_m$)]
Generator
Algebraic Spec.
Variable Order
Behavioral Model (MDG-HDL)

Figure 5. ASM-MDG interface for behavioral models

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 occurs in the ASM-IL guarded update expression of that location. The last element is the location itself, it represents the variable in the next state. Then we treat the list of (*guard, value*) pairs one by one (see Figure 6). 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 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. The last element of each of these rows contains the value of the location as shown in Figure 6.



(("=", 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],
...,

Figure 6. Creating MDG tables from ASM-IL guarded updates

5 Case Study: Island Tunnel Controller

In this section, we provide a case study application of our ASM-MDG interface based on the example of an Island Tunnel Controller [21] in order to illustrate the proposed ASM-MDG interface. The Island Tunnel Controller (ITC) is used to control two traffic lights for a tunnel that connects an island to the mainland. The island allows cars to travel in one direction only. There can be a maximum number of cars in the tunnel at one time, also the number of cars on the island cannot exceed a specific maximum. There are four tunnel sensors to detect vehicles at both sides of the tunnel and four output signals to control the traffic lights at both sides. The ITC is specified using three communicating controllers: Island Light Controller (ILC), Tunnel Controller (TC) and Main Land Controller (MLC), and two counters: Tunnel Counter (TCR) and Island Counter (ICR), as shown in Figure 7. The Tunnel Counter counts the number of cars inside the tunnel, and the Island Counter counts the number of cars on the island. Initially, both lights are assumed to be red and both counters are set to zero and no vehicles are in the tunnel or on the island.

5.1 ASM Modeling

The maximum number of cars to be on the island at one time can be taken in ASM as a parameter of an abstract type that represents any natural number. We can then define a cross-operator for the operation "$tc < maxcar$". This allows modeling the controller for any number of cars, nicely illustrating the advantage of using abstract types that are

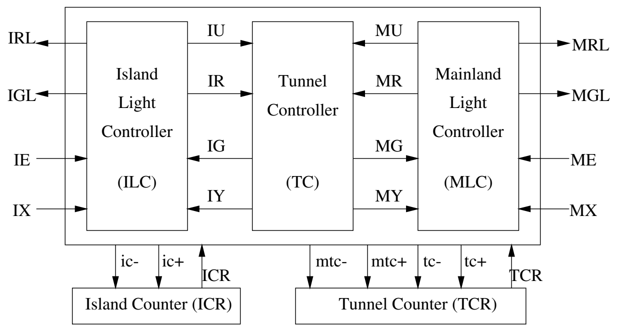


Figure 7. Three-Controllers design of the ITC

supported by our framework as we are able to verify this system for any arbitrary counter size. With this abstraction, some properties that are related to the counters would not hold for this model, however, we are interested in the cases in which the number of cars on the island (or in the tunnel) is equal to either *zero*, or *maxcar*. All other values can be abstracted into one value since they would represent the same state for the controller. We choose two blocks; the MLC and ILC as parts of the design to be modeled in ASM and verified using the MDG tool. For each, we developed a behavioral model (specification) and a structural model (implementation). MLC is described in details below. Since the ILC is similar, we just show the models for the MLC.

5.1.1 Behavioral Modeling in ASM

Figure 8 shows the state transition diagram for the MLC, where "&" means logical AND, "|" means logical OR, and the bar above the variable means complement. It is assumed to be initially in the *RED* state, where *IRL* is set to 1 while in this state.

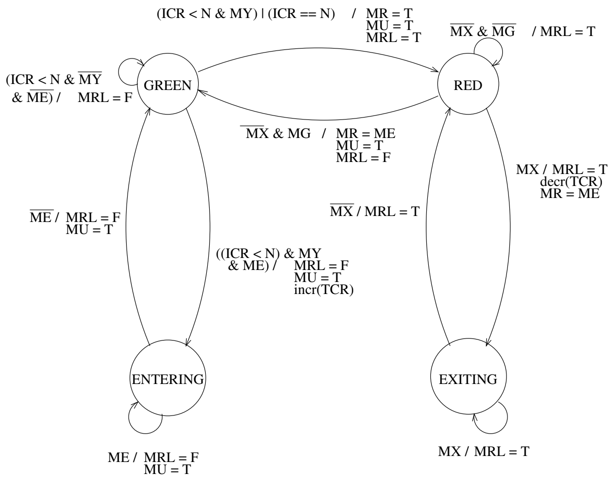

(ICR < N & MY) | (ICR == N) / MR = T
MU = T
MRL = T
$\overline{\text{MX}}$ & $\overline{\text{MG}}$ / MRL = T
(ICR < N & $\overline{\text{MY}}$
& $\overline{\text{ME}}$) / MRL = F
GREEN
RED
$\overline{\text{MX}}$ & MG / MR = ME
MU = T
MRL = F
MX / MRL = T
decr(TCR)
MR = ME
$\overline{\text{ME}}$ / MRL = F
MU = T
$\overline{\text{MX}}$ / MRL = T
((ICR < N) & MY
& ME) / MRL = F
MU = T
incr(TCR)
ENTERING
EXITING
ME / MRL = F
MU = T
MX / MRL = T

Figure 8. ASM transition system for the MLC

The behavior of the MLC is modeled in ASM by defining a free type that represents the states of the controller. Increment and decrement operations on the counters are generally infinite mappings over integers. In our model, we specify those as abstract *static functions*, which map abstract values. These functions are left uninterpreted in our transformation. Also comparison operation between tunnel counter and maximum number of cars allowed to be in the tunnel is modeled with a cross–operator *lt*. *External functions* are used to represent environment operations for detecting vehicles at entrance or exit in addition to signals from the TC, e.g., *carentering* (*ME*), *carexiting* (*MX*), *mainlandgranted* (*MG*), and *mainlandrelease* (*MY*) (see Figure 7).

We describe the controller states using the *dynamic function*: *mainlandstate* which is of the type `IS_SORT` that has the enumeration {`green, red, exiting, entering`}

All Boolean outputs of this controller are also described by *dynamic functions*, e.g., *mainlandgreen* (*MGL*), *mainlandred* (*MRL*), *mainlanduse* (*MU*), and *mainlandrequest* (*MR*). We then describe the behavior of the system using the *if-then-else* rules. One example is shown below for the *GREEN* state. The behavior of the ILC is modeled by the same way.

```
if (mainlandstate = green) then
     mainlandred := false
     mainlandgreen := true
     mainlandrequest := false
```

5.1.2 Structural Modeling in ASM

We developed a structural model (implementation) for the MLC model as shown in Figure 9, which was derived from the specification given above in Figure 8. For the ASM modeling, we used *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
```

An abstract state variable is used to model the abstract counter. Black-box representation is used to model the increment (*incr*) and decrement (*decr*) functions, while the comparator "*tc* < *maxcar*" is modeled with a black-box representing a cross–operator.

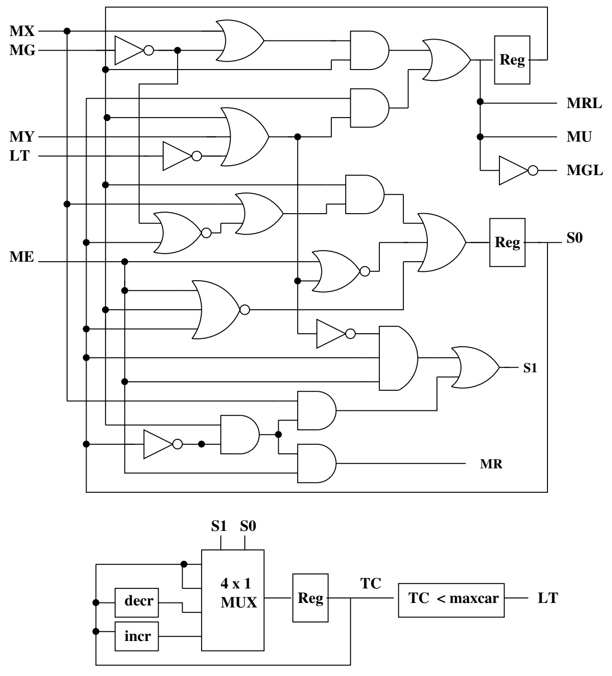


Figure 9. MLC Implementation

5.2 MDG Verification

Using our ASM-MDG tool, we generated the corresponding MDG-HDL models for both behavioral and structural models for each block, including: circuit description, algebraic specifications, and variable order [1].

Once the generated MDG-HDL structural and behavioral models were compiled successfully with the MDG tool, we applied both sequential equivalence checking and model checking on the generated models.

To verify that the MLC structural model is equivalent to its behavioral model, we applied MDG sequential equivalence checking on the generated MDG-HDL models. In the following, however, we verify, for illustration purposes, the MLC implementation including one error, which we injected into the model. The assertion of the equivalence of two models is done by the assertion that the corresponding observable outputs of the two designs are equivalent. While verifying the faulty design, the equivalence was violated and the tool generated a counter–example.

The CPU execution time and resource requirements, including memory usage and MDG nodes generated, are given in Table 1 shown below. The experimental results were conducted on a Sun enterprise server with Solaris 5.7 OS and 6.0 GB memory.

Next, a set of properties were specified in $\mathcal{L}_{\mathcal{MDG}}$ format,

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

Table 1. MDG sequential equivalence checking results

	CPU Time (Sec)	Memory (MB)	# of MDG Nodes
MLC (Original)	0.730	155	955
MLC (Faulty)	1.040	1.41	1180
ILC (Original)	0.580	0.92	668
ILC (Faulty)	0.580	1.00	763

and then MDG model checking was applied to verify that the generated MDG-HDL models satisfy them all. For example, a (liveness) property on the MLC states that:

```
if the mainland is requesting the
controller, then it will be using it
at future time.
```

This property is formally specified as following, where the symbols AG and F mean "for all paths, for all states" and "there exists a state in the future", respectively:

Property 1:
$AG((mainlandrequest = 1) => (F(mainlanduse = 1)));$

Another (safety) property on the MLC states that:

```
mainland green light and mainland red
light should never be active simultane-
ously
```

and it is formally specified as following, where & is the logical AND and ! is the logical NOT:

Property 2:
$AG(!((mainlandgreen = 1)\ \&\ (mainlandred = 1)));$

All properties were verified successfully. Verification results for a set of properties on the MLC (including the above two) and on the ILC are given in Table 2.

6 Conclusions

In this paper, we introduced a formal verification framework interfacing ASMs (Abstract State Machines) to the MDG (Multiway Decision Graphs) tool. This new interface, called "ASM-MDG", enables ASM users to exploit the fully automated verification techniques that are provided by the MDG tool, namely equivalence checking and model checking. On the other hand, MDG users will be

Table 2. MDG model checking results

Property	CPU Time (Sec)	Memory (MB)	# of MDG Nodes
Property 1 (MLC)	0.690	1.29	888
Property 2 (MLC)	0.540	1.75	606
Property 3 (MLC)	0.560	0.83	608
Property 4 (ILC)	0.460	0.86	507
Property 5 (ILC)	0.390	1.60	407
Property 6 (ILC)	0.410	0.29	399

provided by a high-level modeling language, namely ASM, which as MDG, supports abstract data sorts and uninterpreted functions. The interface automatically transforms the ASM specification language, ASM-SL, into the MDG hardware description language, MDG-HDL. This transformation is done in two directions. In the first, we translate ASM-SL behavioral models into an intermediate language, ASM-IL, and then transform this intermediate model into the appropriate MDG-HDL behavioral code. In the second, we translate ASM-SL structural models directly into MDG-HDL netlist components using syntactic analysis and transformation. Besides the MDG-HDL code, the interface produces a static variable ordering that satisfies the restrictions given by the MDG approach, as well as algebraic specification necessary for the checking procedures, such as sort and functions definitions, etc.

The first approach was built upon existing work [8] that proposes to transform ASM-SL to ASM-IL. 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 is lost.

We have applied the ASM-MDG interface on the Island Tunnel Controller as a case study. We conducted MDG model checking and equivalence checking on the generated MDG-HDL models. We succeeded in verifying several properties on the Mainland Light Controller (MLC) and Island Light Controller (ILC), and also verified that both MLC and ILC implementations are equivalent to their respective specifications.

Although the case study of the Island Tunnel Controller is a hardware example and could have also been modeled directly 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 behavioral hardware problems, with ASM we can model on different levels of abstraction in the same formalism (namely ASM). 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] O. Ait-Mohamed, X. Song, E. Cerny. On the nontermination of MDG-based abstract state enumeration. In Proc. IFIP Conference on Correct Hardware and Verification Methods, Montreal, Canada, October 1997, pp. 218–235.

[2] 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.

[3] 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.

[4] E. Borger and J. Huggins, "Abstract State Machines 1988-1998: Commented ASM Bibliography. " Formal Specification Column (H. Ehrig, ed.), EATCS Bulletin 64, February 1998, 105-127.

[5] 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 component, LNCS 1287, State-of-the-Art Survey, Springer-Verlag, 1997, pp. 79–113.

[6] 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.

[7] 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.

[8] A. Gawanmeh, S. Tahar and K. Winter. Interfacing ASMs with the MDG Tool, In E. Borger, A. Gargantini and E. Recobene (eds), Abstract State Machines - Advances in Theory and Applications, LNCS 2589, Springer Verlag, 2003, pp 278-292.

[9] A. Gawanmeh, Interfacing Abstract State Machines with Multiway Decision Graphs, MASc. thesis, Concordia University, Montreal, Canada 2003.

[10] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Borger (ed), Specification and Validation Methods. Oxford University Press, 1995.

[11] J.K. Huggins. Abstract State Machines home page. EECS Department, University of Michigan. http://www.eecs.umich.edu/gasm/.

[12] 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.

[13] 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.

[14] M.L. McMillan. Symbolic Model Checking. Norwell, MA: Kluwer, 1993.

[15] M. Spielmann. Automatic verification of abstract state machines. In N. Halbwachs and D. Peled (eds.), Computer Aided Verification, LNCS 1633, Springer Verlag, 1999, pp 431-442.

[16] N. Shankar. Symbolic Analysis of Transition Systems. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele (eds.), Abstract State Machines, Theory and Applications, LNCS 1912, Springer-Verlag, 2000, pp. 287–302.

[17] 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.

[18] K. Winter. Model Checking Abstract State Machines, Ph.D. thesis, Technical University of Berlin 2001.

[19] K. Winter. Model Checking with Abstract Types, Workshop on Software Model Checking SOFTMC01, July, 2001, Paris, France.

[20] Y. Xu, E. Cerny, X. Song, F. Corella, O. Mohamed. Model Checking for First-Order Temporal Logic using Multiway Decision Graphs. In A. Hu and M. Vardi (eds.), Computer Aided Verification, LNCS 1427, Springer Verlag, 1998, pp. 219–231.

[21] 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.

[22] Z. Zhou and N. Boulerice. MDG Tools (v1.0) User's Manual. University of Montreal, Dept. of Information and Operation Research, 1996.

[23] 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, April 2001.