Formal Verification of ASM Designs using the MDG Tool

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

[1]Department of Electrical and Computer Engineering,
Concordia University, Montreal, Canada
Email: {amjad, tahar}@ece.concordia.ca

[2]SVRC,
University of Queensland, Australia
Email: kirsten@svrc.uq.edu.au

Technical Report

June, 2003

Abstract

Digital systems are becoming very large and complex making the process of finding bugs and design validation in early stages of the design cycle a must. As a contribution towards catching this goal, we propose 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 mainly used for modeling hardware systems. Both ASM and MDG are based on a subset of many-sorted first order logic, making it appealing to link these two concepts. 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 consider both structural and behavioral models of hardware in our implementation. We have applied this transformation schema on some examples and case studies where our tool generates the corresponding MDG-HDL models automatically.

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 VLSI 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 in particular model checking [16, 18]. 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, they have, however, the problem of state space explosion [16]. Theorem proving, on the other hand, is not automatic approach, but it can be applied on larger systems.

MDGs (Multiway Decision Graphs) [7] 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 [27]. MDG tool can support verification of larger systems as different case studies show [2, 6, 23, 29, 31]. However, the main problem of verification with MDG tool, is that it does not support hardware description languages like VHDL and Verilog, instead MDG-HDL, which contains no support for advanced modeling features like modularity and hierarchy.

ASM (Abstract State Machines) [13] 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, 14]. 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 [24].

We present a tool to interface the ASM-WB (ASM Workbench) [8, 9] with the MDG applications in order to enable the formal verification of ASM descriptions. 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, enabling the abstract representation of data. They both also 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.

Due to the advanced facilities of MDGs, this interface supports an easy abstraction mechanism for ASM as introduced by the work of Winter [24, 25, 10]. At the same time, in contrast to the work in [25], it enables us to make use of the existing MDG tool that provides equivalence checking and model checking. This work has been motivated by results obtained in [20] on the verification of hardware designs based on ASM models.

For behavioral models, we intend to develop the ASM-MDG interface in two steps: in the first step, the ASM model is transformed into a flat, simple transition system, called the *Intermediate Language* (ASM-IL) [24]. The second step provides a transformation from IL into the syntax of the input language of the MDG tool, MDG-HDL. For structural models we implemented a syntax transformation interface directly from ASM to MDG-HDL where the ASM model is restricted to the MDG-HDL library components.

We have applied the ASM-MDG interface on an Island Tunnel Controller as a case study, where we conducted MDG model checking and equivalence checking on the generated MDG-HDL models. We succeeded in model checking several properties on the Mainland Tunnel Controller and Island Tunnel Controller, and we also verified that the implementation of each block is equivalent to its specification.

The work introduced in [24] 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 [24].

The work in [24] also provided an interface for ASM models with the SMV model checker [19], 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 [22], 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 [21] and Katz and Grumberg [15] 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 [17], 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 Formal Hardware Verification

Validation techniques include simulation, testing, prototyping and formal verification. Traditionally, testing and simulation are used to check the designs correctness, and because they are inadequate to certify that a system behaves correctly, the evolution of alternative verification approaches has emerged, such as formal methods. Formal methods have the potential for significantly reducing

the number of design faults in designs and at the same time reducing the cost of the design [16]. Formal hardware verification uses mathematically-based methods to overcome the weakness of non-exhaustive simulation by proving the correspondence between some abstract specification and the design in hand. There are three different techniques of formal hardware verification, namely:

- Theorem Proving
- Equivalence Checking
- Model Checking

In theorem proving both the implementation as well as the specification are described in a formal logic. The correctness result was then obtained by proving in the logic, that the specification and implementation were suitably related. Theorem proving based verification requires a large amount of effort on the part of the user in developing specifications of each component and in guiding the theorem prover through a large number of lemmas. Therefore, for designs that are not safety critical, theorem proving techniques are too expensive.

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) [5] or their derivatives, which are then structurally compared. The drawback of this type of verification is that it cannot handle the equivalence checking between RTL (Register Transfer Level) and behavioral models. In sequential equivalence checking, given two sequential circuits using the same state encoding, their equivalence can be established by building the product finite state machine and checking whether the values of two corresponding outputs are the same for any initial states of the product machine. Sequential equivalence checking only considers the behavior of the two designs while ignoring their implementation details such as latch mapping. Therefore, sequential equivalence checking is able to verify the equivalence between RTL and behavioral model. The drawback of this technique is that it cannot handle large designs due to state space explosion problem.

Model checking is an algorithm that can be used to determine the validity of formulas written in some temporal logic with respect to a behavioral model of a system. Model checking is based on the state space exploration technique, and uses the reachability state graph as a Kripke structure, which encodes the set of all possible sequences of states for a system over computation trees.

Model checking tools are effective as debugging aids for industrial designs, and since they are fully automated, minimal user effort and knowledge about the underlying technology is required to be able to use them. However, there are two drawbacks with model checking. The first is the state space explosion and how to avoid it, and the second is the difficulty of judging whether the verified properties completely characterize the desired behavior of the system.

3 Abstract State Machines

Abstract State Machines (ASM) [13, 14] 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 [14].

3.1 ASM Language

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.

3.1.1 States

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. 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. 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*** whose 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*** whose 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.

3.1.2 Terms

Variables and *terms* are used over the signature as objects of the structure. The syntax of terms is defined recursively, as in first-order logic:

- A variable is a term. If a variable is Boolean, the term is also Boolean.
- If f is an $r-ary$ function name in a given vocabulary and $t_1, \ldots t_r$ are terms, then $f(t_1, \ldots t_r)$ is a term. The composed term is Boolean if f is relational.

3.1.3 Locations and Updates

States are described using functions and their current interpretations. The state transition into the next state occurs when its function values change. *Locations* and *updates* are used to capture this notion.

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 a location and a 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.

3.1.4 Transition Rules

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*, *update*, *block*, and *conditional rules*.

The *skip* rule is the simplest transition rule. This rule specifies an "empty step". No function value is changed. It is denoted as

$$skip$$

The *update* rule is an atomic rule denoted as

$$f(t_1, t_2, \ldots, t_n) := t$$

It describes the change of interpretation of function f at the place given by $(t_1, t_2, \ldots, t_n)$ to the current state value of *t*.

A *block* rule is a group of sequence of transition rules. The execution of a block rule is the simultaneous execution of the sequence of the transition rules. All transition rules that specify the behavior of the ASM are grouped into a block indicating that all of them are fired simultaneously.

```
block
    R1
    R2
endblock
```

In *conditional rules* a precondition for updating is specified.

```
 if g
then R1 else R2
 endif
```

where g is a first order Boolean term. R_1 and R_2 denote arbitrary transition rules. The condition rule is executed in state S by evaluating the guard g, if *true* R_1 fires, otherwise R_2 fires.

3.2 Modeling with ASM-SL

The ASM Specification Language (ASM-SL) [9] is the language used to describe systems in ASM. 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 our primitives. From these primitives we build a hierarchical or modular structure of the system.

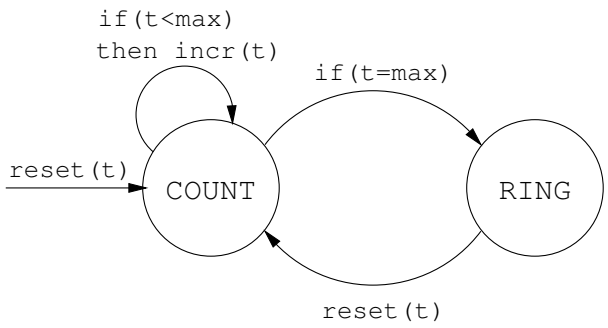

Figure 1: Generic counter state machine

We show the example of a *generic counter* [25] to illustrate the use of ASM in systems modeling. Figure 1 shows the state machine and Figure 2 shows the ASM model of the counter. The example shows clearly the usage of abstract and concrete sorts to model the internal state machine of the system, and it also shows how uninterpreted functions are declared using the "MAP_TO_FUN" operator.

3.3 The ASM Workbench

The ASM Workbench (ASM-WB) [9] provides a number of basic functions including: parsing, type checking, pretty printing and evaluation of terms and rules. It supports computer aided specification, modeling, analysis and validation based on the method of ASM. The ASM-WB supports the ASM specification language (ASM-SL). The main characteristics of ASM-WB is its *kernel* which is a set of program modules implemented in the functional programming language Standard ML [**?**], each module corresponding to a relevant data structure (e.g, abstract syntax trees, signatures) or functionality (e.g, type checker, evaluator). The nature of SML allows exporting an executable image of the ML compiler itself containing the precompiled and preloaded ASM modules called `sml-asm`. This provides a first – not very friendly – user interface for the ASM Workbench, but it also provides immediate access to the data structures and functions of the ASM-WB.

The ASM-WB is designed as an extensible tool, where *transformation algorithms* might be added that serve as interfaces. One interface with the SMV model checker called "ASMSMV Translator" was suggested and implemented in [10]. Since our work is built on preliminary work [24, 25], we can furthermore exploit the notion of *abstract types*. This feature can be essential when applying automated verification techniques like model checking. Figure 3 shows the ASM-WB interface [24].

```
freetype DATA == { abstract }
freetype MODE == { count, ring }
static function Bool  == { true, false }
static function Data == { abstract }
static function Mode == { count, ring }
static function max_time == abstract
static function zero == abstract

dynamic function mode : MODE with mode in Mode initially count
dynamic function t : DATA with t in Data initially max_time
static function incr == MAP_TO_FUN {abstract -> abstract}

transition R1 ==
 if ((mode = count) and (t <= max_time))  then
  t := incr(t)
 endif

transition R2 ==
 if (mode = count) and (t = max_time)   then
     mode := ring
 endif

transition R3 ==
 if (mode = ring)  then
   t := zero
   mode := count
 endif
```

Figure 2: ASM modeling of the generic counter example

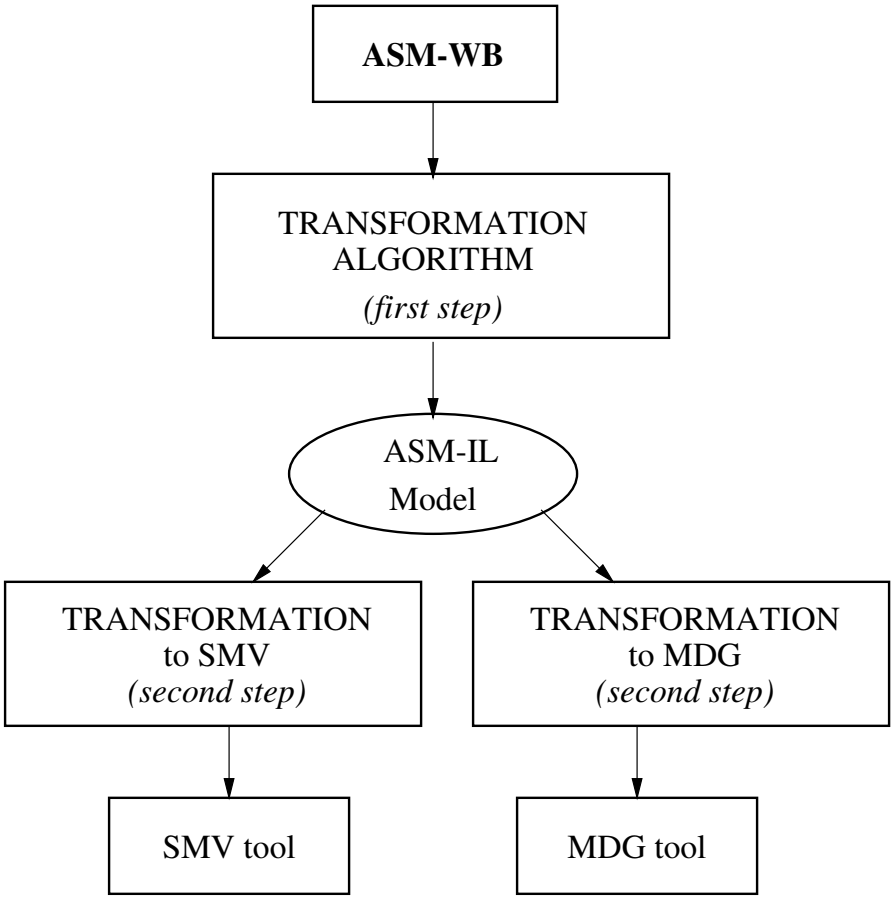

Figure 3: The ASM-WB interface

4 Multiway Decision Graphs

Multiway Decision Graphs (MDGs) [7] have been proposed as a solution to the state space explosion problem of ROBDD (Reduced Order Binary Decision Diagrams) [5] based verification tools. MDGs subsume ROBDDs, while accommodating abstract sorts and uninterpreted function symbols. This significantly enhances the capability to verify a broader range of systems as classical ROBDD based tools.

MDG [7] 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 $\alpha_1 \times \alpha_2 \times \cdots \times \alpha_n \rightarrow \alpha_{n+1}$. 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).

4.1 Modeling with MDG

Logic gates can be represented by MDGs similarly to ROBDDs, because all inputs and outputs are of Boolean type. Figure 4 shows the MDG for an AND gate for a given variable order. A design description on RTL, however, involves the use of more complex functions and data inputs that go beyond the capacity of ROBDDs [28]. For example, Figure 5 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.

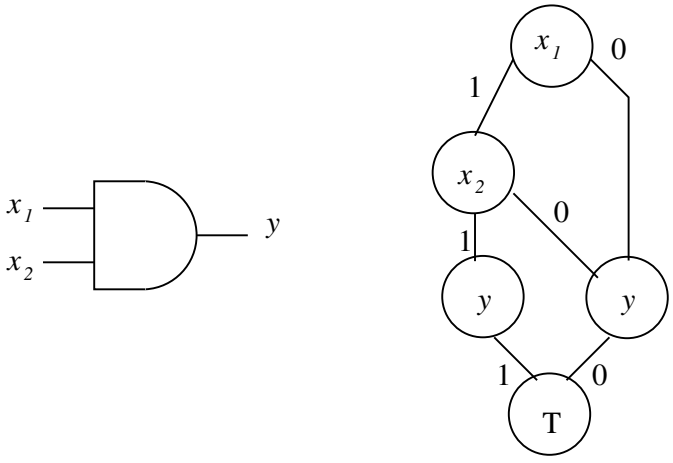

Figure 4: MDG for an AND gate

For system descriptions the MDG tool comes with a Prolog-style hardware description language called MDG-HDL [27]. It allows the use of abstract as well as concrete variables for repre-

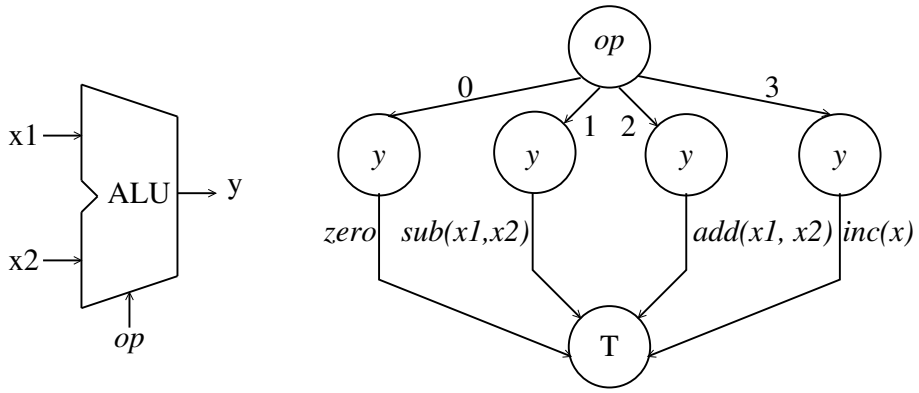

Figure 5: MDG for an ALU

senting data operations. A circuit can be described on the structural level, as an *implementation*, or on the behavioral level, as a *specification.* Often models on both levels of abstraction are given and shown to have equivalent behavior (e.g., by means of sequential 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. The behavioral description is represented by abstract descriptions of state machines [1] 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 6 shows a tabular description of a simple state machine, with its MDG representation for a 4×1 multiplexer, where x and y are Boolean inputs, a is an abstract state variable and a' is its next state variable. It performs *inc* operation when $x = 1$, and *dec* operation when $x = 0$ and $y = 1$, where *inc* and *dec* are uninterpreted function symbols. Figure 7 shows the MDGHDL code for the same example.

4.2 Verification using the MDG Tool

The MDG tool is a tool set for the formal verification of finite state systems (machines) that is based on MDG. It includes application procedures for combinational and sequential equivalence checking [7], invariant checking [7] and model checking [26]. The MDG tool has been used to verify a number of non-trivial systems such as communication switches and protocols [2, 6, 23, 29, 31].

[1] In the MDG literature [7] 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.

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

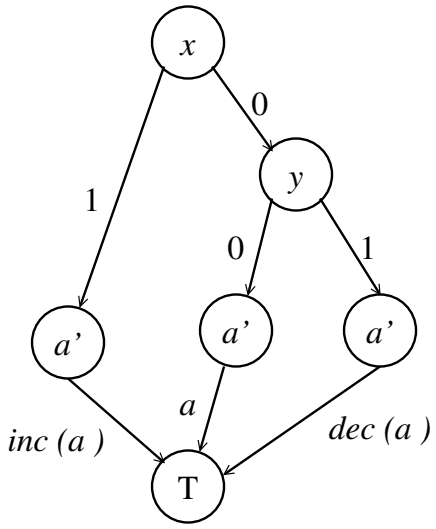

Figure 6: Table and MDG for a simple behavioral state machine

```
component(INC_DEC, table([[x, y, a_next],
                          [0, 0, a],
                          [0, 1, dec(a)],
                          [1, *, inc(a)]])).
```

Figure 7: MDGHDL specification for the example above

For combinational verification, corresponding algorithms based on ROBBDs can be used in the MDG tool because MDGs as well as ROBDDs have canonical forms. Reachability analysis is used in the MDG tool to perform property checking and sequential equivalence checking on designs. Sequential equivalence checking of two state machines (sequential circuits) is performed by checking whether the two machines produce the same sequence of outputs for every sequence of inputs. This is achieved by forming the product machine of both while feeding them with the same inputs and verifying an invariant asserting the equality of the corresponding outputs in all reachable states [7]. A Model checking facility has also been recently developed [26] and incorporated into the existing MDG tool. This provides both safety and liveness property checking using implicit abstract enumeration [7]. The properties are represented in a universally quantified first-order branching time temporal logic, called $\mathcal{L}_{\mathcal{MDG}}$ [7]. When any of the verification procedures fails, a counter-example is generated. This includes assumptions, inputs, and a sequence of states, which provides a trace leading from the initial state to the state where the two designs are not equivalent.

Figure 8 summarizes the MDG tool applications. In order to verify designs with this tool, we first need to specify the design in MDG-HDL in terms of a behavioral and/or structural description (design specification and design implementation in Figure 8). 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. However, there are some requirements on the node ordering of abstract variables and cross-operators (but not for concrete variables). 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 [11].

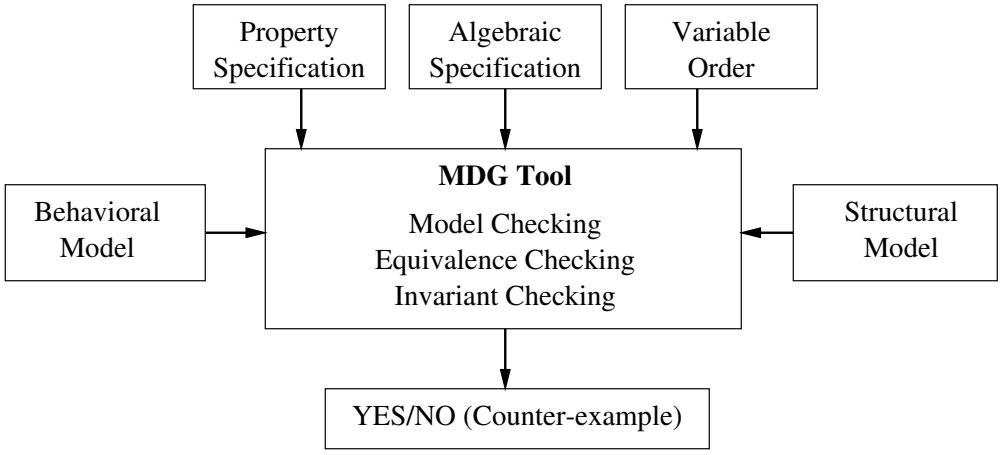

Figure 8: MDG verification tool

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 [30].

5 ASM-MDG Tool

The interface was motivated by the fact that ASM as a modeling language can be 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-term or an abstract function and has to be left uninterpreted. MDG is able to handle these abstract sorts, cross-terms, and uninterpreted functions since they can be part of the graph structure as well as the MDG-HDL syntax [6].

This interface will ultimately allow the formal verification of ASM models using the MDG tool. Figure 9 shows an overview of the expected ASM-MDG verification procedure.

The proposed ASM-MDG tool consists of two complementary parts: the first part generates MDG-HDL behavioral models from ASM specifications, while the second part generates MDG-HDL structural 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.

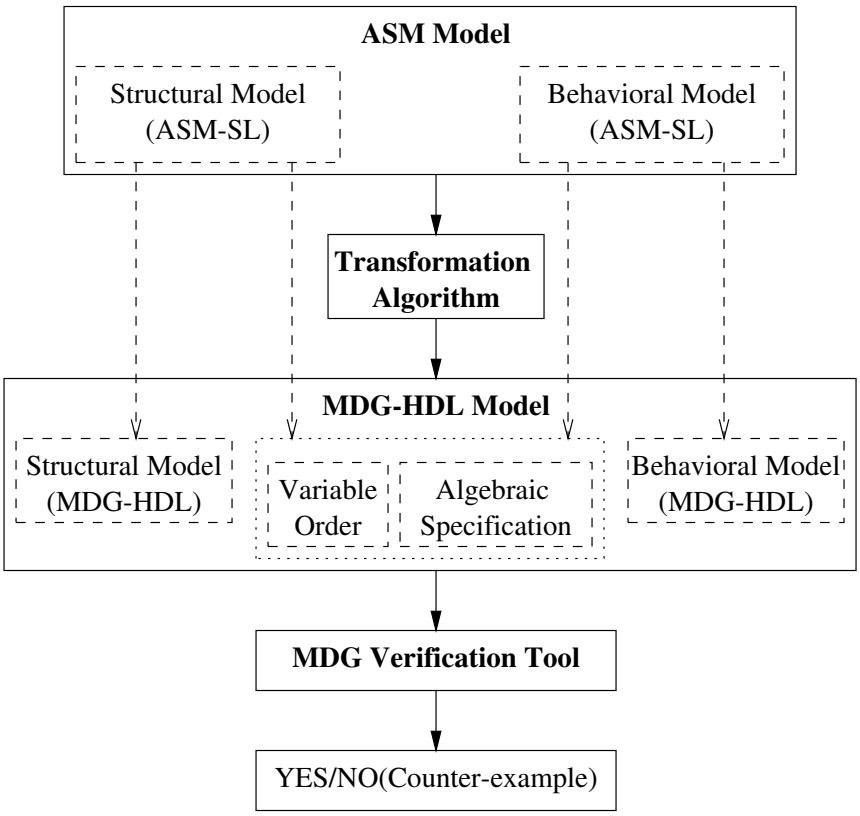

Figure 9: ASM-MDG verification procedure

5.1 ASM-MDG Interface for Behavioral Models

In order to provide a generic interface for the ASM-WB with different tools, ASM models are automatically translated into the ASM-IL as proposed in [25]. This interface allows various tools to be applied to the same language, which is ASM-SL. 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.

MDG specifications are represented by **tables** similar to truth tables. We create these tables by mapping ASM models through the intermediate language into MDG-HDL tables along with variable order and algebraic specifications as shown in Figure 10. To treat behavioral ASM-SL specifications, ASM models are first translated into the ASM-IL as shown in Figure 10. The model is first parsed for syntax check, ASM universes, functions, and transition rules are collected. Then

an analyzer generates the ASM Intermediate Language, 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 [24]. The behavior of the model is described as a set of guards and updates for each state variable (*update location*, the value of the state variable in the next state is the corresponding value to the satisfied guard in the list, otherwise it keeps the same value as in previous state. Based on this ASM-IL, MDG-HDL behavioral descriptions are generated in terms of tabular representation similar to truth tables. In addition, variable order and algebraic specifications are produced [12].

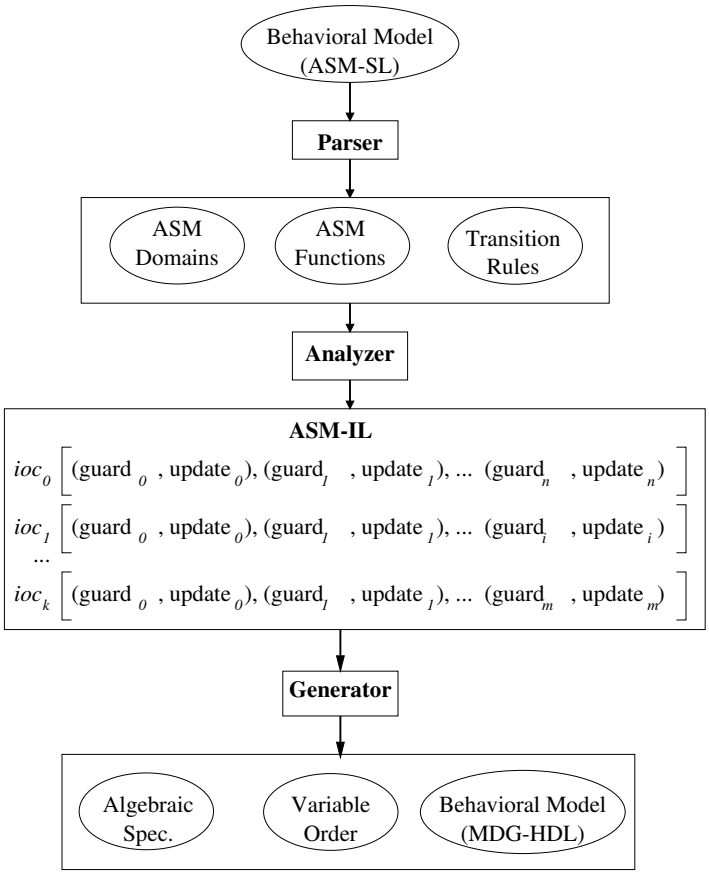

Figure 10: ASM-MDG internal interface for behavioral designs

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. 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. 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 11.

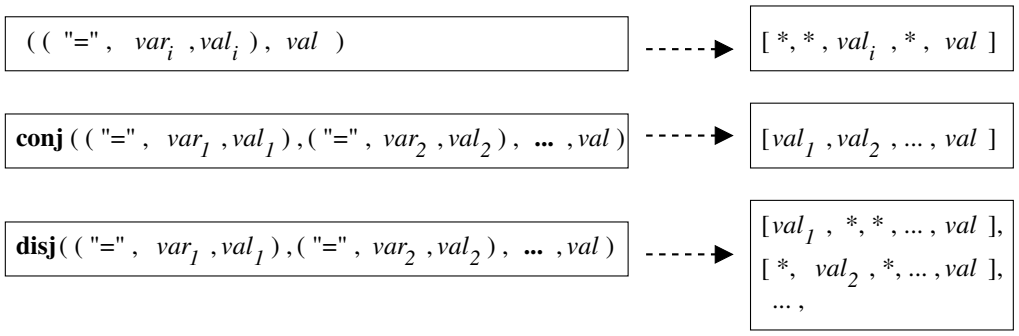

Figure 11: Creating MDG tables from guarded updates

5.2 ASM-MDG Interface for Structural Models

5.2.1 ASM-MDG Interface for Structural Models via ASM-IL

Starting from the ASM-IL language, we built our interface to the MDG tool as shown in Figure 12.

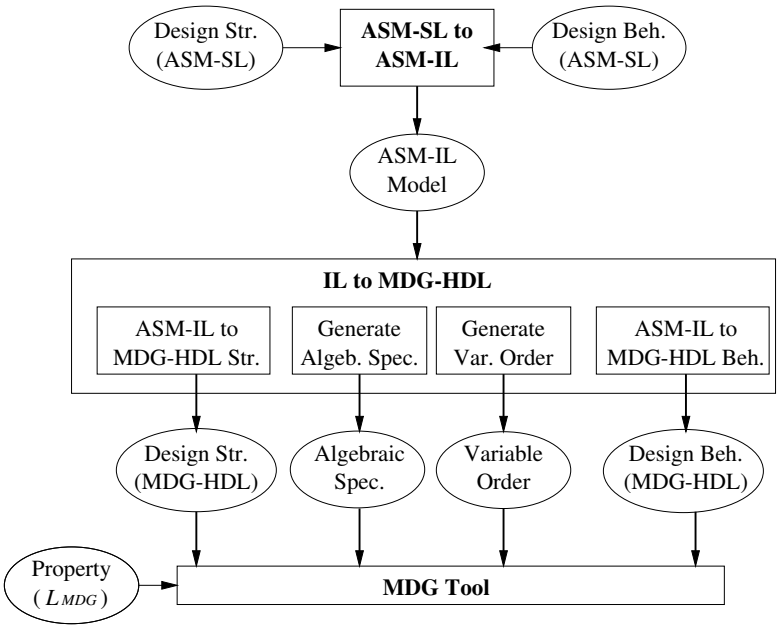

Figure 12: ASM-MDG interface via ASM-IL

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.

First, each location is mapped into a state component in MDG-HDL. Since locations are considered as 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 resultant 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 13. 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.

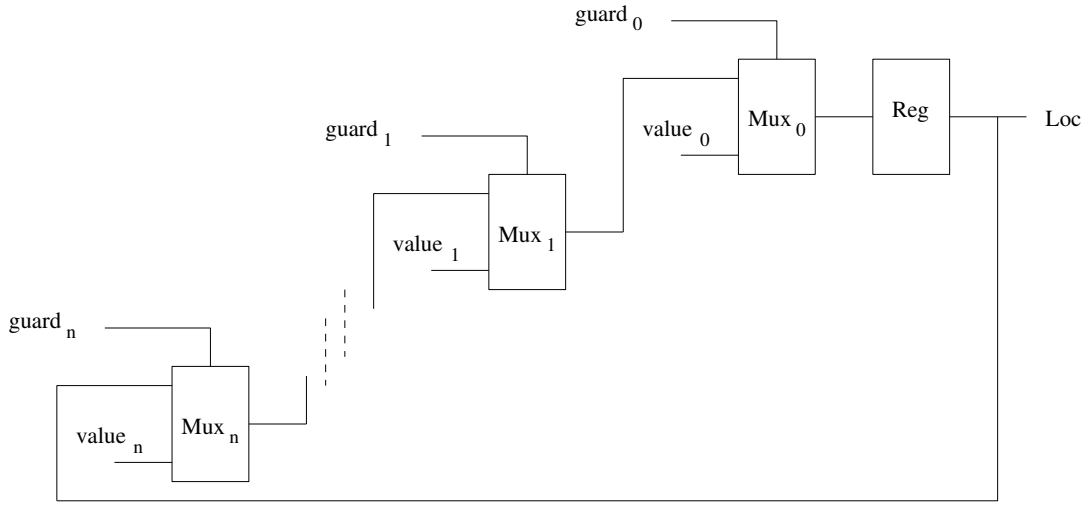

Figure 13: Mapping ASM-IL expression for one location into MDG-HDL

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 is simply mapped into a signal with the variable name. Equality expressions for a variable and the value false is mapped into the corresponding negation MDG-HDL component, **not**. Relational operators, 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**.

Relational operators can be used with different data sorts in ASM models, when they are used with abstract data sorts, they are mapped into a cross-operator. Operators which can be used with concrete data types other than Boolean, *equal*$(=)$ and *not equal*$(!=)$, are mapped into tables. The first table clearly indicates that the output signal of the table equals to *true* (1) when *var* equals to *val* and *false* (0) otherwise.

5.2.2 ASM-MDG Syntactic Transformation for Structural Models

For structural designs, ASM-IL does not preserve the structure of the original ASM model, because it does not provide 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. Moreover, large number of components results also in a large number of variables which makes it very hard to generate a good variable order. To solve this problem, we provide for structural designs a direct interface between ASM-SL and MDG-HDL without going through the ASM-IL. In order to keep this interface simple and feasible, we implement it for a set of predefined ASM functions without going into the semantics of those functions. In other words, we define ASM *static functions* that correspond to MDG-HDL primitive components, then we use them to built our ASM structural model, which then can be translated into MDG-HDL structural design easily.

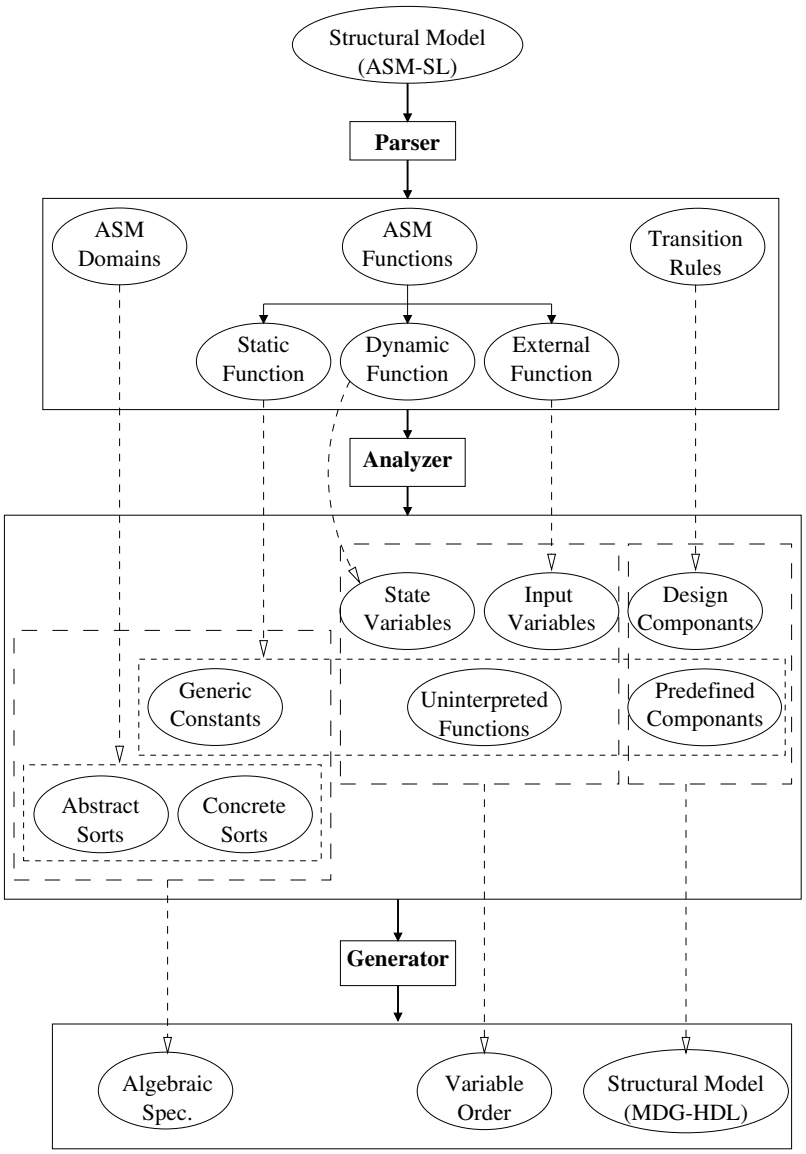

Figure 14: ASM-MDG internal interface for structural designs

Figure 14 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.

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 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 15 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.

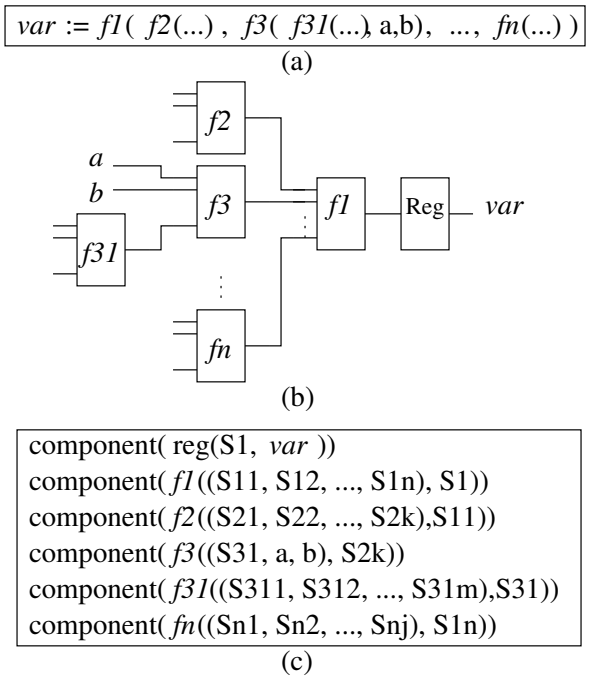

Figure 15: Mapping structural ASM-SL into MDG-HDL components

The MDG-HDL is generated by a one-to-one mapping from ASM structure of static functions to MDG-HDL library of components. The current implementation supports only a set of ASM functions that can be mapped directly to MDG-HDL, in addition to uninterpreted functions and cross operators.

5.2.3 Algebraic Specifications

We have to declare all data sorts and functions before we use them in our MDG-HDL models. 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 abstract or concrete sorts must be declared explicitly. An ASM-IL representation preserves the enumeration for each variable. Based on this, we declare a concrete sort for each different enumeration. Abstract sorts are declared according to the distinguished sorts used in the ASM-SL model.

All functions and cross terms are also declared in the algebraic specification in the same way. This includes uninterpreted functions, cross terms and relational operators. We declare any function that occurs in the ASM-IL expressions in the algebraic specification according to its arguments and target sorts. We find its target sort from the domain of the expression where it occurs.

5.2.4 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*. Figure 16 depicts these dependencies.

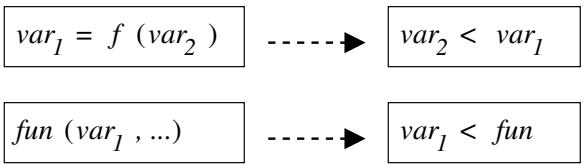

Figure 16: Variable order constraints

6 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 [31] 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 17. 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.

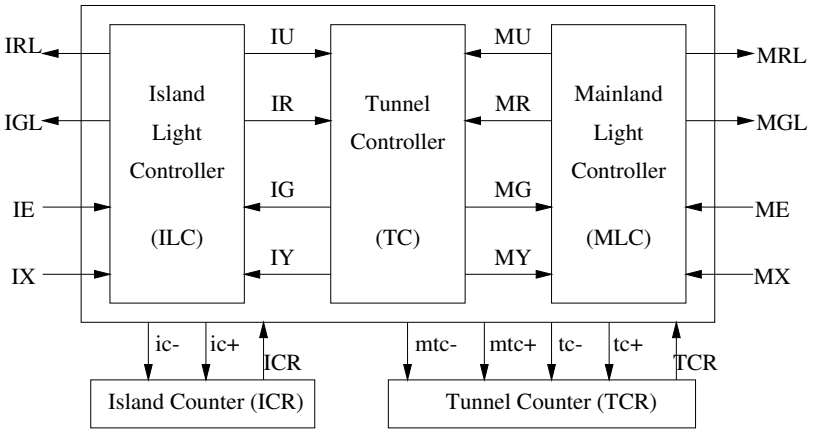

Figure 17: Three-Controllers design of the ITC

6.1 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 supported by our framework as we are able to verify this system for any arbitrary counter size. 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.

6.1.1 Behavioral Modeling in ASM

Figure 18 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.

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 term *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 17).

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
```

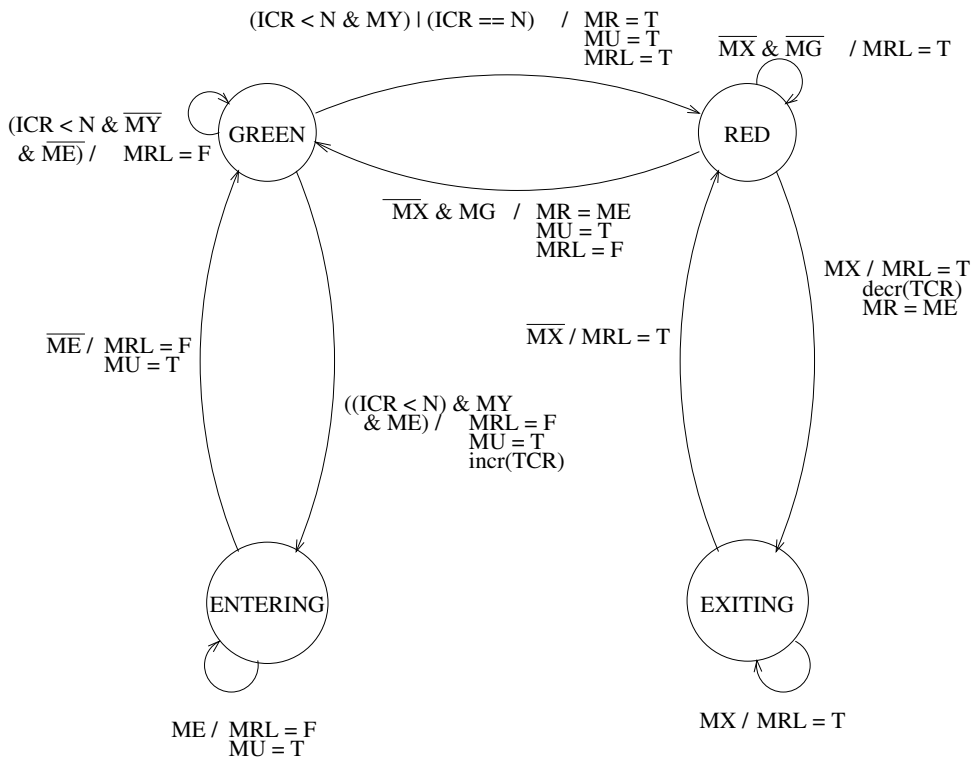

Figure 18: ASM transition system for the MLC

```
mainlandred := false
mainlandgreen := true
mainlandrequest := false
```

6.1.2 Structural Modeling in ASM

We developed a structural model (implementation) for the MLC model as shown in Figure 19. 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
```

This MLC design was derived from the specification given above in Figure 18. 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–term.

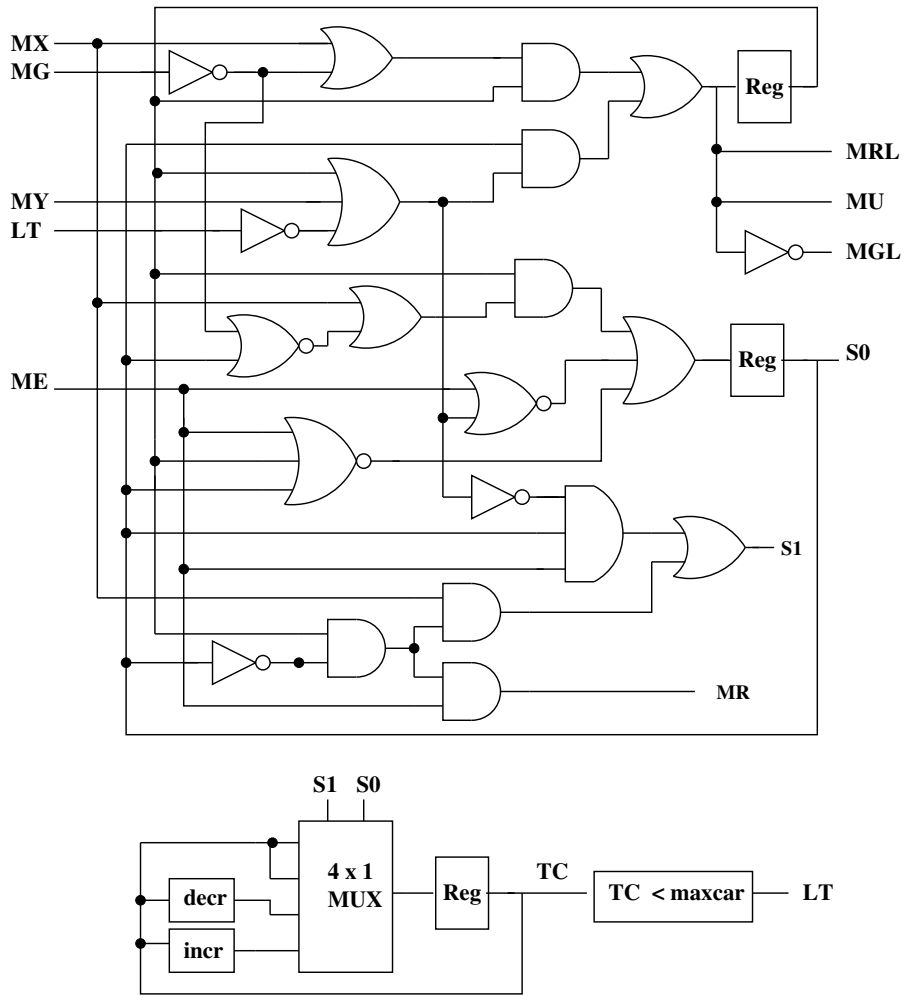

Figure 19: MLC Implementation

6.2 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 [2].

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

[2]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/

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.

Table 1: MDG 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

Next, a set of properties were specified in $\mathcal{L}_{\mathcal{MDG}}$ format, and then MDG model checking was applied to verify that the generated MDG-HDL models satisfy them all. In the following, we describe three properties on the MLC, including safety and liveness properties, where the symbols AG means "`for all paths, for all states`", F means "`there exists a state in the future`", & is the logical AND, and ! is the logical NOT. The same properties were specified for the ILC.

Property 1:`if the mainland is requising the controller, then then it will be using it at the future.`

This property is formally specified as following:
$AG((mainlandrequest\_B = 1) => (F(mainlanduse\_B = 1)));$

Property 2: `mainland green light and mainland red light should never be active simultaneously.`

and it is formally specified as following:
$AG(!((mainlandgreen\_B = 1)\ \&\ (mainlandred\_B = 1)));$

Property 3: `Mainland controller should never request the tunnel while it is using it.`

and it is formally specified as following:
$AG(!((mainlandrequest\_B = 1)\ \&\ (mainlanduse\_B = 1)));$

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

7 Conclusion and future work

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,

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

namely equivalence checking and model checking. On the other hand, MDG users will be 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 [12] 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, Mon-

treal, 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.), LNCS 1214, Springer-Verlag, 1997, pp. 202–212.

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

[5] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation, In IEEE Transactions on Computers, Vol. C-35, No. 8, August 1986, pp. 677–691.

[6] 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.

[7] 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.

[8] G. Del Castillo. The ASM Workbench: an Open and Extensible Tool Environment for Abstract State Machines, In Proceedings of the 28th Annual Conference of the German Society of Computer Science, Technical Report, Magdeburg University, 1998.

[9] 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.

[10] 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, 2000, pp. 331–346.

[11] Y. Feng and E. Cerny. Term Ordering Problem on MDG. In Proceedings of the ACM 12th Great Lakes Symposium on VLSI, New York City, New York, USA, April 2002, pp. 160–165.

[12] 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.

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

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

[15] 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, pp. 145–164.

[16] C. Kern and M. Greenstreet. Formal Verification in Hardware Design: A Survey, ACM Transactions on Design Automation of Electronic Systems, Vol. 4, April 1999, pp. 123–193.

[17] 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.

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

[19] M.L. McMillan. Symbolic Model Checking, Norwell, MA, USA, Kluwer, 1993.

[20] Y. Mokhtari, M. Shirazipour, and S. Tahar. A Case Study on Model Checking and Refinement of Abstract State Machines. In Proceedings of the Eigth International Conference on Computer Aided Systems Theory, Las Palmas de Gran Canaria, Canary Islands, Spain, February 2001, pp. 239–242.

[21] 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.

[22] 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.

[23] 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.

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

[25] K. Winter. Model Checking with Abstract Types, In Scott D. Stoller and Willem Visser (eds.), Workshop on Software Model Checking, Electronic Notes in Theoretical Computer Science, Vol. 55, Issue 3, July 2001.

[26] 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.

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

[28] Z. Zhou. Multiway Decision Graphs and their Applications in Automatic Verification of RTL Designs. PhD. Thesis, University of Montreal, Dept. of Information and Operational Research, 1997.

[29] 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, 2001.

[30] M.H. Zobair and S. Tahar. Formal Verification of a SONET Telecom System Block, In C. George, H. Miao (eds.), Formal Methods in Software Engineering, LNCS 2495, Springer Verlag, 2002, pp. 447–458.

[31] Z. Zhou, X. Song, S. Tahar, E. Cerny, F. Corella, M. Langevin. In M. Srivas and A. Camilleri (eds.), 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.