Interfacing Abstract State Machines with Multiway Decision Graphs

Amjad Gawanmeh

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science at

Concordia University

Montréal, Québec, Canada

April 2003

CONCORDIA UNIVERSITY

Division of Graduate Studies

This is to certify that the thesis prepared

By: **Amjad Gawanmeh**

Entitled: **Interfacing Abstract State Machines with Multiway Decision Graphs**

and submitted in partial fulfilment of the requirements for the degree of

Master of Applied Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Dr. Mohammad Soleymani

_____ Dr. Alagar Vasu

_____ Dr. Katarzyna Radecka

_____ Dr. Sofiène Tahar

Approved by _____

Chair of the ECE Department

_____ 2002 _____

Dean of Engineering

ABSTRACT

Interfacing Abstract State Machines with Multiway Decision Graphs

Amjad Gawanmeh

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 in this thesis 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. The proposed interface uses two steps: first, the ASM model is transformed into a flat, simple transition system as an intermediate model. Second, this intermediate model is transformed into the syntax of the input language of the MDG tool, MDG-HDL. We consider both structural and behavioral models of hardware. We have applied this transformation schema on some examples and case studies where our tool generates the corresponding MDG-HDL models automatically.

To my parents, sisters, and brothers.

ACKNOWLEDGEMENTS

I would start with special thanks to my research advisor, Dr. Sofiène Tahar for his support. His encouragements and believe in me was the main motivation for me during my research. Also, I would like to thank him for choosing me and giving me the opportunity to work with HVG (Hardware Verification Group) here at Concordia. I would also like to thank all the members of the HVG for their help and support while working on this project, and specially while writing this thesis. This work would not have been possible without the help of Kirsten Winter (SVRC, University of Queensland, Australia), so I would like to thank her for the effort she put on this project.

Amjad Gawanmeh

Montrèal

November 2002

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

ASM	Abstract State Machines
ASM-IL	Intermediate Language
ASM-WB	ASM Workbench
CTL	Computational Tree Logic
DAG	Directed Acyclic Graph
FSM	Finite State Machine
HDL	Hardware Description Language
HOL	Higher-Order Logic
ILC	Island Light Controller
ITC	Island Tunnel Controller
LTL	Linear Time Temporal Logic
MDG	Multiway Decision Graphs
ML	Meta Language
MLC	Main Land Controller
PVS	Prototype Verification System
ROBDD	Reduced Ordered Binary Decision Diagram
RTL	Register Transfer Lever
RTUML	Real Time Unified Modeling Language
SMV	Symbolic Model Verifier

TC	Tunnel Controller
TROM	Timed-Reactive Object Model
VIS	Verification Interacting with Synthesis
VLSI	Very Large Scale Integration

Chapter 1

Introduction

1.1 Motivation

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 [24, 26]. 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 [24]. Theorem proving, on the other hand, is not automatic approach, but it can be applied on larger systems.

MDGs (Multiway Decision Graphs) [12] 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 [41]. MDG tool can support verification of larger systems as different case studies show [3, 11, 36, 43, 45]. 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) [19] is a formal specification method for software and hardware modeling and provides a powerful means of modeling various kinds of systems. ASM has become a successful methodology for specifying and verifying complex systems [6]. 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, e.g., embedded

systems, protocols, hardware specifications, semantics of programming languages [6, 22]. It has been used both in academic and industry contexts. The wide group of users shows that there is interest in the language and, consequently, there is an interest in tool support. ASM models transition systems in a simple and uniform fashion and give these transition systems an operational semantics [37]. 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.

We propose to build a tool to interface the ASM-WB (ASM Workbench) [13, 14] 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 [37, 38]. At the same time, in contrast to the work in [38], 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 [28] 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) [37]. 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.

1.2 Introduction to Formal 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 [24]. 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

Theorem Proving

One of the earliest approaches to formal hardware verification was to describe both the implementation as well as the specification in a formal logic. The correctness result was then obtained by proving in the logic, that the specification and implementation were suitably related. Among the best known interactive theorem

provers are the Boyer-Moore Theorem Prover 'Nqthm' [7], PVS [30] and the Cambridge HOL System [18]. Unfortunately, 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

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 [8], typically Binary Decision Diagrams (BDDs) [8] or their derivatives, which are then structurally compared. Examples of combinational equivalence checking tools are Cadence Affirma and Synopsys Formality. 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. MDG [12] and VIS [9] are examples of sequential equivalence checking tools.

Model Checking

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. Examples of model checkers are SMV [27], VIS [9], SPIN [21], and FormalCheck [10].

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 [4].

1.3 Related Work

The ASM Workbench (ASM-WB) [14] was developed as a tool environment for parsing, type-checking, simulating and debugging of ASM specifications. Applying model checking algorithms on ASM and a generic interface for the ASM Workbench was introduced in [37]. Transformation algorithms are provided to transform ASM models into an Intermediate Language (ASM-IL) [37] which provides a general interface into different verification tools. Two approaches were suggested: the first is based on the process of transforming an ASM model into the language of a symbolic model checker, SMV. In the second approach, the model is represented as an MDG, then model checking for a subset of first–order branching time temporal logic ($\mathcal{L}_{\mathcal{MDG}}$ [39]) can be adapted. As part of the work in [37], a fully automatic interface with the SMV tool was implemented and supported with two case studies.

In contrast to the SMV model checker, the MDG tool provides a useful means for representing abstract models containing uninterpreted functions, where SMV supports neither abstract data types nor uninterpreted functions. This allows model checking on an abstract level at which the state explosion problem can in some cases be avoided. Although the latter approach in [38] already provides support for abstraction by exploiting the MDG data structure, it does not provide an interface to the actual MDG tool. Our work provides a tool that transforms the specification language ASM-SL [37] of the ASM Workbench, into the hardware description language MDG-HDL [41] of the MDG tool.

There exists some other work on model checking ASM specifications. Spielmann [35] 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 [5], 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 [34] and Katz and Grumberg [23] 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 [25], Kort *et al.* describe a hybrid formal hardware verification tool linking MDG and the HOL theorem prover obtaining the advantages of both verification paradigms, this makes it possible to delegate the verification of HOL subgoals to the MDG tool for automatic proof. The presented interactive proof system is used to automatically manage the proof as well as complete any proof interactively that is beyond the scope of the automated system. The verification of whole blocks in the hierarchy can be done automatically.

Other work on linking verification tools includes combining Voss-ThmTac system [20], in which Voss was interfaced with HOL as a tactic that could be called to perform a symbolic trajectory analysis to verify assertions about sequence of states. The power of this proof system comes from the very tight integration of the two

provers allowing the user to interact directly with them [20]. In [33], Schneider and Hoffmann described linking the SMV model checker to the HOL theorem prover by deeply embedding the SMV specification language in HOL. They described a translation procedure for converting LTL (Linear Time Temporal Logic) formulas to equivalent ω-automata and its implementation in the HOL theorem prover. This allows the usage of SMV as a decision procedure that can be conveniently called as a HOL tactic proof script. The conversion in general enables HOL users to directly verify temporal properties by means of HOL's induction rules.

A work on specifications and modeling language based on finite state machines is found in [1] where a Timed-Reactive Object Model (TROM) was introduced. TROM is an FSM augmented with ports, attributes, and timing constrains. Features of this formal modeling language includes support of non-determinism, information hiding and controlled refinement, and it models multiple reactions which maybe triggered by a single event. There exists a similar work in [29] on Real Time Unified Modeling Language (RTUML) for modeling real-time reactive systems and its mechanized verification within the PVS environment.

In summary, our work results in an ASM-specific solution which extends the interface framework of [15, 38] as well as [25] with another tool interface.

1.4 Scope of the Thesis

In this thesis we present an implementation of a proposed algorithm which interfaces Abstract State Machines (ASMs) with the Multiway Decision Graphs (MDGs) in order to use MDG tool to apply formal verification techniques on ASM models. We support this interface with a case study and provide results of applying model checking and equivalence checking.

The rest of the thesis is organized as follows: Chapter 2 is a brief introduction to ASM in terms of concepts, language, modeling and supporting tool. Chapter 3 is a brief introduction to MDG and its modeling language and verification procedures. Chapter 4 presents the transformation schema from ASM into MDG and how MDG components are generated from ASM-IL syntax, we show how to construct both structural and behavioral components from the ASM-IL model while preserving its semantics. We also show a direct syntactic transformation interface to treat ASM structural designs. In Chapter 5, we present the Island Tunnel Controller as a case study using the tool interface. Conclusions and ideas on future work are presented in Chapter 6.

Chapter 2

Abstract State Machines

Abstract State Machines (ASM) [19, 22] 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 [22].

2.1 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.

2.1.1 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.

- ***Derived functions*** whose interpretation in each state is a function of the interpretation of the dynamic and external function names in the same state. Derived functions depend on the internal state and on the environmental situation (like the output of a Mealy Machine). They represent the view of the system state as accessible to an external observer.

2.1.2 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.

2.1.3 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.

2.1.4 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 R_1 else R_2

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.

2.2 Modeling with ASM-SL

The ASM Specification Language (ASM-SL) [14] 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.

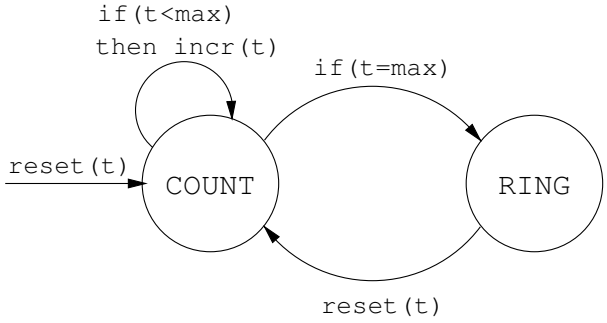

Figure 2.1: Generic counter state machine

We show the example of a *generic counter* [38] to illustrate the use of ASM in systems modeling. Figure 2.1 shows the state machine and Figure 2.2 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.

```
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 2.2: ASM modeling of the generic counter example

2.3 The ASM Workbench

The ASM Workbench (ASM-WB) [14] 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 [31], 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. (Figure 2.3 shows an example session with `sml-asm`.)

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 [15]. Since our work is built on preliminary work [37, 38], we can furthermore exploit the notion of *abstract types*. This feature can be essential when applying automated verification techniques like model checking. Figure 2.4 shows the ASM-WB interface [37].

```
- [flash] [/project/hvg/tools/trafo/ASM2MDG]> ../bin/sml-asm
val it = true : bool
- ASM.reset();
val it = () : unit
- ASM.load_file' "./ASM/timer.asm";
val it =
["DATA","MODE","Bool","Data","Mode","max_time","zero","mode",
"t","incr","R1","R2",...]: string list
- ASM.eval_term' "max_time";
val it = CELL ("abstract",[])
:(ASM_Domains.FINITE_SET,ASM_Domains.FINITE_MAP)ASM_Domains.VALUE'
- ASM.show_value(it);
val it = "abstract" : string
- ASM.eval_term' "mode";
val it = CELL ("count",[])
 :(ASM_Domains.FINITE_SET,ASM_Domains.FINITE_MAP)ASM_Domains.VALUE'
- ASM.show_value(it);
val it = "count" : string
- ASM.eval_term' "t";
val it = CELL ("abstract",[])
 :(ASM_Domains.FINITE_SET,ASM_Domains.FINITE_MAP)ASM_Domains.VALUE'
- ASM.show_value(it);
val it = "abstract" : string
-
```

Figure 2.3: Example session with sml-asm.

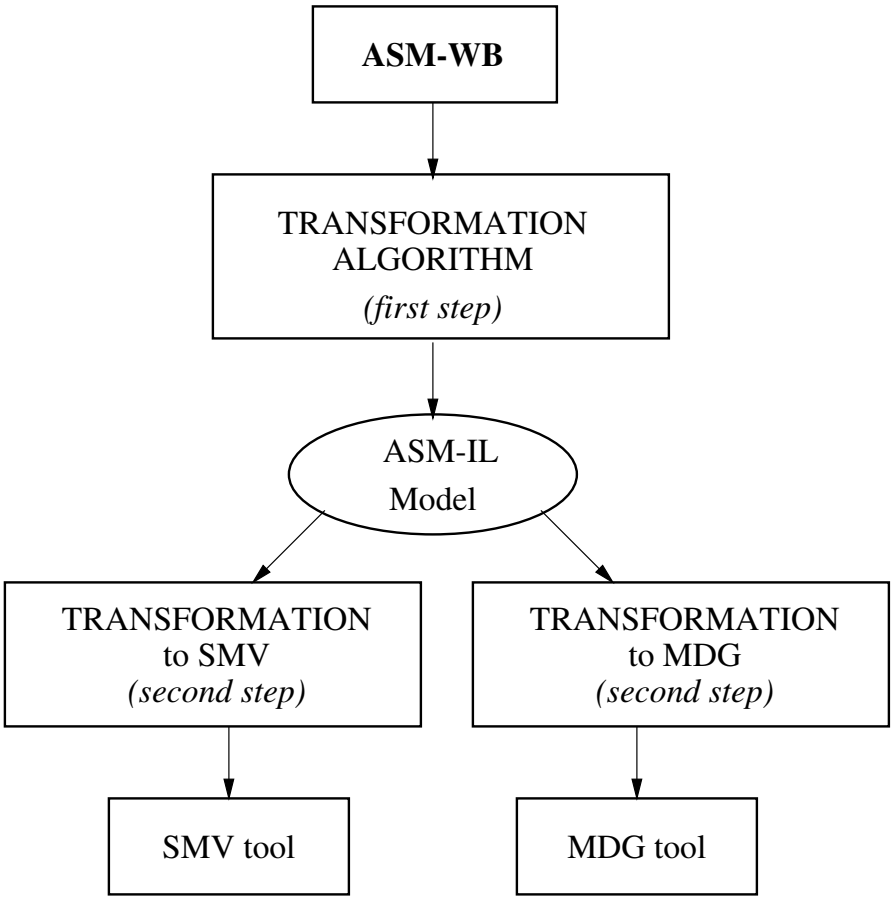

Figure 2.4: The ASM-WB interface

Chapter 3

Multiway Decision Graphs

Multiway Decision Graphs (MDGs) [12] have been proposed as a solution to the state space explosion problem of ROBDD (Reduced Order Binary Decision Diagrams) [8] 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.

3.1 Multiway Decision Graphs (MDGs)

MDG [12] 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 $\mathbf{T}$, 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).

3.2 Modeling with MDG

Logic gates can be represented by MDGs similarly to ROBDDs, because all inputs and outputs are of Boolean type. Figure 3.1 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 [42]. For example, Figure 3.2 shows the MDG of an arithmetic logic unit (ALU), where *op* is a concrete variable with enumeration sort {0,1,3,4}, *x1, x2* and *y* are abstract variables, *zero* is a generic (abstract) constant of the same sort, and *sub, add* and *inc* are uninterpreted functions.

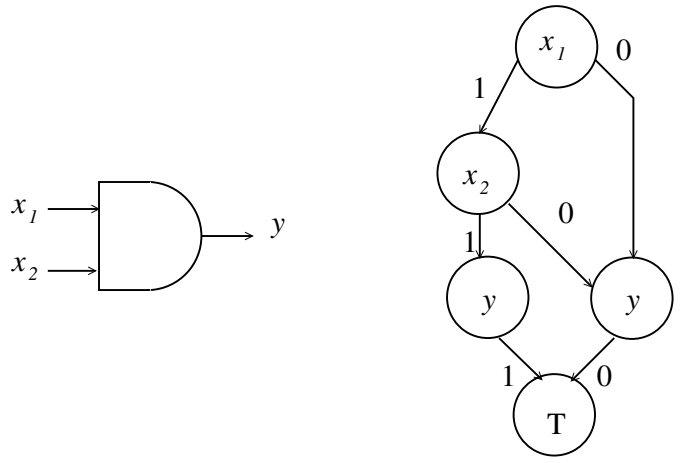

Figure 3.1: MDG for an AND gate

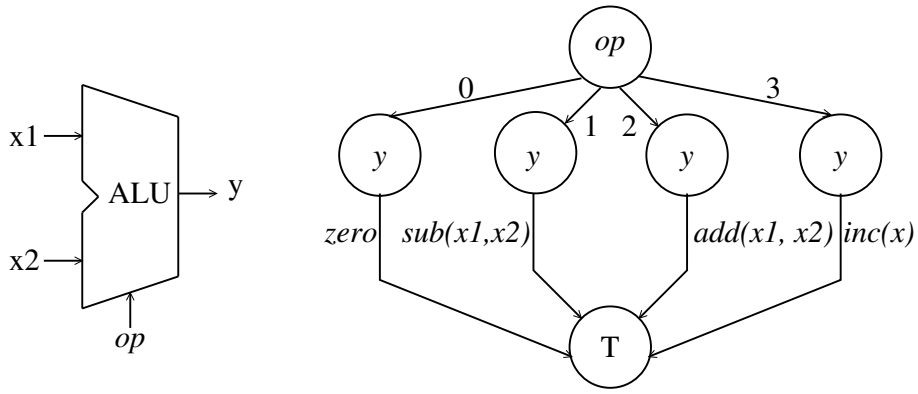

Figure 3.2: MDG for an ALU

For system descriptions the MDG tool comes with a Prolog-style hardware description language called MDG-HDL [41]. It allows the use of abstract as well as concrete variables for representing 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 3.3 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.

[1] In the MDG literature [12] 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)*

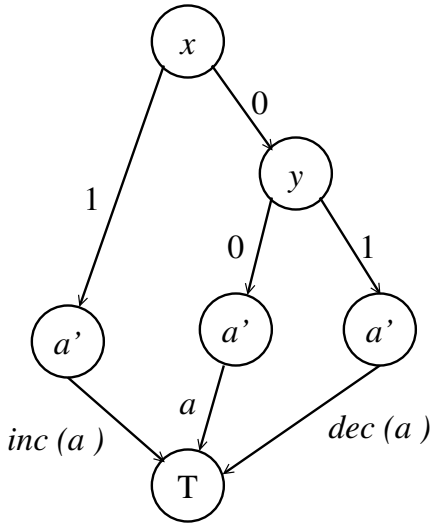

Figure 3.3: Table and MDG for a simple behavioral state machine

3.3 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 [12], invariant checking [12] and model checking [39]. The MDG tool has been used to verify a number of non-trivial systems such as communication switches and protocols [3, 11, 36, 43, 45].

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 [12]. A Model checking facility has also been recently developed [39] and incorporated into the existing MDG tool. This provides both safety and liveness property checking using implicit abstract enumeration [12]. The properties are represented in a universally quantified first-order branching time temporal logic, called $\mathcal{L}_{\mathcal{MDG}}$ [12]. 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 3.4 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 3.4). Moreover, an algebraic specification is to be given to declare sorts, function types, and generic constants that are used in the MDG-HDL description. Rewrite rules that are needed to interpret function symbols should be provided here as well. Like for ROBDDs, a symbol order according to which the MDG is built should be provided by the user. 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 [16].

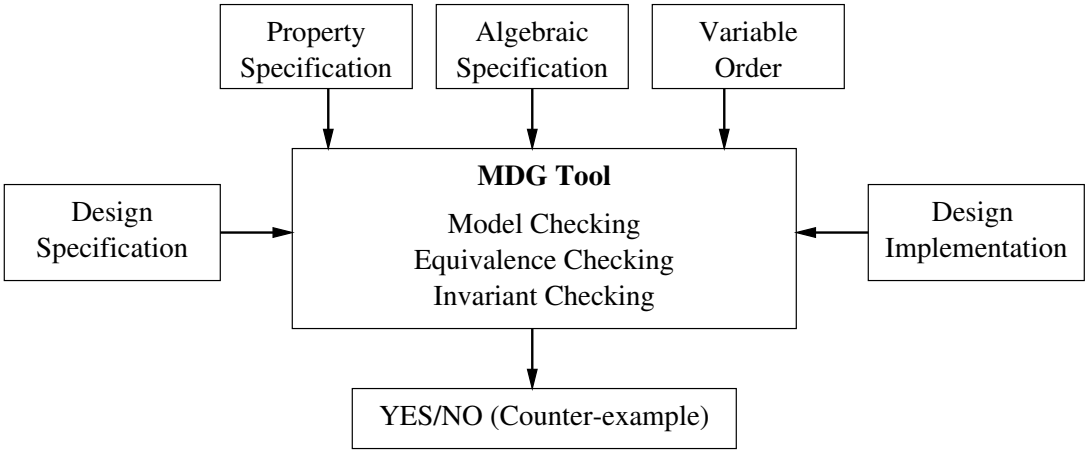

Figure 3.4: MDG verification tool

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 [2]. 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 [44].

Chapter 4

Interfacing ASM with the MDG Tool

Our main objective of this work is to provide the MDG tool with a high-level modeling language, namely ASM, because this notion is becoming widely used to describe different types of systems. We choose to interface ASM with the MDG tool because it contains different automatic verification techniques and both notions are very close to each others as both are based on first-order logic to support abstract data types and uninterpreted functions. This interface will ultimately allow the formal verification of ASM models using the MDG tool. Figure 4.1 shows an overview of the expected ASM-MDG verification procedure.

In this chapter, we describe in details the proposed "ASM-MDG" interface. We will show how different behavioral and structural MDG-HDL models are generated

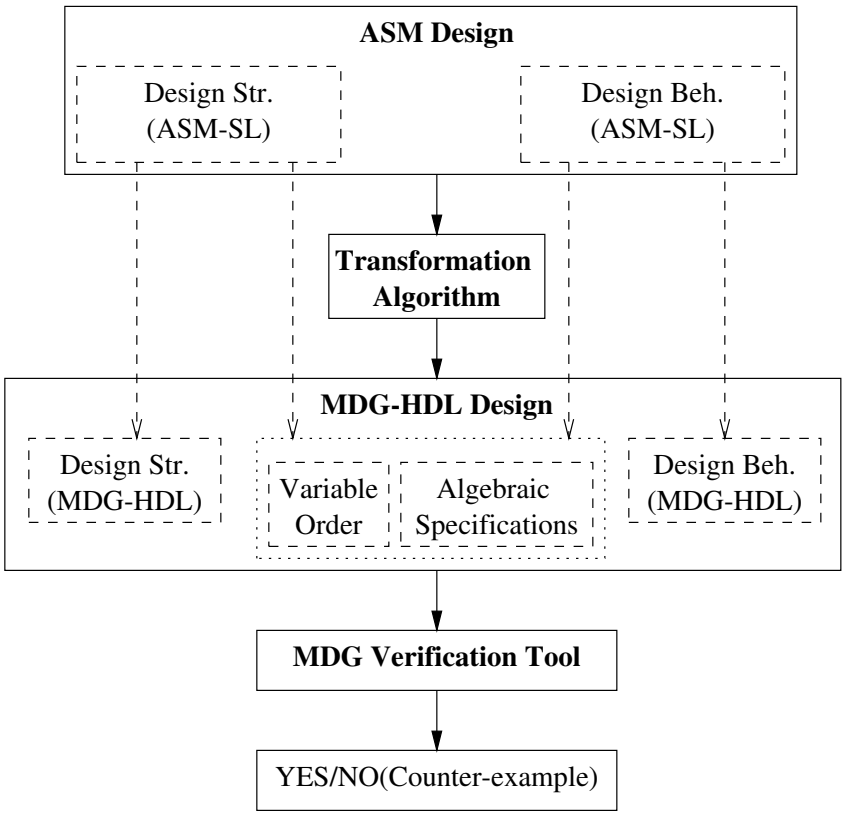

Figure 4.1: ASM-MDG verification procedure

from ASM-IL via the ASM-IL as well as in a syntactic direct fashion.

4.1 ASM-MDG Interface via ASM-IL

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 [38]. 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 this section we describe how ASM-IL is coded in MDG-HDL. First, we show how to generate the MDG-HDL structural netlist of components that represents MDG-HDL implementation model, and then we show how to generate the MDG-HDL tables that represent MDG-HDL specification model.

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

Starting from the ASM-IL language, we built our interface to the MDG tool as shown in Figure 4.2. The interface automatically generates the required tool inputs: MDG-HDL design models (structure and/or behavior), MDG algebraic specification, and MDG symbol order. In the next subsections, we describe the details for each.

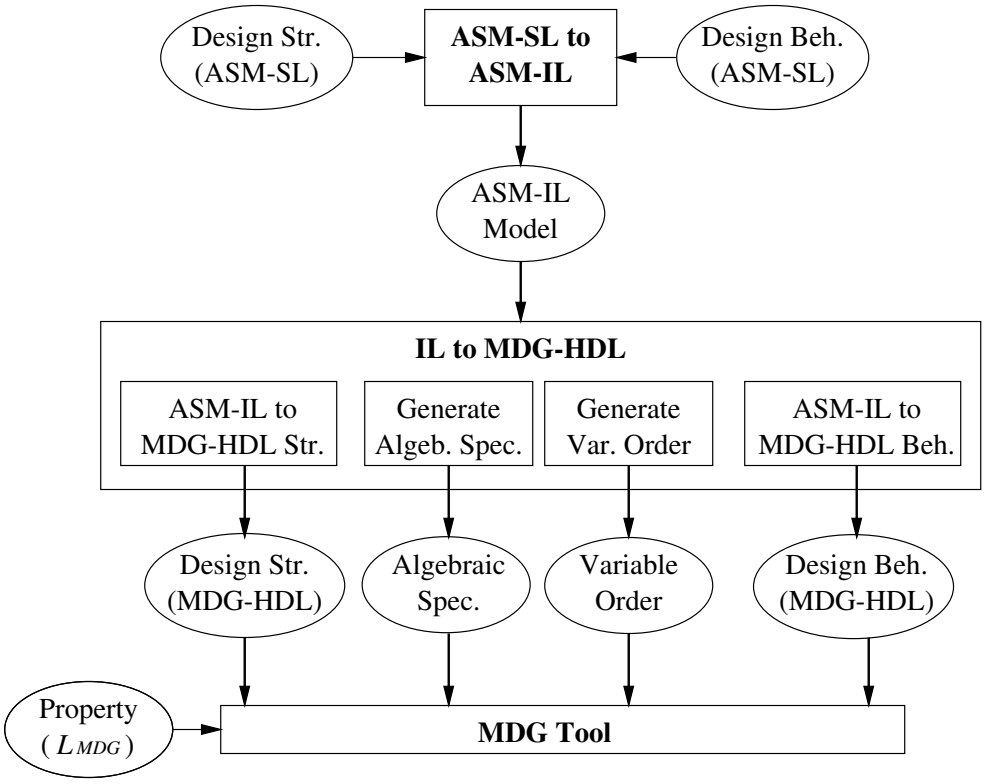

Figure 4.2: ASM-MDG interface via ASM-IL

4.1.1 MDG Structural Description

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

Location

In an ASM-IL representation each location is associated with a set of guarded updates, each consisting of a Boolean guard and an update value. The whole expression

evaluates into one value, $value_i$, which is the next state value of the dynamic location, if there is at least one guard satisfied. Otherwise, the value of the location will be the same in the next state. Figure 4.3 shows the interpretation of guarded updates.

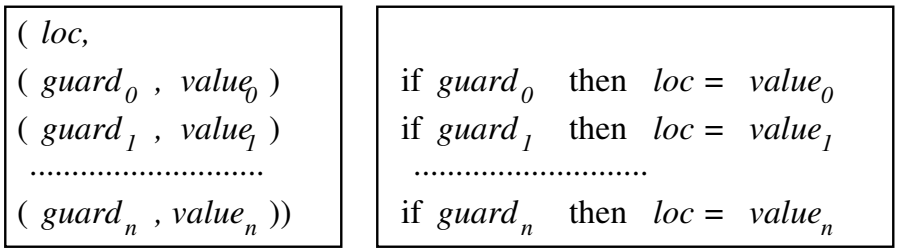

Figure 4.3: ASM-IL guarded updates

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 (see Figure 4.4). We connect these multiplexers together

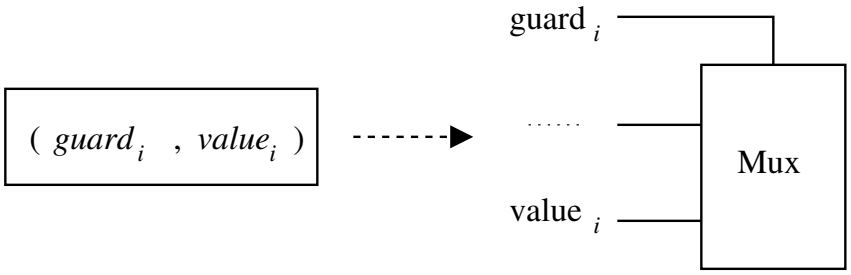

Figure 4.4: Mapping a guarded update into MDG-HDL

in a hierarchical way as shown in Figure 4.5. 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.

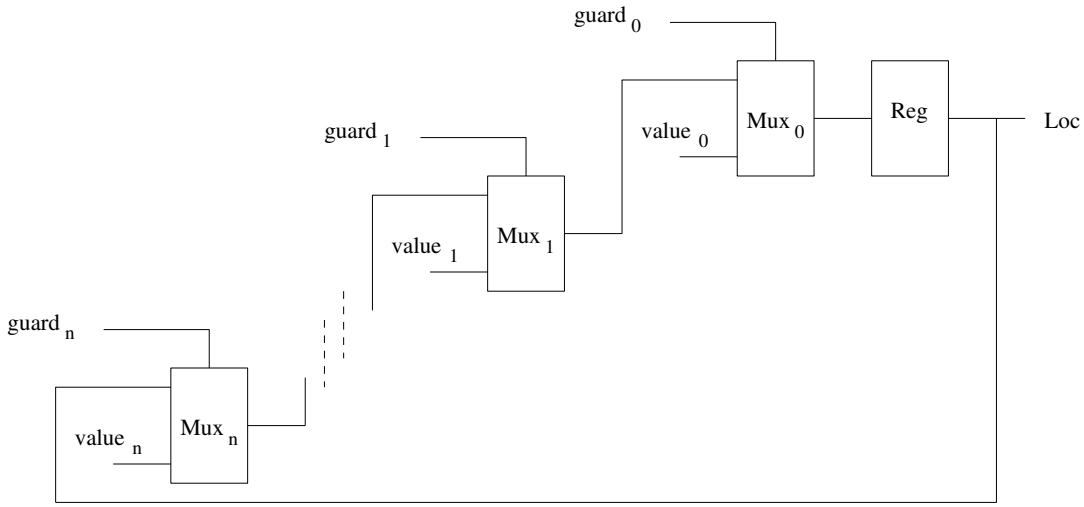

Figure 4.5: Mapping ASM-IL expression for one location into MDG-HDL

Values

Values can be locations, constants, or any variable. If a value is a location or a variable, we map it directly into a signal with the same type. Constants are mapped into an MDG-HDL component called **constant_signal**, that is a component with an

individual value from the enumeration of the concrete data sort or a generic constant in case of an abstract type. For Boolean types we use two default components that have always the values of 0 and 1.

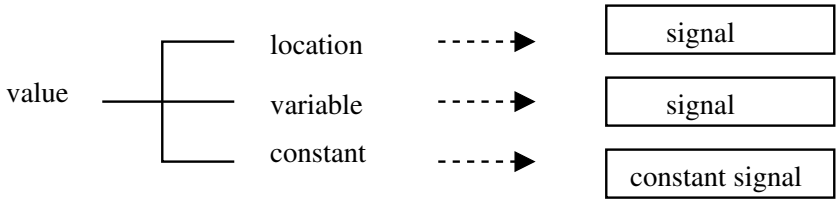

Figure 4.6: Mapping values into MDG-HDL

Guards

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

All default binary operators are mapped into MDG-HDL logic gates. An equality expression for a variable and the value true 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**.

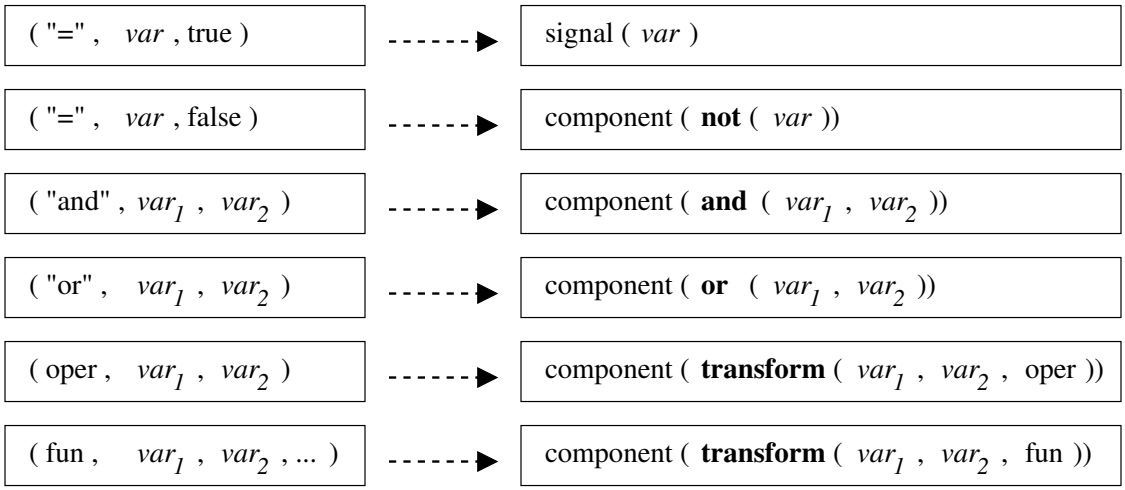

Figure 4.7: Mapping guards into MDG-HDL

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 according to Figure 4.8. Operators which can be used with concrete data types other than Boolean, $equal(=)$ and $not\ equal(!=)$, are mapped into tables according to Figure 4.9. 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.

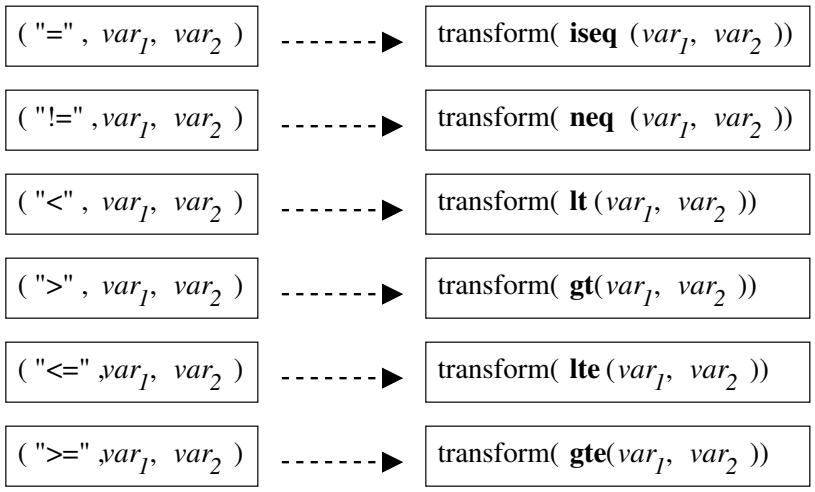

Figure 4.8: Mapping relational operators into MDG-HDL functions

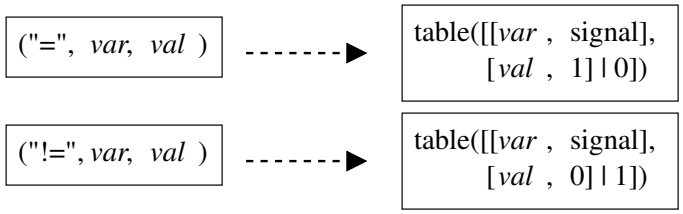

Figure 4.9: Mapping relational operators into MDG-HDL tables

4.1.2 MDG Behavioral Description

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 4.10. To treat behavioral ASM-SL specifications, ASM models are first translated into the ASM-IL as shown in Figure 4.10. 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 [37]. 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 [17].

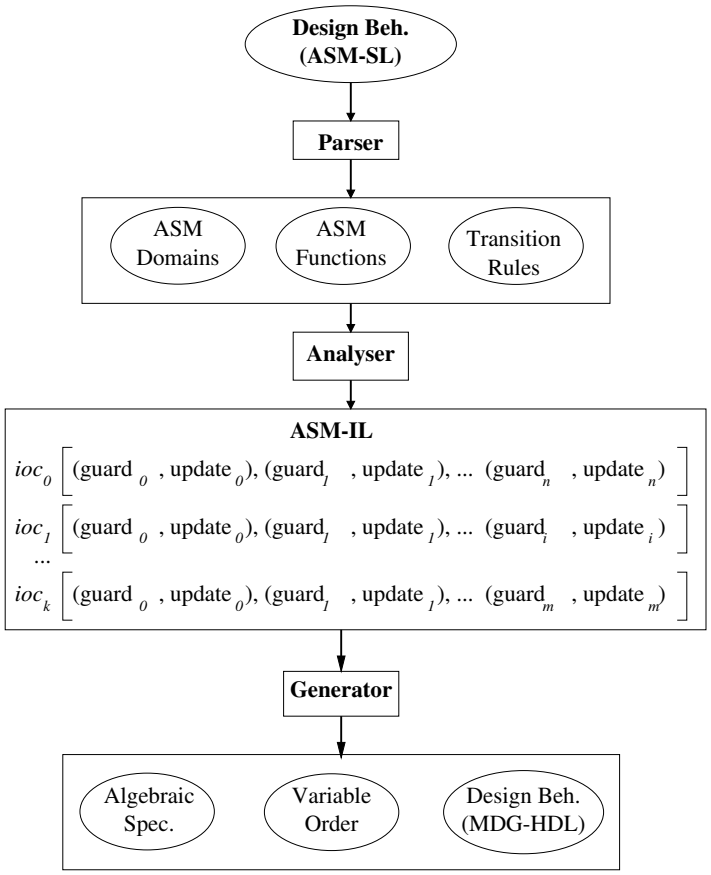

Figure 4.10: ASM-MDG internal interface for behavioral designs

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 (see Figure 4.11). 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 4.11.

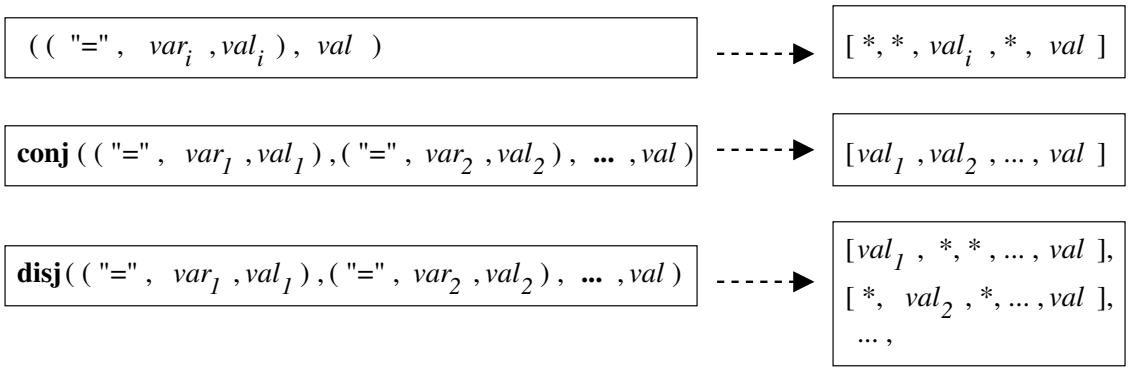

Figure 4.11: Creating MDG tables from guarded updates

In case we have nested operations in the ASM-IL model, we treat them recursively until we find a term, a constant value or a cross-operator. (See Figure 4.12)

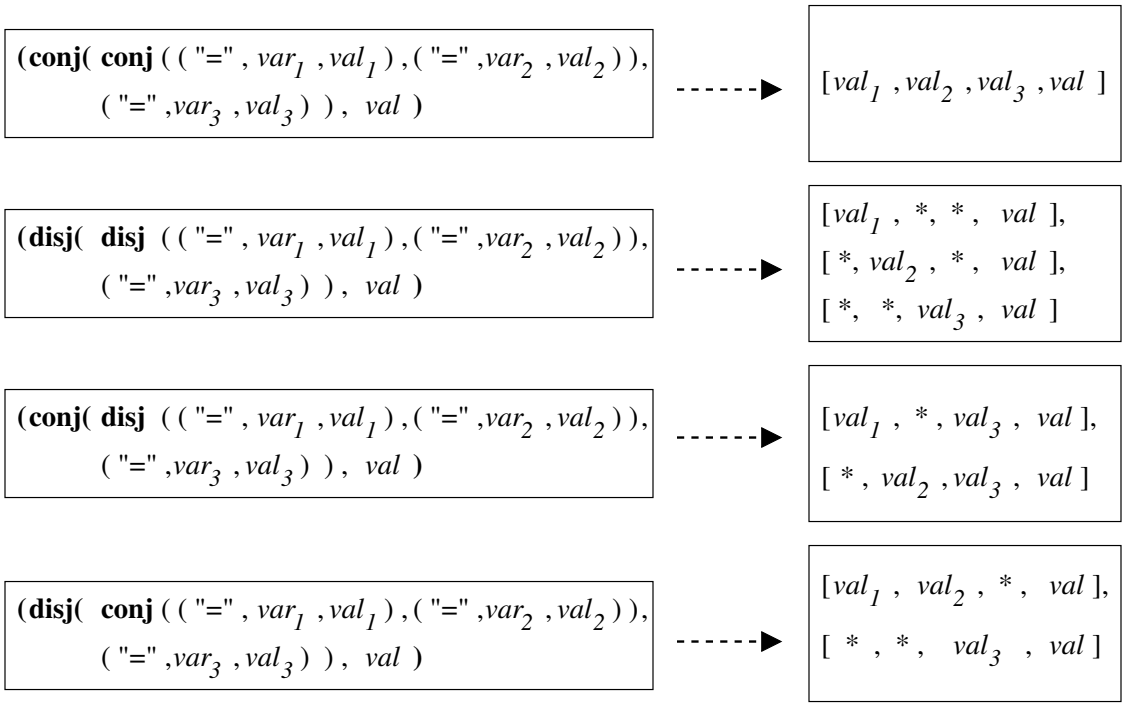

Figure 4.12: Mapping nested guarded updates into MDG-HDL tables

4.1.3 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. Figure 4.13 illustrates the mapping function that we use to generate the MDG algebraic specifications.

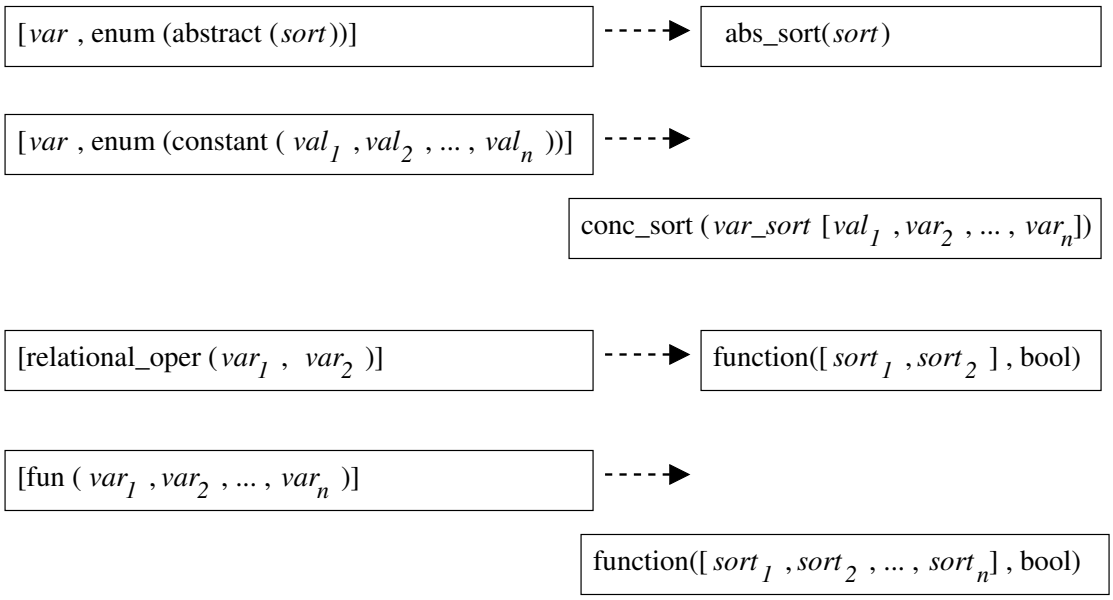

Figure 4.13: Declarations of functions and sorts in the algebraic specifications

4.1.4 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 4.14 depicts these dependencies.

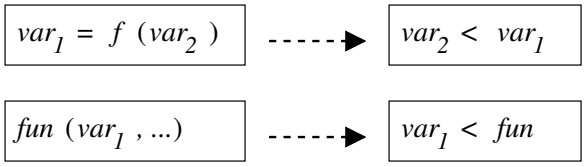

Figure 4.14: Variable order constraints

4.2 ASM-MDG Direct Interface

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 4.15 shows the proposed ASM-MDG direct interface.

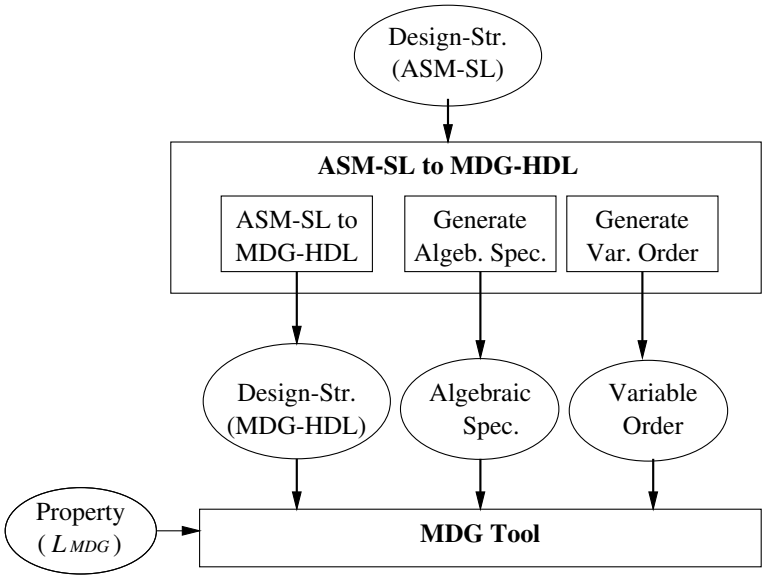

Figure 4.15: ASM-MDG direct interface for structural designs

This direct interface is implemented in three steps; a pars, an analyzer and a generator. Figure 4.16 details the above procedures, where the parser is used to check the input model and validate its syntax, it collects ASM universes including all sorts declarations, ASM functions including static, dynamic and external functions, and transition rules that describe the structure of the model. The analyzer is used to treat the data structures produced by the parser in order to construct design components, variables, functions and sorts that represent the design. In the last step, the generator is used to produce MDG-HDL models 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 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.

4.3 Summary

The work that is introduced in this chapter provides an interface from the ASM Workbench to the MDG tool. The implementation of the provided algorithm was done in standard ML in order to ease the interface to the ASM Workbench which is also implemented in ML. The program outputs a set of MDG-HDL files that represent the ASM model. This includes MDG-HDL structural model, MDG-HDL behavioral model, algebraic specifications, and variable order. These MDG-HDL models can easily be used as input for MDG tool to verify designs. In the next chapter, we will illustrate via a case study the application of the ASM-MDG interface.

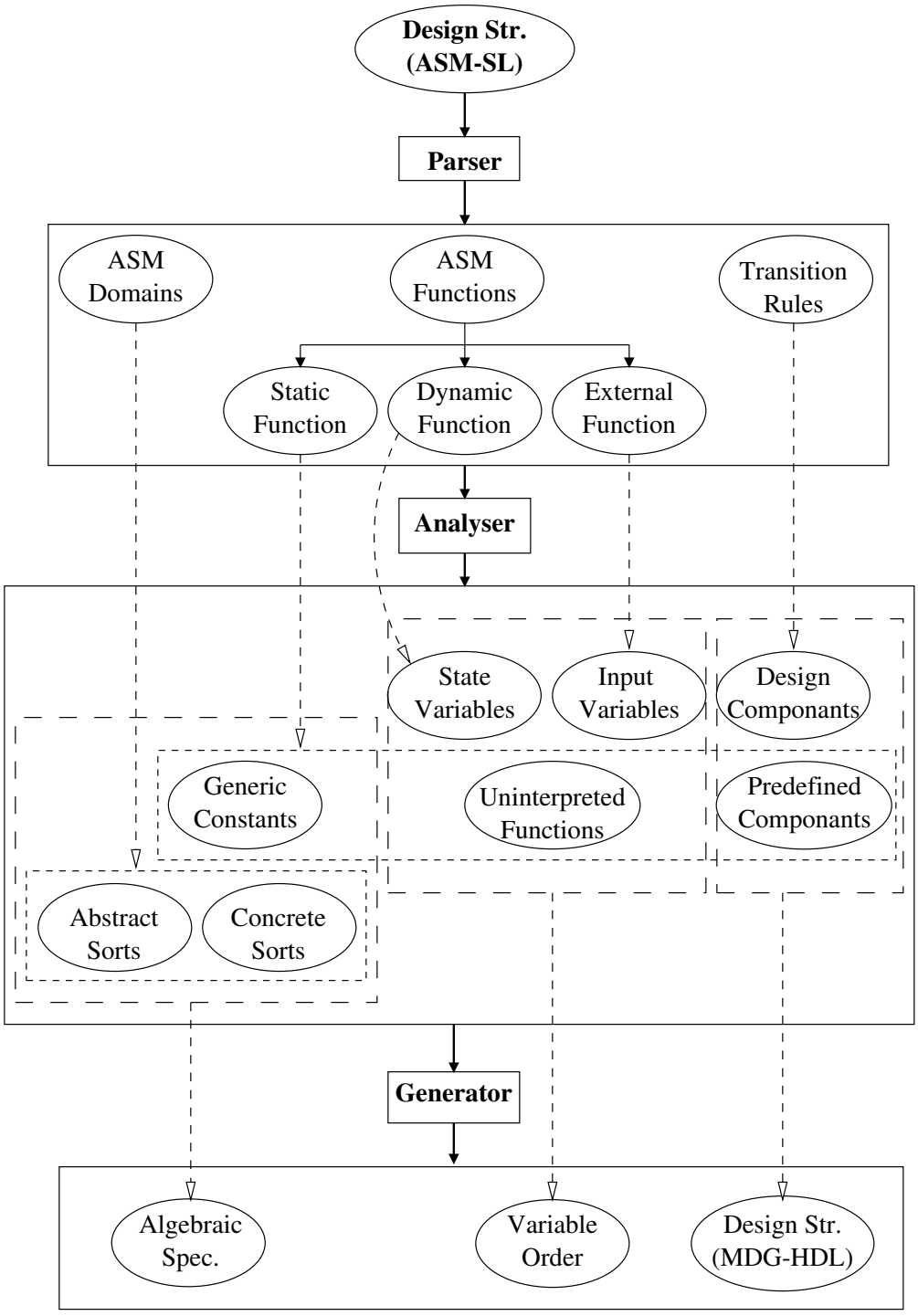

Figure 4.16: ASM-MDG internal interface for structural designs

Chapter 5

Application: Island Tunnel Controller

In this chapter, we provide a case study application of our ASM-MDG interface based on the Island Tunnel Controller (ITC) example [45] in order to illustrate the proposed ASM-MDG interface. The ITC is used to control two traffic lights for a tunnel that connects an island to the mainland as shown in Figure 5.1. 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: IE at the entrance of the island in ASM), IX at the exit of the island, ME at the entrance of the mainland and MX at the exit of the mainland. There are four output signals to control the traffic lights at both sides: IRL for island red light, IGL

for island green light, MRL for mainland red light and MGL for mainland green light.

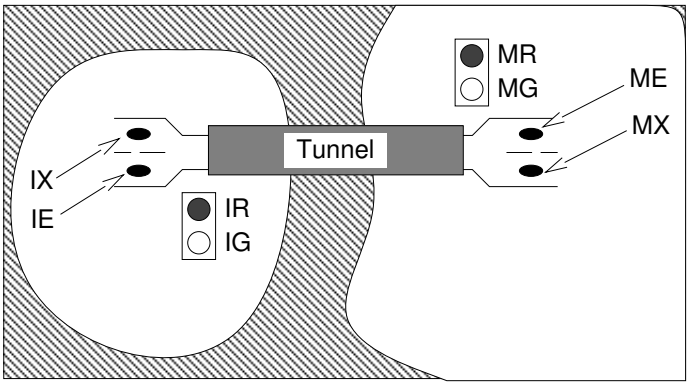

Figure 5.1: Island Tunnel Controller

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 5.2. Following signals are used between the controllers: IU indicates that the island is using the tunnel, IR indicates that the island is requesting the tunnel; IY indicates that the island is being instructed to release the tunnel for the mainland; and IG indicates that the island has been granted control of the tunnel from the mainland. A similar set of signals has been defined for the mainland. The Tunnel Counter (TCR) counts the number of cars inside the tunnel, and the Island Counter (ICR) counts the number of cars on the island. For TC, at each clock cycle, the count TCR is either incremented depending on the signals itc+ and mtc+, decremented

depending on signals itc- and mtc- unless it is already zero, or keep its value otherwise. The counter ICR is incremented and decremented the same way depending on the signals ic+ and ic-. 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.

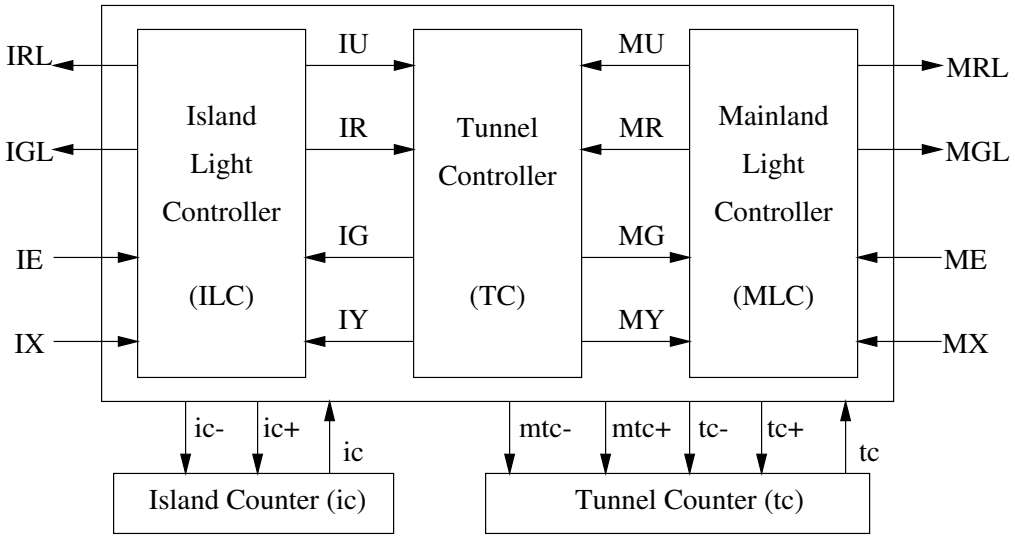

Figure 5.2: Three-Controllers design of the ITC

In the following sections, we will describe the ASM behavioral and structural modeling of the ITC, the transformation of the ASM models into MDG-HDL and finally their verification with the MDG tool [1].

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

5.1 ASM Modeling

The maximum number of cars to be in 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 "$ICR < n$". This allows modeling the controller for any number of cars. This example clearly illustrates 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 developed two models for each of the Island Light Controller (ILC) and the Mainland Light Controller (MLC): a behavioral model (specification) and a structural model (implementation). MLC is described in details below, along with its ASM models. Since the ILC is similar, we just show the behavioral and structural models without much detailed description.

5.1.1 Behavioral Modeling in ASM

Figure 5.3 shows the state transition diagram for the MLC, where & means logical AND, | means logical OR, and the bar above the variable means complement. The state transition diagram is assumed to be initially in the *red* state, where IRL is set to 1 while in this state. If a car is detected to be exiting the island through the tunnel, the controller goes to the *exiting* state and stays there while there are cars exiting from the tunnel, the tunnel counter (TCR) is decremented by activating tc- for every car. The controller goes back to the *red* state when there are no cars

exiting. While in the *red* state, if the mainland has been granted the control of the tunnel and there are no cars detected to be exiting, then the controller goes to the *green* state, IRL is reset to 0, and IGL is set to 1.

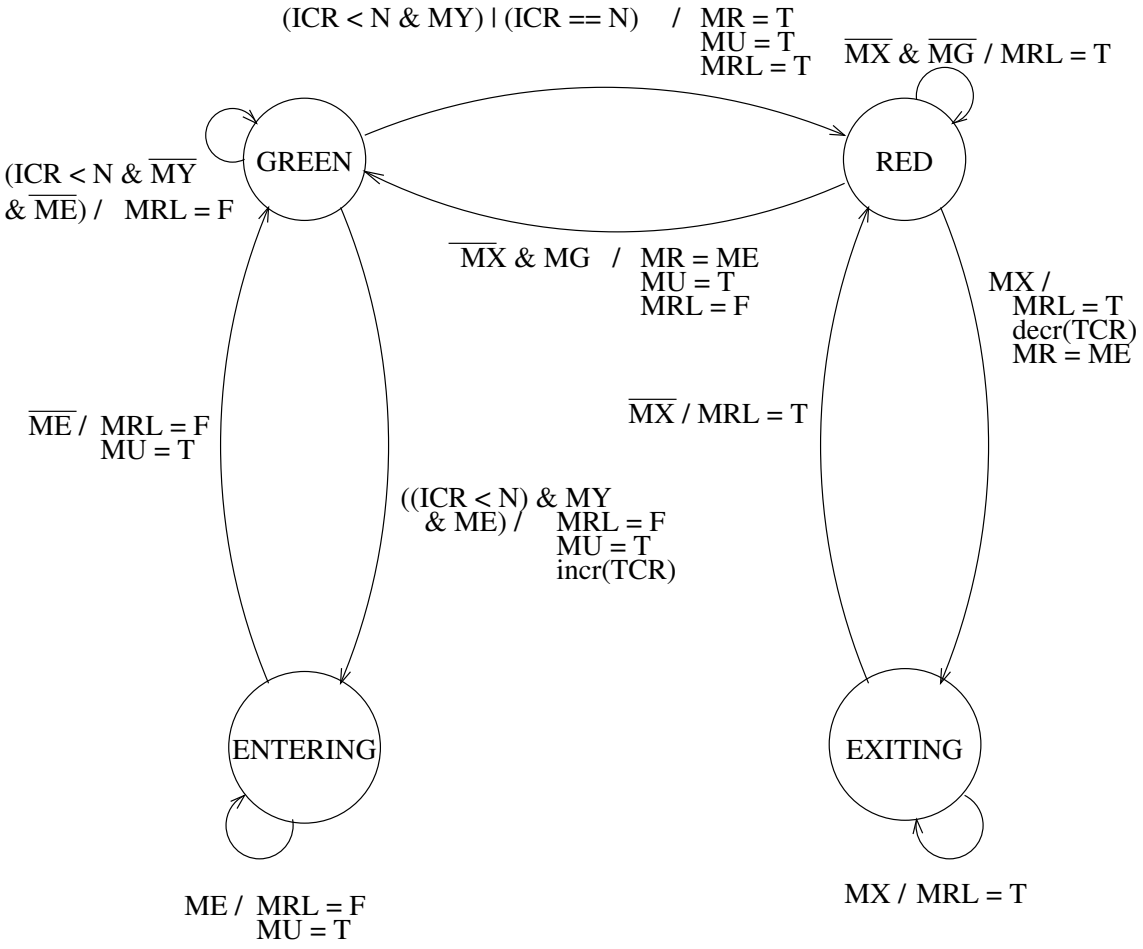

Figure 5.3: ASM transition system for the MLC

While in the *green* state, if the mainland controller is being instructed to release the tunnel for the island, the controller goes back to the *red* state and accordingly sets IRL to 1 and resets IGL to 0. Otherwise, it stays in the *green* state unless there

are cars entering to the tunnel, it goes in this case to the *entering* state and stays there while there are cars detected to be entering to the tunnel, as a result tc+ and ic- are activated to update the count each time a car is detected. If there are no cars entering, the controller goes back to the *green* state. Counters increment and decrement signals are assumed to be reset to 0 where it is not mentioned that they are set to 1. In the same way as MLC, the ILC behavior is described in Figure 5.4.

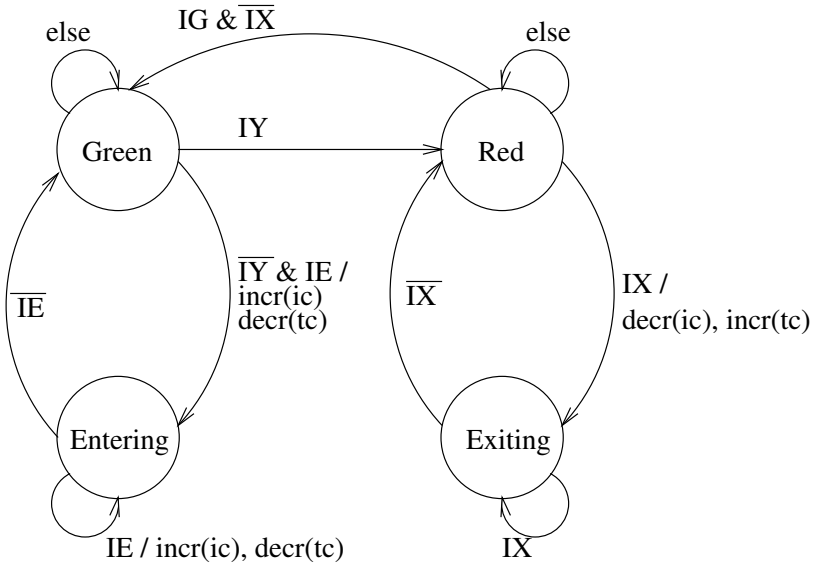

Figure 5.4: State Transition for the ILC

We model this behavior in ASM by defining a free type that represents the states of the MLC as follows:

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

Increment and decrement operations on the counters are generally infinite

mappings over integers. In our model, we specify those as abstract *static functions*, which map an abstract value to an abstract value. 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* as follows:

```
static function incr == MAP_TO_FUN {abstract -> abstract}
static function decr == MAP_TO_FUN {abstract -> abstract}
static function lt == MAP_TO_FUN {abstract*abstract -> Bool}
```

External functions are used to represent environment operations for detecting vehicles at entrance or exit in addition to signals from the TC, e.g.,

```
external function carentering: BOOL // ME
external function mainlandgranted: BOOL // MG
```

the value of *carentering* is indicated by the sensor ME, *carexiting* by MX, *mainlandgranted* by MG, and *mainlandrelease* by MY as in Figure 5.2.

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

```
dynamic function mainlandstate: IS_SORT initially red
```

All Boolean outputs of this controller are also described by *dynamic functions*, e.g.

```
dynamic function mainlandred: BOOL initially false  // MRL

dynamic function mainlanduse: BOOL initially undef  // MU

dynamic function tcr : DATA with tcr in Data initially zero // TCR
```

We then describe the behavior of the system using *if-then-else* rules. One example is shown below for the *entering* state.

```
if (mainslandstate = entering ) then

        mainlandgreen := true

        mainlandred := false

        mainlanduse := true

        if(carentering) then mainlandstate := entering

        else mainlandstate := green

        endif

endif
```

5.1.2 Structural Modeling in ASM

We developed structural models (implementations) for both the MLC and ILC. The implementation of the MLC and ILC are shown in Figures 5.5 and 5.6, respectively. We use *static functions* to define primitive gates (AND, OR, NAND, etc.). An example is shown below for an OR gate with two inputs.

```
static function or2 (in0,in1) ==
```

```
if in0 = true  or  in1 = true

then true

else false

endif
```

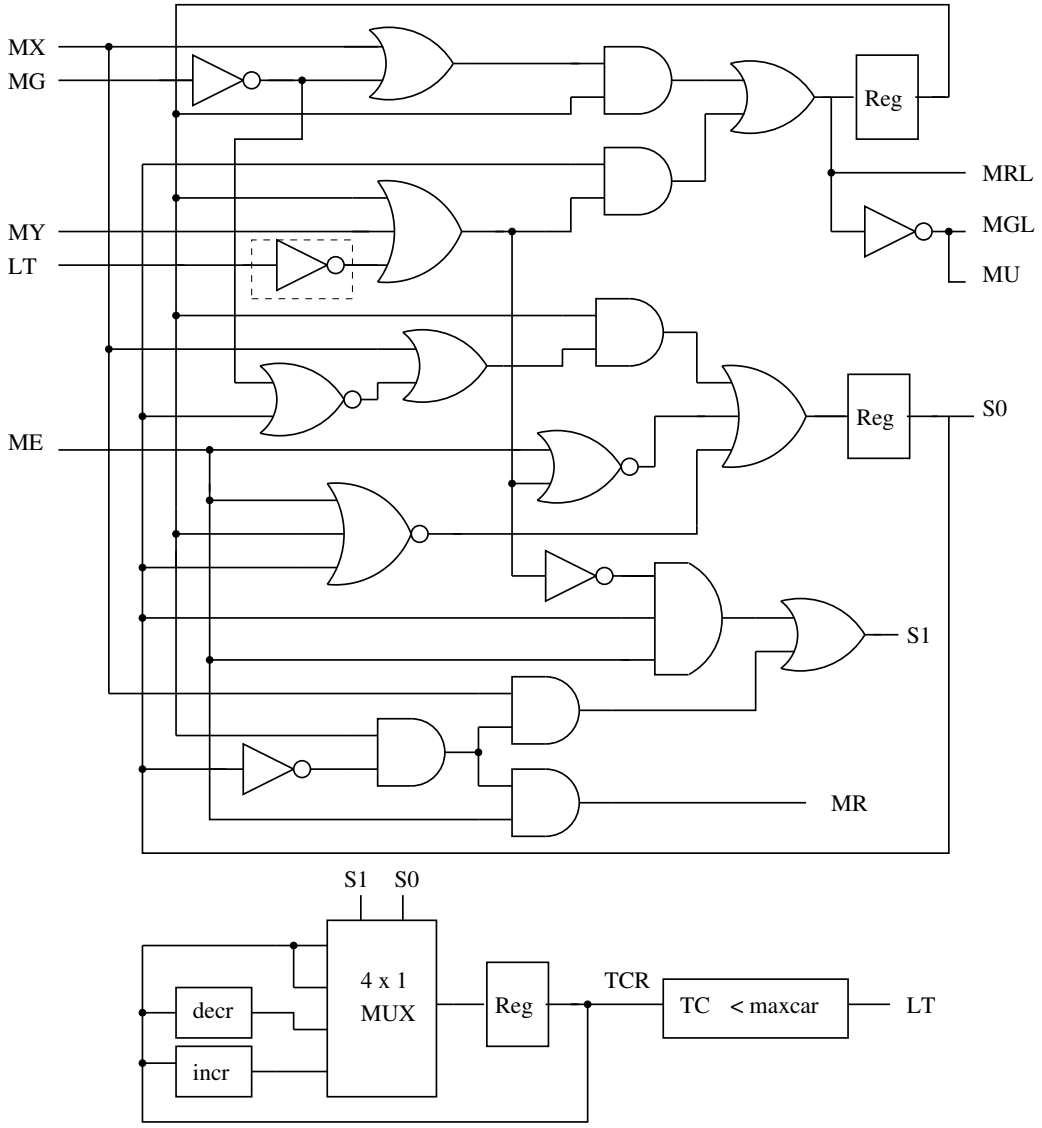

Figure 5.5: MLC implementation

We describe the structure of output signals in addition to internal signals in one transition as following example:

```
d2 := or2(and2(d2,or2(carexiting,inv(mainlandgranted))),
          and2(d1,or3(d2,mainlandrelease,inv(lt(tcr,maxcar)))))
mainlandred := d2
mainlandrequest := and3(d2,inv(d1),carentering)
```

These function were obtained from the specification given above in Figure 5.3, where d1 and d2 represent the state variables of the system which consists of four states. In addition, we use an abstract state variable to model the abstract counter *tcr*. Black-box representation is used to model the *incr* and *decr* functions, while the comparator "$tcr < maxcar$" is modeled with a cross-term, *lt*.

5.2 MDG Verification

Using our ASM-MDG tool, we generated the corresponding MDG-HDL models for both behavioral and structural models including: circuit description, algebraic specifications, and variable order. The abstract and concrete types, uninterpreted functions, and cross-terms are all preserved in the generated models as follows, where *wordn* is a default abstract sort in MDG to denote a word of n bits:

```
gen_const(abstract_,wordn).
gen_const(maxcar_,wordn).
```

```
function(iseq,[wordn,wordn],bool).

conc_sort(mainlandstate_type,[entering_,exiting_,green_,red_]).

function(incr,wordn,wordn).

function(decr,wordn,wordn).
```

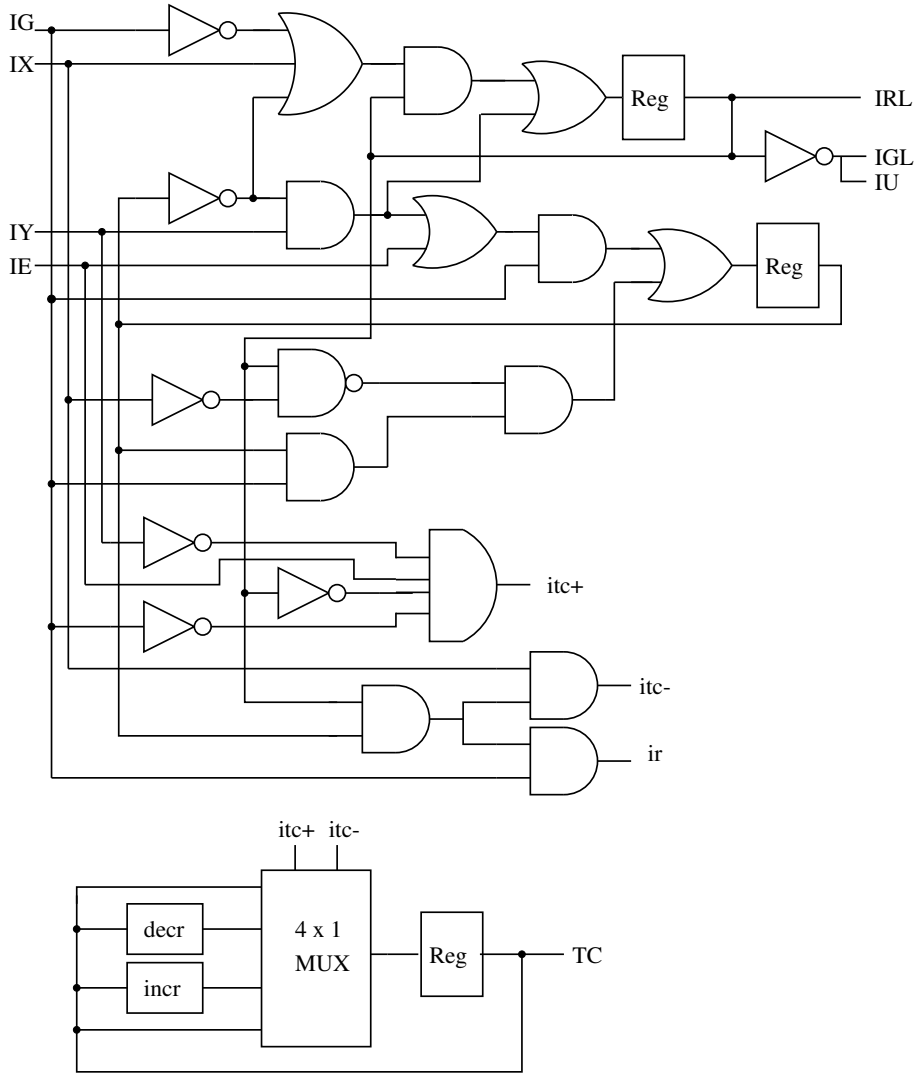

Figure 5.6: ILC implementation

Once the generated MDG-HDL structural and behavioral models were compiled successfully with the MDG tool, we applied both equivalence checking and property checking to verify them.

5.2.1 Equivalence Checking

We succeeded to verify that MLC structural model is equivalent to its behavioral model by applying MDG 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 error injected was removing the inverter marked inside the box in Figure 5.5:

Assumption: $lt(tcr, maxcar\_) = 0$

Initial state: $mainlandstate\_A = red\_, d1\_B = 0, d2\_B = 1$

Clock cycle 1:

Symbolic inputs $carexiting = 0$, $mainlandgranted = 1$

Symbolic state: $mainlandstate\_A = green\_, d1\_B = 1, d2\_B = 0$

Clock cycle 2:

Symbolic inputs $carentering = 0$, $mainlandrelease = 0$

Symbolic state: $mainlandstate\_A = red\_, d1\_B = 0, d2\_B = 0$

Clock cycle 3:

Symbolic inputs $carexiting = 0, carentering = 1$

Symbolic Output: $mainlandrequest\_A = 0, mainlandrequest\_B = 0$

$mainlandgreen\_A = 0, mainlandgreen\_B = 1$

$mainlanduse\_A = 0, mainlanduse\_B = 1$

$mainlandred\_A = 1, mainlandred\_B = 0$

We can see that after the third clock cycle, the MRL output should be active as indicated by the specification, however, the MGL was active after that cycle in the implementation.

We also verified that the ILC implementation is equivalent to its specification, by applying MDG equivalence checking on generated MDG-HDL models. The CPU execution time and resource requirements, including memory usage and MDG nodes generated for both blocks are given in Table 5.1. The experimental results shown below were conducted on a Sun4u machine with Solaris 5.7 OS and 1.0 GB memory.

Table 5.1: MDG equivalence checking results

Model	**Time (Sec)**	**Mem(MB)**	**#MDG Nodes**
MLC Original Model	0.730	1.55	955
MLC Faulty Model	1.040	1.41	1180
ILC Original Model	0.580	0.92	668
ILC Faulty Model	0.580	1.00	763

5.2.2 Model Checking

We have specified a number of properties in $\mathcal{L}_{\mathcal{MDG}}$, and then model checked them on the generated MDG-HDL models. In the following, we describe three properties on the MLC for illustration purposes. The first property states that:

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, where the symbols AG and F mean "`for all paths, for all states`" and "`there exists a state in the future`", respectively:

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

The second property states that:

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

and it is formally specified as following, where & is the logical AND:

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

The last property states that:

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

Property 3: $AG(!((mainlandrequest\_B = 1)\ \&\ (mainlanduse\_B = 1)))$;

All above properties were verified successfully. Verification results for above properties and a similar set on ILC are given in Table 5.2.

Table 5.2: MDG model checking results

Property	Time (Sec)	Mem(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

5.3 Summary

In this chapter we have shown via a case study an application of our ASM-MDG tool. The verification results show that the verification time was very short, and only around 0.1% of the available memory was used. The number of MDG nodes reflects the complexity of the design, it directly affects the time and memory required to complete the verification without state space explosion. This application illustrates the ability of the MDG tool to find bugs in designs and provide counter examples to locate them.

Chapter 6

Conclusions and Future Work

In this thesis, we introduced an interface from the ASM (Abstract State Machines) Workbench to the MDG (Multiway Decision Graphs) tool, called "ASM-MDG". This new interface 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 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 complementary approachs. In the first step, we translate ASM-SL to an intermediate language, ASM-IL, then transform this later to appropriate MDG-HDL code. This approachs works for behavioral or structural

models. The second approach allows a direct mapping from ASM structural components into MDG-HDL netlist of components without unfolding or simplifying the original ASM model. Besides MDG-HDL code, the interface produces a static variable ordering, that satisfies the restrictions given by MDGs, as well as algebraic specification necessary for declaring constants, sorts, functions, etc.

We have applied the ASM-MDG interface on the Island Tunnel Controller as a case study. We used the MDG model checking and equivalence checking on the generated MDG-HDL models. We succeeded in model checking several properties on the Island Tunnel Controller. Through our experiments, we noticed that the direct translation approach is more effincient for structural models since the former one via ASM-IL would generate a large number of components leading quickly to state space explosion.

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

Our approach is built upon existing work that transforms ASM to an intermediate language. While the ASM-IL is tailored to the ASM workbench, we believe that any other ASM tool could reuse our mapping into MDG. As a future work, we think that linking our work to the MDG-HOL hybrid tool developed at Concordia University will further enable theorem proving for ASM models. Also, since both the ASM-WB and MDG tool do not have a very user friendly interface, it is proposed to build a graphical user interface (GUI) for the ASM-MDG tool. This GUI can call the ASM-WB to validate ASM models, then call our ASM-MDG interface to generate MDG-HDL models, and finally call the MDG verification tool to verify the generated models.

Bibliography

[1] R. Achuthan. A Formal Model for Object-Oriented Development of Real-Time Reactive Systems, Ph.D. thesis, Concordia University, Montreal, Canada, October 1996.

[2] 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.

[3] 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.

[4] L. Barakatain. A Practical Model Checking Approach Using FormalCheck. M.A.Sc. Thesis, Concordia University, Department of Electrical and Computer Engineering, Montreal, Canada, August 2000.

[5] 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.

[6] 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.

[7] R.S. Boyer, J S. Moore. A Theorem Prover for a Computational Logic, Technical Report 54, Computational Logic, Inc., 1990, Kaiserslautern, West Germany, July 1990.

[8] 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.

[9] R. K. Brayton et al. VIS: A system for verification and synthesis. In T. Henzinger and R. Alur (eds.), Proceedings Computer Aided Verification, LNCS 1102, SpringerVerlag, 1996, pages 428–432.

[10] Cadence Design Systems, Inc. FormalCheck Users Guide; V2.1, July 1998.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] M. Gordon and T. Melham. Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic, Cambridge, U.K., Cambridge Univ. Press, 1993.

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

[20] S. Hazelhurst and C.-J. H. Seger. A Simple Theorem Prover based on Symbolic Trajectory Evaluation and OBDDs. Technical Report TR–93–41, Computer Science Department, University of British Columbia, 1993.

[21] G.J. Holzmann: The Model Checker SPIN, IEEE Transactions on Software Engineering, Vol. 23, No. 5, May 1997, pp. 279–295.

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

[23] 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.

[24] 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.

[25] 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.

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

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

[28] 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.

[29] D. Muthiayen. Real-time Reactive System Development - A Formal Approach based on UML and PVS. Ph.D Thesis, Concordia University, January 2000.

[30] S. Owre, J.M. Rushby, and N. Shankar. PVS: a Prototype Verification System, In Proceedings of the International Conference on Automated Deduction, Saratoga Springs, NY, USA, 1992, pp. 748–752.

[31] L. Paulson. ML for the Working Programmer, Cambridge University Press, 1997.

[32] C. Plonka. Model checking for the design with abstract state machines. Master's thesis, University of Ulm, Department of Applied Information Processing, 2000.

[33] K. Schneider and D. Hoffmann. A HOL conversion for translating linear time temporal logic to ω-automata, In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery (eds.), Higher Order Logic Theorem Proving and its Applications, LNCS 1690, Springer Verlag, September 1999, pp. 255–272.

[34] 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.

[35] 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.

[36] 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.

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

[38] 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.

[39] 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.

[40] Y. Xu. Model Checking for a First-order Temporal Logic Using Multiway Decision Graphs. PhD Thesis, University of Montreal, Dept. of Information and Operational Research, 1999.

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

[42] 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.

[43] 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.

[44] 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.

[45] 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.