Formal Verification of ASMs using MDGs

A. Gawanmeh [a], S. Tahar [a] and K. Winter [b]

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

[b] *School of ITEE*
University of Queensland
Brisbane, Australia
kirsten@svrc.uq.edu.au

Abstract

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

Key words: Formal Verification, Abstract State Machines, Multiway Decision Graphs, Model Checking

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 [18].

Abstract State Machines (ASM) [15] is a formal specification method for software and hardware systems that has become successful for specifying and verifying complex systems [5]. The formalism is used as a modeling language in a variety of domains as it has been used both in academic and industry contexts [5,16]. Multiway Decision Graphs (MDGs) [8] are decision diagrams based on abstract representation of data and are used for modeling hardware systems in first place. MDGs subsume and extend traditional ROBDDs (Reduced Ordered Binary Decision Diagrams) [6] by abstract data sorts and uninterpreted function symbols.

This paper presents a tool to interface the ASM Workbench [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. Second, MDGs as a data structure for representing transition systems provide a powerful means for abstraction that fit perfectly with those of ASMs. Both notations support the use of abstract types and uninterpreted functions. This allows the user to model and to verify large or potentially infinite models. Finally, providing 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 [12].

For behavioral models, we 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) [30]. The second step provides a transformation from ASM-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. This is proved to be more efficient than translating via ASM-IL, which would provide a very large model. We have applied the ASM-MDG interface to the Fairisle ATM switch [20] as a case study, where we conducted MDG model checking on the generated MDG-HDL models. We succeeded in model checking several properties on the ATM switch controller.

Interfacing ASM and MDG, has been already introduced in [30]. This work is closely related to ours and parts of the results have been re-used here. However, the overall aim of [30] has been to map ASM models directly onto MDG data structures, without utilizing the input notation MDG-HDL. As a consequence, the interface is not providing the user with the facilities of the MDG tool as a black-box verification tool since the tool is only working on MDG-HDL models. The work in [30] also provided an interface from ASM models to the SMV model checker [23] (using the SMV input language). In contrast to SMV, 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. This paper extends our work presented in [12] and [13] by applying the methodology on a real application, which is, the Fairisle ATM switch. The Fairisle ATM switch is a major case study compared to the applications provided in [12] and [13]. It posed several experimental challenges and much space about it is devoted in this paper.

The rest of the paper is organized as follows. Section 2 provides related work to ours. In Section 3, we present the description of ASM and MDG. Section 4 presents our ASM-MDG interface. Section 5 illustrates the efficiency our approach by applying the interface on the case study of the Fairisle ATM switch. Finally, Section 6 concludes the paper with an outlook to future work.

2 Related Work

Related work on the verification of ASM models include the work of Spielmann [28], who investigated the complexity of verifying a class of restricted abstract state machine programs automatically. Also in the work on real-time systems by Beauquier and Slissenko [3], the verification problem is discussed for ASMs. These results are complemented by our work since they remain in theory and neither of the work above provides actual tool support for the verification task.

Gargantini *et al.* [11] presented a framework for automatic translation from ASM to PVS which is a theorem prover based on higher-order logic [25]. They developed a set of PVS theories to define types and functions modeling ASM universes and rules. A set of PVS strategies is provided in order to simplify the proof conduction. In [26], Schellhorn defined a generic proof for the correctness of ASM refinement. The author provided an embedding of ASM into dynamic logic that allows formalizing properties of ASM, then developed a theory for the modularization of correctness proofs for ASM refinements. The results are integrated into the KIV (Karlsruhe Interactive Verifier) system.

Similar to the above two works, we are proposing a framework to allow the verification of ASM models. We differ, however, through the fact that we intend to apply automatic verification techniques, namely model checking and sequential equivalence checking, rather than interactive theorem proving such as in PVS. Furthermore, our approach is founded on the use of the same natural level of abstraction in both ASM and MDG. Finally, our framework is targeted towards the modeling and verification of hardware systems.

In [19], Kort *et al.* describe a hybrid formal hardware verification tool linking MDG and the HOL theorem prover [14] obtaining the advantages of both verification paradigms. They provide an embedding of the MDG input language in HOL, implementing a linkage between HOL and MDG and a series of HOL tactics that automate hierarchical verification. The MDG tool can be called from HOL to perform verification of components that are within its capabilities. Our work provides an external interface to verify ASM designs, while this work is based on using the MDG tool to verify sub-gaols for the HOL theorem prover.

In [4], Börger and Stark provide a history and survey of ASM research including ASM verification, tool integration and linking ASM to verification tools. From a more general perspective, the work described by Shankar [27] and Katz and Grumberg [17] are also related in that they provide a tool framework comprising a general intermediate language which allows one to interface a high-level modeling language with a variety of tools.

3 Preliminaries

3.1 Abstract State Machines

Abstract State Machines (ASM) [15,16] is a specification method for software and hardware modeling. The system is modeled by means of states and transition rules. The latter specify the behavior of the system in terms of state changes which might be guarded. The notation of ASM is efficient for modeling a wide range of systems and algorithms as the number of case studies demonstrates [16].

States are many-sorted first-order structures. A structure is given with respect to a signature which is a finite collection of function names, each of a fixed arity. The given structure fixes the syntax by naming sorts and functions. An algebra provides *domains* (i.e., carrier sets) for the sorts and a suitable symbol interpretation for the function symbols on these domains, which assigns a meaning to the signature. Therefore, a state is defined as an algebra of a given signature with *domains* and an interpretation for each function symbol.

ASM provides *static functions*, *dynamic functions* and *external functions. Sta-*

tic functions have a fixed interpretation in each computation state and therefore, static functions never change their evaluation during a run. They represent constants or primitive operations of the system, such as combinational logic blocks (in hardware specifications). *Dynamic functions* change their interpretation during a run as a result of the specified system's behavior. Their evaluation can be changed through the transitions occurring in a computation step. They represent the internal state of the system. The interpretation of *external functions* is determined in each state by the environment. Changes in external functions which take place during a run are not controlled by the system.

Variables and *terms* are used over the signature as objects of the structure. A state transition into the next state occurs when dynamic functions change their evaluation. *Locations* and *updates* capture this notion.

A *location* of a state is a pair $loc = (f, \overline{a})$, where f is a dynamic function symbol and $\overline{a}$ is a tuple of elements in the domain of the function. The element $f(\overline{a})$ at a state is the *value* of the location $(f, \overline{a})$ in that state.

For changing values of locations the notion of an *update* is used. An *update* of a state is a pair $\alpha = (loc, val)$ where $loc = (f, \overline{a})$ is a location and val, the update value, is a value in the function domain. To fire an update at a state, the update value is set to the new value of the location. As a consequence, the overall dynamic function f is redefined to map the location onto the new value.

Transition rules define the state transitions of an ASM. While terms denote values, transition rules denote *update sets*, which define the dynamic behavior of an ASM. At each state all update sets are fired simultaneously which causes a state change. All locations that are not referred to in the update sets remain unchanged. The ASM-SL is a specification language developed for modeling in ASM [9].

3.2 Multiway Decision Graphs

Multiway Decision Graphs (MDGs) [8] have been proposed as a solution to the state space explosion problem of verification tools based on ROBDD (Reduced Order Binary Decision Diagrams) [6]. MDGs subsume ROBDDs, while accommodating abstract sorts and uninterpreted function symbols. This significantly enhances the capability to verify a broader range of systems than classical ROBDD-based tools.

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 an *enumeration* while abstract sorts do not. The enumeration of a concrete sort is a set of distinct constants of that sort. If a function is of a concrete sort, while at least one of its domain variables is abstract, then the function is referred to as a *cross-operator*. The constants occurring in the enumeration are referred to as *individual constants*, and other constants as *generic constants*. Concrete function symbols must have an explicit definition; they can be eliminated and do not appear in the MDG. Abstract function symbols and cross-operators are *uninterpreted.*

Logic gates can be represented by MDGs similarly to ROBDDs, because all inputs and outputs are of Boolean type. Design descriptions at the RTL involve the use of more complex functions and data structures. For system descriptions the MDG tool comes with a hardware description language called MDG-HDL [32]. 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. A behavioural description (specification) ia an MDG table, whivh 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.

Based on MDGs, a tool set for the formal verification of finite state systems (machines) has been developed. It includes application procedures for combinational and sequential equivalence checking [8], invariant checking [8] and model checking [31]. The MDG tool has been used to verify a number of non-trivial systems such as communication switches and protocols [2,7,29,33–35]. 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 model. 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.

4 ASM-MDG Interface

To interface ASM and MDG we can benefit from the fact that both formalisms have similar features. Especially when modelling hardware systems, the sim-

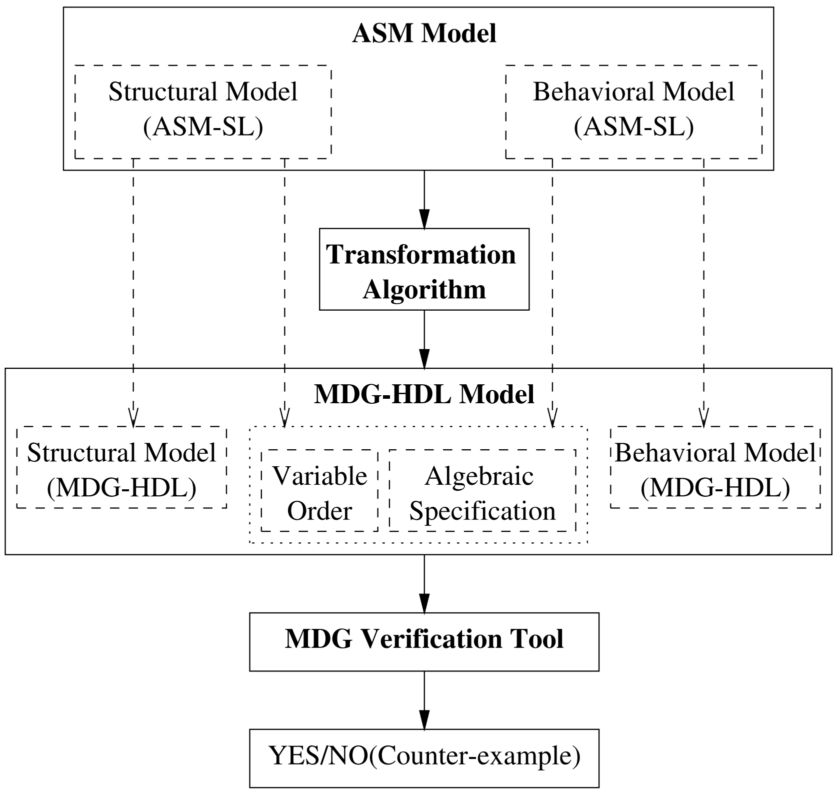


Fig. 1. Proposed ASM-MDG framework

ilarities in the way of modelling become apparent. Both formalism provide a powerful means to model data issues using *abstract* types 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 [32]. This interface will ultimately allow the formal verification of ASM models using the MDG tool. Figure 1 shows an overview of the expected ASM-MDG verification framework.

It 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. The two ASM models, one describing the behavior in terms of transition rules, the other describing the structure of the design in terms of static functions, are separately transformed into the corresponding MDG-HDL models.

4.1 ASM-MDG Interface using ASM-IL

In order to provide a generic interface for the ASM-WB with different tools, ASM models are automatically translated into the intermediate language ASM-IL as proposed in [30]. Based on ASM-IL, we propose to built an interface to the MDG tool. To transform an ASM model into an ASM-IL model, all nested transition rules of the original ASM model are flattened and complex data structures and functions are unfolded. Thus, ASM-IL provides an interface language for representing state transitions in a very general way. It can be readily transformed into the different input languages when interfacing various tools. This generality, however, comes at the price of loosing structural information of the original ASM model.

Starting from the ASM-IL language, we built our interface to the MDG tool as shown in Figure 2.

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. 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 the form (*loc, [guard, val]*). This set specifies for each location a set of guarded updates. The new value of a location in the next state will be the one for which the corresponding guard is satisfied in the current state. If none of the given guards evaluates to true in a state, the value of the location remains unchanged in the next state.

To transform ASM transition rules into an ASM-IL representation as above, 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.

4.1.1 Transforming ASM-IL to MDG Behavioral Models

To treat behavioral ASM-SL specifications, ASM models are first translated into the ASM-IL as shown in Figure 3. The model is first parsed for syntax check, ASM universes, functions, and transition rules are collected. Then an analyzer generates the ASM-IL representation. The behavior of the model is described as a set of guards and updates for each state variable (*update*

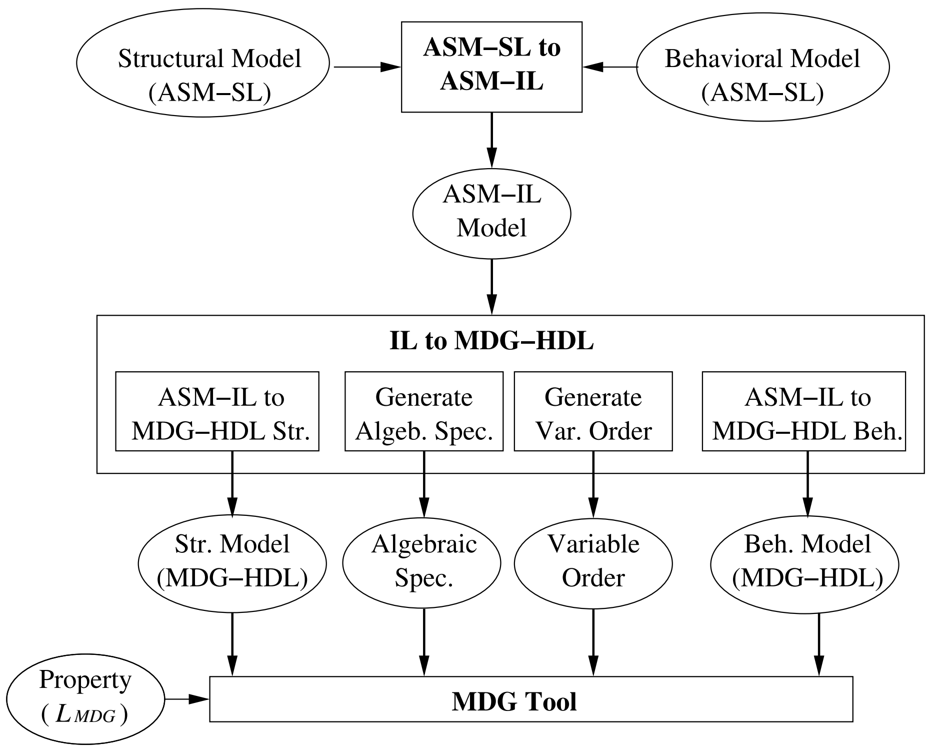
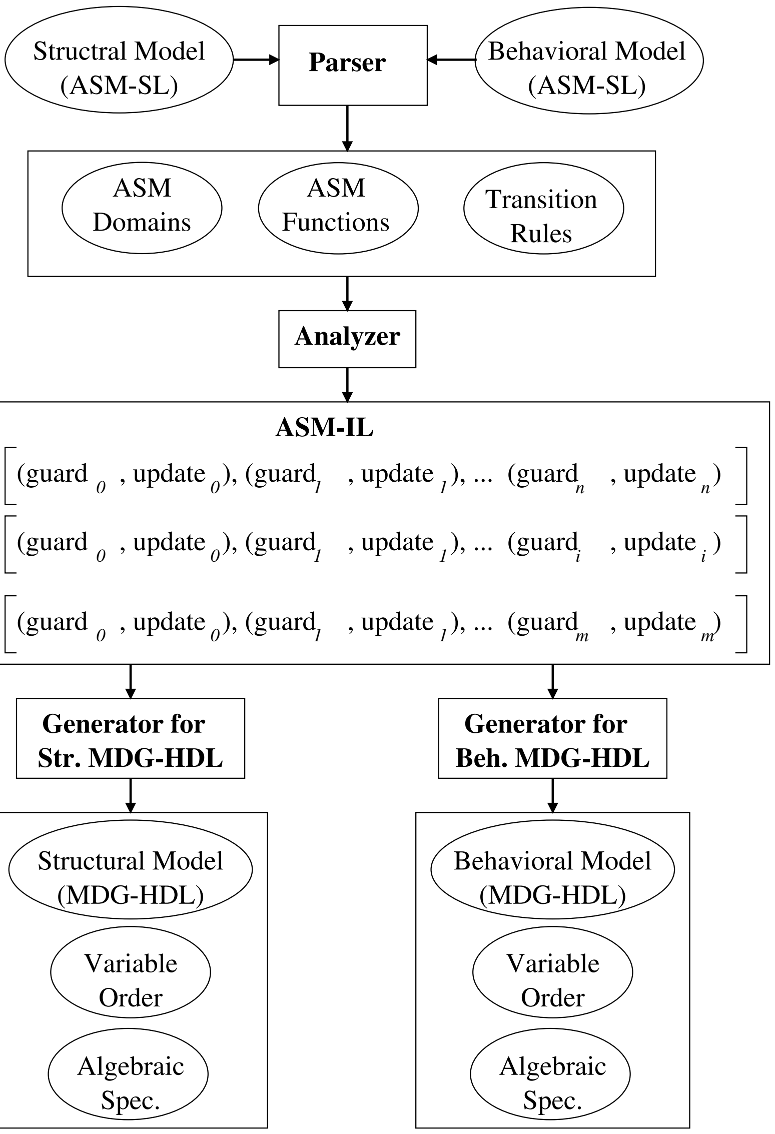


Fig. 2. ASM-MDG interface via ASM-IL

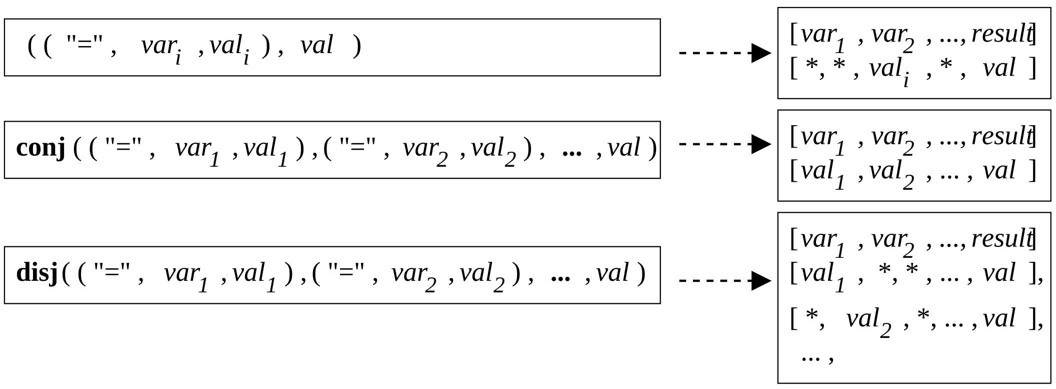
location), the next state value for each location is the corresponding value to the first satisfied guard in the list. Otherwise, if there is no guard satisfied, the location keeps its current state value in the next state. From this ASM-IL model, MDG-HDL behavioral descriptions are generated in terms of MDG tables. In addition, variable order and algebraic specifications are produced.

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* becomes the entry of 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.



Structral Model (ASM-SL)
Parser
Behavioral Model (ASM-SL)
ASM Domains
ASM Functions
Transition Rules
Analyzer
ASM-IL
loc_0 [(guard$_0$, update$_0$), (guard$_1$, update$_1$), ... (guard$_n$, update$_n$)]
loc_1 [(guard$_0$, update$_0$), (guard$_1$, update$_1$), ... (guard$_i$, update$_i$)]
...
loc_k [(guard$_0$, update$_0$), (guard$_1$, update$_1$), ... (guard$_m$, update$_m$)]
Generator for Str. MDG-HDL
Generator for Beh. MDG-HDL
Structural Model (MDG-HDL)
Variable Order
Algebraic Spec.
Behavioral Model (MDG-HDL)
Variable Order
Algebraic Spec.

Fig. 3. ASM-MDG transformation using ASM-IL



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

Fig. 4. Generating MDG tables from guarded updates

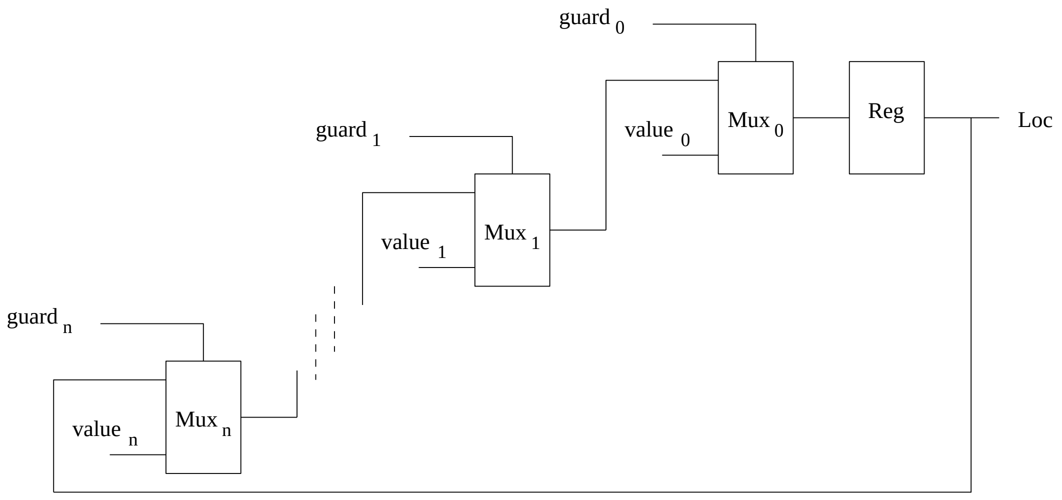


Fig. 5. Mapping ASM-IL expressions for one location into MDG-HDL

4.1.2 Transforming ASM-IL to MDG Structural Models

To transform an ASM-IL representation into MDG-HDL structural code, we map locations, guards and values into registers and signals. First, for each location we create a state component in MDG-HDL, represented as register. Each location name is mapped into a signal that is connected to the register's output. The resulting value of the location is mapped to a signal that is connected to the register's input. Transforming a whole model results in a state machine (sequential circuit) in which the number of state variables is equal to the number of updated locations in the model.

Second, guards and values are transformed into 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 multiplexors together in a hierarchical way as shown in Figure 5. The output of the cascade is connected to 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 and the value of the location remains unchanged.

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 recursively to each guard expression - creating the corresponding MDG-HDL components 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 MDG *transform* components that can be viewed as uninterpreted. 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. A table indicates that the output signal of the table equals to *true* (1) when *var* equals to *val* and *false* (0) otherwise.

4.2 ASM-MDG Syntactic Transformation for Structural Models

When an ASM model is translated into the ASM-IL rules, all structured functions are flattened into the primitive ones. The location-update pairs are used to build the MDG-HDL structural model, which is a set of components interconnected by internal signals. The resulting MDG-HDL structural model becomes very large as only the predefined basic MDG-HDL components are used. Moreover, a large number of components results in a large number of variables which makes it very hard to generate a good variable order. As a consequence, the transformation as introduced in Section 4.1.2 provides a potential bottleneck in our approach.

To solve this problem, we provide for structural designs a direct interface between ASM-SL and MDG-HDL without using the intermediate representation of ASM-IL. In order to keep this interface simple and feasible, we implement it for a set of predefined ASM functions without going into their semantics. In other words, we define ASM *static functions* that correspond to MDG-HDL primitive components. We use these to built our ASM structural model, which then can be readily translated into MDG-HDL structural model.

Figure 6 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 generated 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

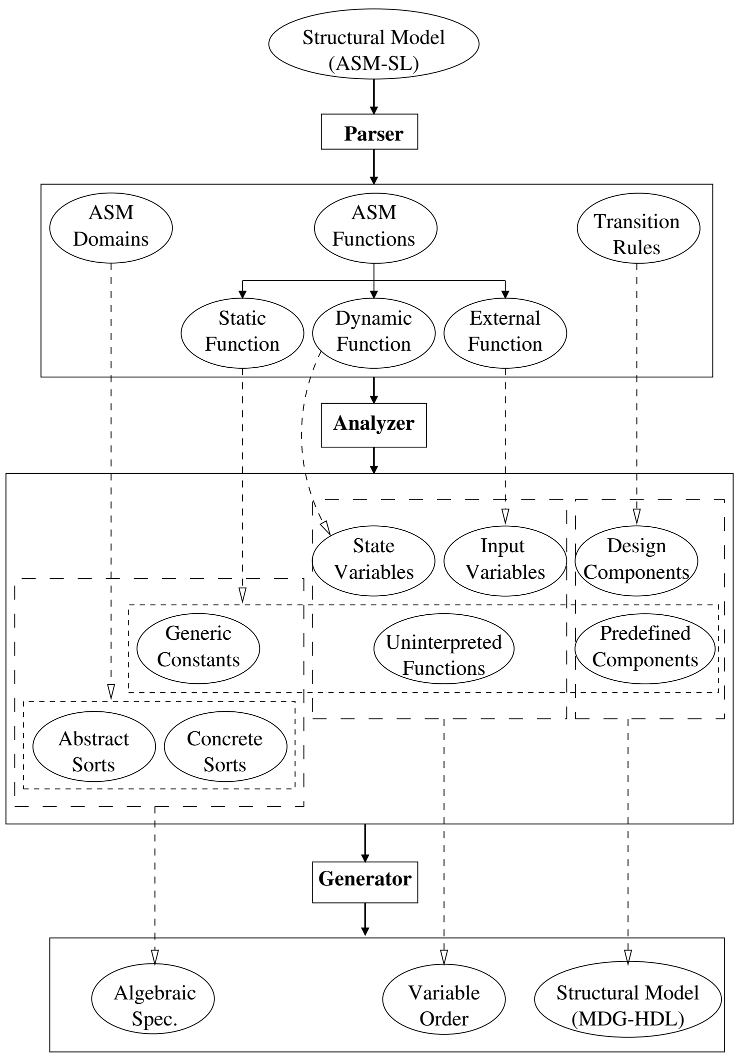


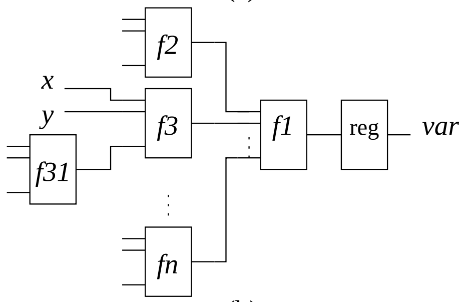
Fig. 6. ASM-MDG syntactic transformation for structural models

and functions in the design such that the order obeys the restrictions imposed by the MDG tool [32]. It includes all variables and internal signals used in the model.

The generated MDG-HDL structural model is a circuit description given as a netlist of components interconnected with signals. Besides uninterpreted functions and cross-operators, the current implementation of the tool supports the set of ASM functions that can be mapped directly to MDG-HDL library components [32]. Figure 7 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*, ..., *fn* can be any of the MDG-HDL library functions, an uninterpreted function or a cross operator, *var* is the state variable, *Sij* are internal signals, and finally x and y are ASM variables. All functions are declared as $function((inputs), output)$. This structure is recursively treated until a predefined function is found, which is syntactically

```
var := f1( f2(...) , f3( f31(...), x,y), ..., fn(...) )
```

(a)



(b)

```
component( reg(S1, var ))
component( f1((S11, S12, ..., S1n), S1))
component( f2((S21, S22, ..., S2j),S11))
component( f3((S31, x, y), S2j))
component( f31((S311, S312, ..., S31m),S31))
component( fn((Sn1, Sn2, ..., Sni), S1n))
```

(c)

Fig. 7. Mapping structural ASM-SL into MDG-HDL components

mapped into the corresponding MDH-HDL library components.

4.3 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.

4.4 Variable Order

MDGs have some restrictions on the order of abstract variables and cross-operators [32]. In order to obey these restrictions, we explore all functions and cross-operators in the ASM-IL expressions and order the variables according

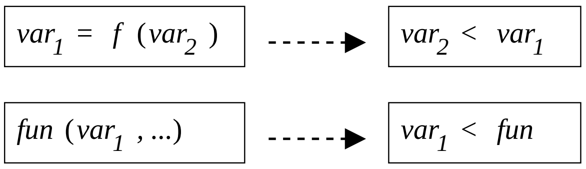


Fig. 8. Variable order constraints

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 *var2* 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 8 depicts these dependencies.

For the direct syntactic mapping of structural models, we build the variable order in the same way as the design structure is constructed. Since dependent variables or signals come last while building the components, we just put them on bottom of the variable order (e.g., signal *S1* comes below *S12* in Figure 7).

We illustrate the transformation on a case study of an Island Tunnel controller [10], where we provide ASM models for the specification and implementation of the controller. Using our ASM-MDG tool, we generated the corresponding MDG-HDL models for both behavioral and structural models for each block, including: circuit description, algebraic specifications, and variable order[1] [13].

5 Case Study: Fairisle ATM Port Controller

In this section, we present our results of formally verifying an ATM (Asynchronous Transfer Mode) switch [20] using the ASM-MDG tool proposed in this paper. By this example, we show how to use model checking to verify a design modeled in ASM. The device we investigated is a part of a network which carries real user data: the Fairisle ATM network, designed and in use at the Computer Laboratory of the University of Cambridge. The switch consists of a Fairisle 4 by 4 switch fabric and four Fairisle ATM port controllers. It performs the actual switching of data cells and forms the heart of the ATM Fairisle communication network. Figure 9 shows the Fairisle ATM switch ports.

The Fairisle ATM switch consists mainly of a port controller and a switching fabric. The port controller does only VCI (Virtual Channel Identifier) mapping

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

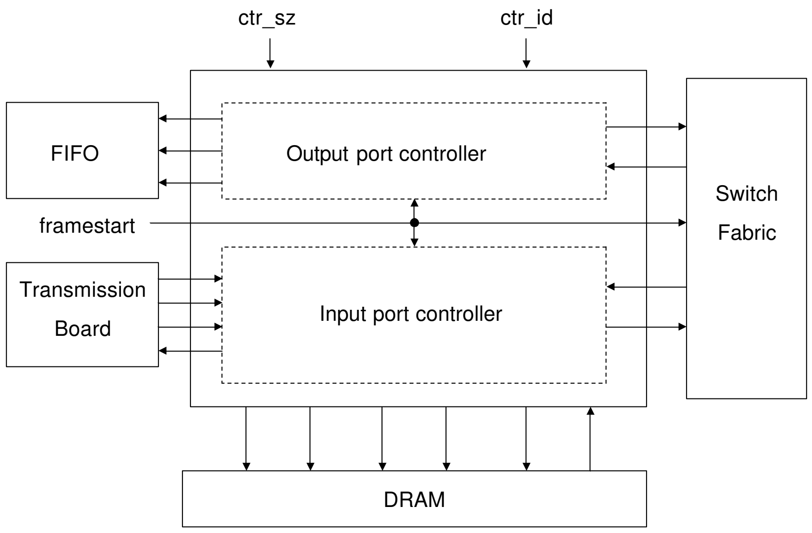


Fig. 9. The Failisle ATM switch.

and FIFO (First In First Out) queuing. In the original design, a Xilinx chip controls all its functions, and it uses triple ported DRAMs to look up the new VCI. It also uses a FIFO to do speed matching with the transmission board. As shown in Figure 9 , the port controller is connected to the Fairisle ATM switch fabric, transmits ATM cells to the fabric and receives acknowledgment signals from it. Both the port controller and the switch fabric use the same *framestart* signal to synchronize the overall behavior [22].

The port controller consists of an input port controller and an output port controller. It is able to transmit one cell every 128 clock cycles. With a clock frequency of 20 MHz, the maximum bit rate is 80 Mbps. There are no service classification, no scheduling or traffic shaping, no monitoring and policing in this port controller, but we can give a priority to an ATM cell, and this is done by preloading the priority bit into the memory. The priority bit will be used for arbitration in the switch fabric.

Figure 10 shows the format of an ATM cell. Received cells have 52 bytes: 48 data bytes, 2 VCI bytes and 2 FAS (Frame Assignment Sequence) bytes. Transmitted cells have 54 bytes: 48 data bytes, 1 Fabric Routing Byte (FRB), 1 Port controller Routing Byte (PRB), 2 VCI bytes and 2 FAS bytes. Since each cell consumes 64 bytes memory, the memory, which is 256k x 8 bit, can contain 4096 ATM cells. This means that the port controller supports 4096 connections. To prevent two cells with the same VCI arriving at the memory consecutively, only one cell is allowed in the memory [22].

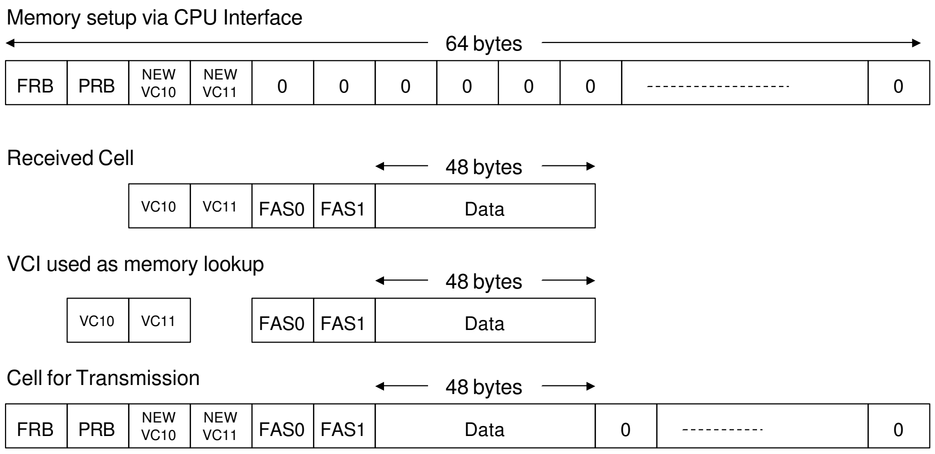


Fig. 10. Format of received and transmitted cells

5.1 *Behavior of the Fairisle Port Controller*

The Fairisle port controller consists of an input port controller and an output port controller. The input port controller receives ATM cells from the transmission board, and writes them into the memory at an address based on the value of the VCI. In addition, the input port controller reads ATM cells out of the memory and transmits them into the switch fabric. If it receives a positive acknowledgement signal, the input port controller will continue transmitting data; otherwise, it will stop sending data. The output port controller receives data cells from the fabric, and sends acknowledgment signals back to the fabric. If the output port controller receives a data cell, it gives a positive acknowledgment signal; otherwise, it sends a negative acknowledgment.

The state transition of the input port controller with 8 states (*ip_idle, rx_wait, rx_store1, rx_store2, rx_data, tx_addr, tx_first_5 and tx_data*) is shown in Figure 11, where conditions numbered from 1 to 15 are guards for the transition from one state to another. Basically, *rx_idle* is the idle state; *rx_wait* means the state of waiting for the start of the cell signal (*rx_ip_soc*) to be asserted; *rx_store1* and *rx_store2* indicate the states that the input port controller stores the first and second VCI byte, respectively; *rx_data* is the state of data transfer from the transmission board to the input port controller; *tx_addr* is the state of setting the memory address; *tx_first_5* means the state of transmitting the first 5 bytes of data to the fabric; *tx_data* indicates the state of transmitting the remaining data into the fabric [22].

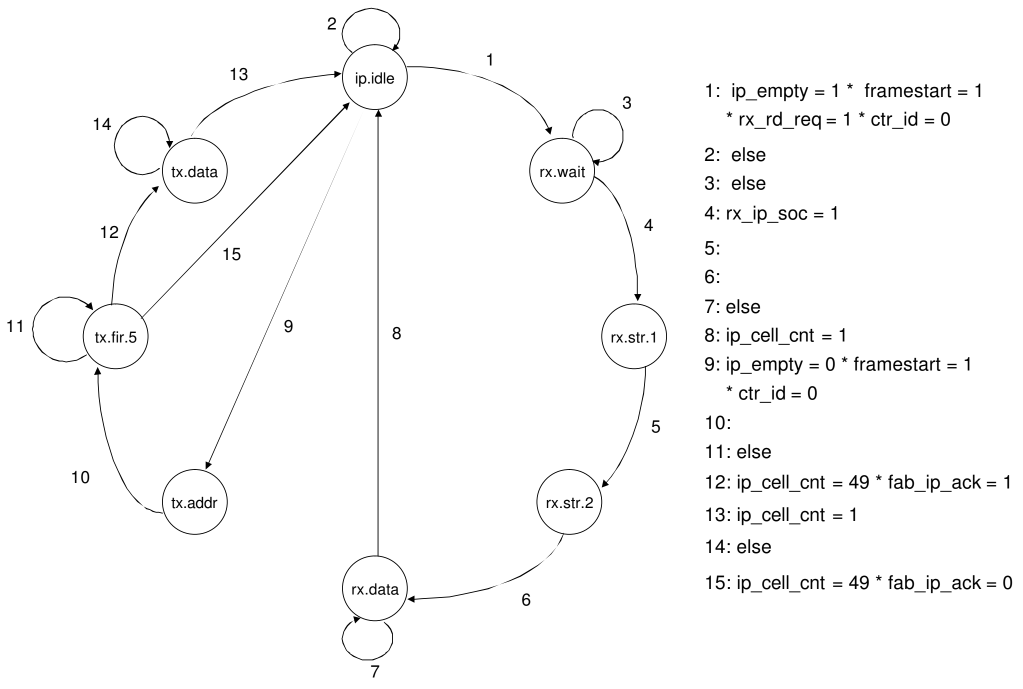

ip.idle
rx.wait
rx.str.1
rx.str.2
rx.data
tx.addr
tx.fir.5
tx.data
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
1: ip_empty = 1 * framestart = 1 * rx_rd_req = 1 * ctr_id = 0
2: else
3: else
4: rx_ip_soc = 1
5:
6:
7: else
8: ip_cell_cnt = 1
9: ip_empty = 0 * framestart = 1 * ctr_id = 0
10:
11: else
12: ip_cell_cnt = 49 * fab_ip_ack = 1
13: ip_cell_cnt = 1
14: else
15: ip_cell_cnt = 49 * fab_ip_ack = 0

Fig. 11. State transition diagram of the ATM switch controller.

5.2 Structure of the Fairisle Port Controller

Figure 12 shows the structure of the port controller. It consists of an input port controller and an output port controller. The input port controller processes the signals from the transmission board, the memory and the fabric. The output port controller interfaces with the signals from the fabric and the output FIFO.

The input port controller consists of an *ip controller*, an *ip cell counter* and an *address accumulator*. The *ip controller*, which coordinates the *ip cell counter* and the *address accumulator*, controls the data reception, transmission, and memory read and write. The *ip cell count* and *address accumulator* are up counters that increment by 1 per data byte transfer. In Figure 12, the signals *ip_mem_data, ip_mem_wr_en, ip_mem_addr_r, ip_mem_addr_c, ip_mem_rd_req* and *mem_ip_data* are the interface signals between the input port controller and the cell memory. The signals *ip_mem_data* and *mem_ip_data* mean the data outputs to the cell memory and the data inputs from the cell memory, respectively. Both signals have an 8-bit bus width. The signals *ip_mem_wr_en* and *ip_mem_rd_req* are the memory write enable and memory read request signals, respectively. The memory row and column addresses are provided by *ip_mem_addr_r* and *ip_mem_addr_c*, respectively. The *rx_ip_data* is an 8-bit data bus which is the data input from the transmission board. The signals

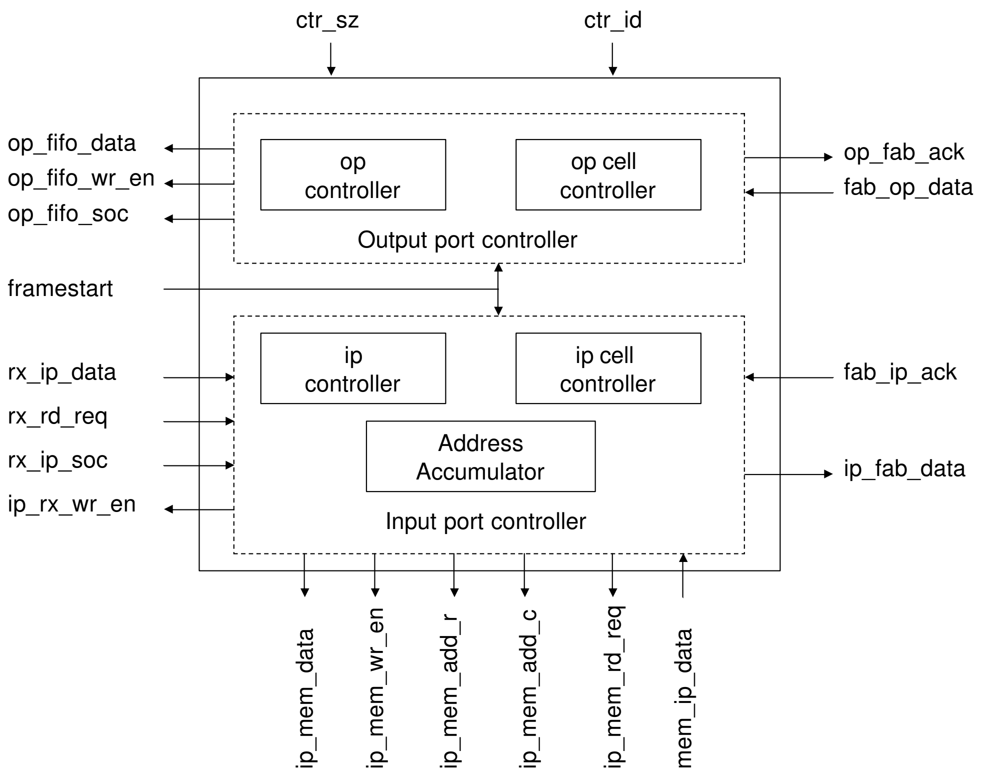


Fig. 12. Structure of the port controller.

rx_rd_req and *rx_ip_soc* indicate cell availability in the transmission board and the start of a cell, respectively. The *rx_ip_soc* signal corresponds to the framestart mentioned above. The signal *ip_rx_wr_en* demonstrates whether the input port controller is able to accept a cell or not. The *ip_fab_data* is an 8-bit data bus which transfers data from the input port controller to the fabric. The *fab_ip_ack* is the acknowledgment signal which indicates whether the current cell succeeded the transfer to the destined switch fabric.

The output port controller consists of an *op controller* and an *op cell counter*. The *op controller* generates the acknowledgment and SOC signals, and controls the *op cell counter*. The *op cell counter*, which is very similar to the ip cell counter, increments by one per data transfer. In Figure 12, *op_fab_ack* and *fab_op_data* are the signals in the interface between the output port controller and the fabric. *fab_op_data* is an 8-bit data bus from the fabric to the output port controller. *fab_op_ack* is acknowledgment signal generated by the output port controller. In addition, there are *op_fifo_data, op_fifo_wr_en* and *op_fifo_soc* signals between the output port controller and the output FIFO. The *op_fifo_data* is an 8-bit data path from the output port controller to the FIFO. The *op_fifo_wr_en* is the write enable signal for the output FIFO. The signal *op_fifo_soc* indicates the start of a cell, and it is asserted before the first byte data transfer. The switch controller has, in addition, an external reset signal (*npc_rst_n*).

There are two control signals (*ctr_id* and *ctr_sz*) and one signal (ip_empty) inside the port controller. *ctr_id* is an input disable register. When *ctr_id* asserts, all the inputs are disable. During the period of *ctr_id* =1, the microprocessor could pre-load the new VCIs, FRB and PRB into the memory. The register *ctr_sz* is for debugging purpose. When *ctr_sz* is high, the memory address of the incoming cell is not based on the old VCI values, instead, the row address of the incoming cell is 0 and the column address is from 0 to 63. The control signal *ip_empty* is used to indicate the status of the port controller. When it is asserted, the input port controller can accept a cell from the transmission board; otherwise, a cell can be transmitted into the fabric from the input port controller [22].

5.3 Modeling in ASM

We first modeled the behavioral state machine of the port controller in ASM. The bytes counter can be taken in ASM as a parameter of an abstract type that represents any natural number. For illustration purposes. Figure 13 shows parts the ASM code for the state machine which describes the behavior of the port controller.

We first define a concrete domain *CS_SORT*, that represents the state of the controller, with all the enumerated state values. Then we define an *abstract* data type and an abstract function *ip_cell_cnt* that represents the bytes counter. We use the dynamic function *controllerState* of sort *CS_SORT* to represent the state of the controller. Then we define the inputs of the controller as external functions. At this point, we can describe the behavior of the controller using transitions, where each transition is guarded with a set of conditions as shown above.

Next, we modeled in ASM the structural model of the Fairisle port controller. This model represents an implementation model, with abstraction applied on the data transmission, since we consider transmitting one byte similar to transmitting the rest of data as discussed above. For illustration purposes, Figure 14 shows parts of the ASM code for the implementation of the state transition system of the controller.

In the ATM switch, 50 bytes data are transferred from transmission board to the input port controller. Because a byte of data transfer has the same behavior as the data transfer of other 49 bytes, we could reduce the number of data transfers in the verification. In the port controller, the number of data transfer is controlled by counters, so we abstract the scale of the counter to

```
freetype  CS_SORT == {ip_idle, rx_wait, rx_store_1, rx_store_2,
                rx_data, tx_addr, tx_first_5, tx_data, ip_idle}
static function Data == { abstract }
static function one == abstract
static function zero == abstract

dynamic function ip_cell_cnt: DATA with
      ip_cell_cnt in Data initially zero

dynamic function controllerState: CS_SORT initially ip_idle
external function ip_empty: BOOL
external function framestart: BOOL
external function rx_rd_req: BOOL

transition T2 == block
    if (controllerState = rx_wait) then
        if(irx_ip_soc = true) then
            controllerState = rx_store_1
        else
            controllerState = rx_wait
        endif
    endif
endblock transition T3 == block
    if (controllerState = rx_store_1) then
        controllerState = rx_store_2
    endif
endblock transition T4 == block
    if (controllerState = rx_store_2) then
        controllerState = rx_data
    endif
endblock
```

Fig. 13. ASM code for the port controller behavioral state machine

simplify our verification. In the port controller, the acknowledge signal should be available 5 clock cycles after the input port controller sends the first byte of data to the fabric, so we could apply 12 bytes data in a cell which includes 2 bytes VCIs, 2 bytes FAS and 8 bytes data. Accordingly, the counter size should be reduced by 40 (52-12). Then we have to change our environment machine from 64 state to 15 states (10 states for data transfers and 5 states for handshaking).

We also implemented the counter as a state transition system in ASM as shown in Figure 15 below. Note in here that we use the abstract values *one, zero*, and

```
transition IP_MAIN == block
    if(clk = true or npc_rst_n = true) then
        if (npc_rst_n = false)
            Dx := false
            Dy := false
            Dz := false
        else
        block
        if(x = false and y = false and z = false) then
            if (ip_empty = true and framestart = true and
                    rx_rd_req = true and ctr_id = false) then
                Dx := false
                Dy := false
                Dz := true
             else if (ip_empty = false and framestart = true and
                    ctr_id = false) then
                Dx := true
                Dy := true
                Dz := true
             else
                Dx := false
                Dy := false
                Dz := false
             endif
             endif
        elseif(x = false and y = false and z = true) then
            if (rx_ip_soc = true) then
                Dx := false
                Dy := true
                Dz := false
            endif
        elseif(x = false and y = true and z = false) then
                Dx := false
                Dy := true
                Dz := true
        endif
```

Fig. 14. ASM code for the main state implementation

max to represent the abstract counter. Similarly, the uninterpreted function *decr* is used to represent the operation of decrementing the counter, and is defined in ASM as $static\ function\ decr\ ==\ MAP\_TO\_FUN\{abstract \rightarrow abstract\}$.

```
transition IP_CNTR ==
block
    if(clk = true or npc_rst_n = true) then
       if (npc_rst_n = false)
                ip_cell_cnt = zero;
        elseif (x = false and y = true and z = true) then
                // ip_state_i == 'rx_store2)
                ip_cell_cnt = max;
        elseif (x = true and y = false and z  = true) then
                //(ip_state_i == 'tx_addr)
                ip_cell_cnt = max;
        elseif (ip_cell_cnt = one or ip_cell_cnt = zero)
                ip_cell_cnt = zero;
        else
            decr(ip_cell_cnt);
        endif
        endif
        endif
        endif
    endif
endblock
```

Fig. 15. ASM code for the abstract counter in the port controller

5.4 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 model checking on the generated models. A set of properties were specified in $\mathcal{L}_{\mathcal{MDG}}$ for this purpose. The Fairisle port controller appends the new VCIs, FRB and PRB onto ATM cells and transfers them into the fabric, so its major properties could include registers reset, memory addressing, cell counting, data and acknowledgment transfer. Accordingly, we defined the following six major properties, including safety and liveness properties. Then we described them formally in $\mathcal{L}_{\mathcal{MDG}}$, where the symbols AG means "*for all paths, in all states*", F means "*eventually*

[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/ATM/

/ *in the future*", "&" is the logical AND, and "!" is the logical NOT.

Property 1: The port controller will be reset properly when either the reset signal (*npc_rst_n*) is zero or the port controller input disable signal (*ctr_id*) is asserted.

```
AG(((npc_rst_n = 0) or (ctl_id = 1))
       => (x = 0 and y = 0 and z = 0))
```

Property 2: When the input port controller can accept a cell, the transmission board has a cell to send, and the input port controller is in debugging state ($ctr\_sz = 1$), then the cell will be transferred to the input port controller and stored in the memory at the right location.

```
AG(((x = 1) and  (y = 0) and (z = 1)  and (rx_rd_req = 1) and
(ctr_sz = 1)) => (ip_mem_data = rx_ip_data))
```

Property 3: When the input port controller can accept a cell, the transmission board has a cell to send, and the input port controller is in the normal operation state ($ctr\_sz = 0$), then the cell will be transferred to the input port controller and stored in the memory at the right location.

```
AG(((x = 1) and  (y = 0) and (z = 1)  and (rx_rd_req = 1) and
(ctr_sz = 0)) => (ip_mem_data = rx_ip_data))
```

Property 4: While the input port controller is receiving data, if the cell counter is equal to "1", it will go to the initial state. In order to model this property, we have to use the cross-operator $iseq(ip\_cell\_cnt, one)$ which only returns true if the counter has the abstract value *one*.

```
AG((iseq(ip_cell_cnt, one_)) and (x = 1 and y = 0 and z = 0) =>
(x = 0 and y = 0 and z = 0 ))
```

Property 5: The memory cannot be read and written at the same time.

```
AG(!((ip_mem_rd_req = 1) and (ip_mem_wr_en = 1)))
```

Property 6: The output port controller will send an acknowledgment signal after it detects an incoming cell.

```
AG((fab_op_data_0 = 1) => (op_fab_ack = 1) )
```

All properties were verified successfully. The verification results for the set of properties on the Fairisle ATM switch controller are given in Table 1. The table indicates the CPU time in seconds, the memory usage in MB, and the number of generated MDG nodes. We clearly notice that the CPU time, memory usage and MDG nodes heavily depend on the property under verification. Thanks to the abstraction, we have a smaller number of MDG nodes than the number

Table 1
MDG model checking results

Property	CPU Time (Sec)	Memory (MB)	MDG Nodes
Property 1	35	8	12644
Property 2	47	11	15678
Property 3	132	17	22462
Property 4	82	13	18551
Property 5	22	7	9832
Property 6	10	4	7225

Table 2
VIS model checking results

Property	CPU Time (Sec)	Memory (MB)	BDD Nodes
Property 1	52	92	12644
Property 2	198	198	284563
Property 3	109	156	293354
Property 4	378	201	304731
Property 5	34	77	153980
Property 6	76	89	197091

of BDD nodes generated by the VIS tool for the same properties, in addition to less memory usage and less CPU time (see Table 2).

This ATM switch has been verified previously using different approaches. Tahar *et al.* [29] used the MDG tool to model the switch in MDG-HDL and verified several properties for this switch. In another work, Lu *et al.* [21,22] used the VIS model checker in order to verify a Verilog implementation of the switch, and they succeeded in verifying several properties. Table 2 shows the verification results they achieved. The fourth property in our experiments is different from the one shown in [21] in order to illustrate the use of cross-operators and abstract data types in the modeling of the property. We also model the switch on a higher level of abstraction and used an automatic translation in order to generate an MDG model that could be verified in the MDG tool. In addition, we achieved better performance in terms of CPU time, memory consumed, and complexity of the graph built by the verification tool, this is because the VIS tool is based on BDDs and therefore has no support for abstract data types. The abstraction we applied on the model, which is supported by ASM modeling and MDG verification, cannot be supported in other verification tools such as VIS.

In our previous work [13], we 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 where we verified several properties on the design, and also verified that the implementation is equivalent to its specifications [3].

6 Conclusion

We introduced a formal verification framework interfacing ASMs (Abstract State Machines) to the MDG (Multiway Decision Graph) tool. This new interface, called "ASM-MDG", enables ASM users to exploit the fully automated verification techniques provided by the MDG tool, namely equivalence checking and model checking. On the other hand, MDG users will be provided with a high-level modeling language, namely ASM, which as MDG, supports abstract data sorts and uninterpreted functions. The interface automatically transforms models in the ASM specification language, ASM-SL, into descriptions in the MDG hardware description language, MDG-HDL. This transformation is done in two ways. Firstly, we translate ASM-SL behavioral and structural models into an intermediate language, ASM-IL, and then transform this intermediate model into the appropriate MDG-HDL behavioral code. Secondly, 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 the algebraic specification necessary for the checking procedures, such as sort and function definitions.

We have applied the ASM-MDG interface on the Fairisle ATM switch controller as a larger case study. We conducted MDG model checking on the generated MDG-HDL models and succeeded in verifying several properties on the switch controller. Although the case study of the Fairisle ATM switch 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 problems, which can be modeled on different levels of abstraction in the same formalism (namely ASM). Furthermore, the case study clearly demonstrates the benefits of the MDG tool over ordinary ROBDD-based tools, like VIS, namely, parameterized models can be checked without instantiating the parameters. In the case of the ATM switch Controller, the model could be checked for an arbitrary number of transferred bytes over the switch.

[3] 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/

As a future work, we think that linking our work to the MDG-HOL [19] hybrid tool will further enable theorem proving for ASM models. Then, different verification approaches can be applied on one model: equivalence checking, model checking and theorem proving. This is not yet available in any verification framework for ASMs. The idea is basically to divide the verification process into tasks, that can be scheduled semi-automatically according to the most suitable verification approach. Interfacing new ASM languages, such as AsmL [24], to the MDG tool can be interesting and implemented following the same approach. Also properties can be specified in a standard language such as PSL [1] instead of the limited syntax of the $\mathcal{L}_{\mathcal{MDG}}$.

References

[1] Accellera Organization. Accellera Property Specification Language Reference Manual, version 1.1. www.accellera.org, 2007.

[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 Theory and Practice of Software Development, LNCS 1214, Springer-Verlag, 1997, pp. 202–212.

[4] E. Börger and R. Stark. Abstract State Machines: A method for High-Level System Design and Analysis. Springer Verlag, 2003.

[5] E. Börger and J. Huggins. Abstract State Machines 1988-1998: Commented ASM Bibliography. In Formal Specification Column, EATCS Bulletin 64, February 1998, pp. 105–127.

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

[7] E. Cerny, F. Corella, M. Langevin, X. Song, S. Tahar, Z. Zhou. Automated Verification with Abstract State Machines Using Multiway Decision Graphs. In Formal Hardware Verification: Methods and Systems in Comparison, LNCS 1287, State-of-the-Art Survey, Springer-Verlag, 1997, pp. 79–113.

[8] 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.

[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] K. Fisler and S. Johnson, Integrating design and verification environments through a logicsupporting hardware diagrams. In Proc. IFIP Conference on Hardware Description Languages. IEEE Computer Society Press, Sep 1995, pp. 669–674.

[11] A. Gargantini and E. Riccobene, Encoding Abstract State Machines in PVS. Abstract State Machines. TIK-Report 87, Swiss Federal Institute of Technology (ETH) Zurich, March 2000, pp. 152–173.

[12] A. Gawanmeh, S. Tahar and K. Winter: Interfacing ASMs with the MDG Tool. In Abstract State Machines - Advances in Theory and Applications, LNCS 2589, Springer Verlag, March 2003, pp. 278–292.

[13] A. Gawanmeh, S. Tahar, and K. Winter: Formal Verification of ASM Designs using the MDG Tool. In IEEE International Conference on Software Engineering and Formal Methods. IEEE Computer Society Press, September 2003, pp. 210–219.

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

[15] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In Specification and Validation Methods, Oxford University Press, 1995.

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

[17] S. Katz and O. Grumberg. A Framework for Translating Models and Specification. In Integrated Formal Methods. LNCS 2335, Springer- Verlag, 2002, pp. 145–164.

[18] 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.

[19] S. Kort, S. Tahar, and P. Curzon: Hierarchical Formal Verification Using a Hybrid Tool. In International Journal on Software Tools for Technology Transfer, Vol. 4, Springer Verlag, 2002, pp. 1–10.

[20] I. Leslie and D. McAuley. Fairisle: An ATM Network for the Local Area. ACM Communication Review, Vol. 19, No. 4, Sep. 1991, pp. 327-336.

[21] J. Lu. On the Formal Verification of ATM Switches, MaSc. Thesis, Concordia University, Canada, 1999.

[22] J. Lu, S. Tahar, D. Voicu, and X. Song. Model Checking of a real ATM Switch. Proc. IEEE International Conference on Computer Design. October 1998, IEEE Computer Society Press, pp. 195–198.

[23] M.L. McMillan. Symbolic Model Checking, Kluwer, 1993.

[24] AsmL for Microsoft .NET (version 2.1.5.7), Microsoft. http://www.research.microsoft.com/foundations/asml, 2003.

[25] S. Owre, J.M. Rushby, and N. Shankar. PVS: a Prototype Verification System. In Automated Deduction, LNCS 607, Springer-Verlag, 1992, pp. 748–752.

[26] G. Schellhorn. Verification of Abstract State Machines. PhD thesis, University of Ulm, Germany, 1999.

[27] N. Shankar. Symbolic Analysis of Transition Systems. In Abstract State Machines, Theory and Applications, LNCS 1912, Springer-Verlag, 2000, pp. 287–302.

[28] M. Spielmann. Automatic verification of abstract state machines. In Computer Aided Verification, LNCS 1633, Springer Verlag, 1999, pp 431–442.

[29] 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.

[30] K. Winter. Model Checking Abstract State Machines, Ph.D. Thesis, Technical University of Berlin, Germany, 2001.

[31] Y. Xu, E. Cerny, X. Song, F. Corella, O. Mohamed. Model Checking for First-Order Temporal Logic using Multiway Decision Graphs. In Computer Aided Verification, LNCS 1427, Springer Verlag, 1998, pp. 219–231.

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

[33] 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.

[34] 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.

[35] Z. Zhou, X. Song, S. Tahar, E. Cerny, F. Corella, M. Langevin. Formal Verification of the Island Tunnel Controller using Multiway Decision Graphs. In Formal Methods in Computer-Aided Design, LNCS 1166, Springer Verlag, 1996, pp. 233–246.