CORE
Metadata, citation and similar papers at core.ac.uk
Provided by University of Queensland eSpace

SOFTWARE VERIFICATION RESEARCH CENTRE

SCHOOL OF INFORMATION TECHNOLOGY

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 01-20

Supporting Abstraction when Model Checking ASM

Kirsten Winter

Version 1, June 2001

Phone: +61 7 3365 1003
Fax: +61 7 3365 1533
http://svrc.it.uq.edu.au

To appear in *Proceedings of ASM'2001 Workshop*.

Note: Most SVRC technical reports are available via anonymous ftp, from `svrc.it.uq.edu.au` in the directory `/pub/techreports`. Abstracts and compressed postscript files are available via `http://svrc.it.uq.edu.au`

Supporting Abstraction when Model Checking ASM

Kirsten Winter

Software Verification Research Centre,
The University of Queensland, Queensland 4072, Australia
(*kirsten@svrc.uq.edu.au*)

Abstract. Model checking as a method for automatic tool support for verification highly stimulates industry's interests. It is limited, however, with respect to the size of the systems' state space. In earlier work, we developed an interface between the ASM Workbench and the SMV model checker that allows model checking of finite ASM models. In this work, we add a means for abstraction in case the model to be checked is infinite and therefore not feasible for the model checking approach. We facilitate the ASM specification language (ASM-SL) with a notion for abstract types and introduce an interface between ASM-SL and *Multiway Decision Graphs* (MDGs). MDGs are capable of representing transition systems with abstract types and functions and provide the functionality necessary for symbolic model checking. Our interface maps *abstract ASM models* into MDGs in a semantic preserving way. It provides a very simple means for generating abstract models that are infinite but can be checked by a model checker based on MDGs.

1 Introduction

In previous work, we showed how model checking can be applied to ASM (c.f., [Win97] and [CW00]). We implemented an interface from the ASM Workbench (see [Cas00]) to the model checker SMV for translating ASM models (given in the specification language of the Workbench, ASM-SL) into SMV code. By means of two case studies, we showed how to make use of this approach: errors can be found in the early phase of developing a model through the counterexamples provided by the model checker. This approach serves very well for debugging and provides a better insight into the ASM model. Due to its fully automatic nature, model checking is very appealing for industry.

The main disadvantage of model checkers, however, is given by its limitation. Since it is a fully algorithmic approach, the model under consideration has to be finite and, moreover, it has to be small enough in order to provide termination of the checking procedure in a suitable time. *Abstraction* is a means to adjust the size of the model.

ASM supports the idea of modelling on different levels of abstraction already. However, this choice of a modelling level is guided by understandability and succinctness of the specification text rather than by reducing the size of the

model's state space. From the viewpoint of model checking, more abstract models may be even bigger than the concrete ones because they may use some infinite data type in order to abstract from implementation details. For instance, in a protocol specification, passing of messages may simply be specified by means of the `extend`-rule applied to an infinite domain of messages in transit rather than introducing the notion of queues (c.f., [Dur98]). Such a model with infinite domains, however, is not feasible for model checking. Therefore, we need another means for supporting either abstraction or the treatment of abstract models that include infinite domains.

In the literature concerned with model checking, most attempts for supporting abstraction are based on the idea that an abstraction function is applied to the given *concrete* model, which is too large to be treated, for computing an *abstract* model, which can be handled by the model checker. Two tasks have to be solved: firstly, finding an appropriate abstraction function and secondly, computing the abstract model and proving that it preserves the properties of the concrete model (see e.g., [GS97,SS99,BH99]). This technique involves the use of an (interactive) theorem prover and some insight of how to define the abstraction function.

Another direction aims at the treatment of infinite systems by means of *uninterpreted functions*. Functions with an infinite domain or range that are not relevant to the properties to be checked are treated "symbolically" instead of exploring their infinitely many possible evaluations. For model checking hardware systems, uninterpreted functions are used in [BD94,CN94].

In our work, we follow the second direction: We introduce a notion for *abstract types* in the ASM-SL. We regard models that comprise abstract types as *abstract ASM* since functions over these abstract types have no fixed interpretation. We provide a mapping from abstract ASM to *Multiway Decision Graphs* (MDGs) ([CZS$^+$97,CCL$^+$97]), a graph structure that is capable of representing abstract transition systems. MDGs are a generalisation of BDDs and provide the basic functionality for symbolic model checking. We refer to the library of these functions as the MDG-Package ([Zho96]). Exploiting this framework, the user interaction for generating an abstract model is limited to the task of choosing the types to be abstracted.

This paper is organised as follows: ASM and MDGs are presented briefly in Sections 2 and 3. Section 4 describes the generation of abstract models that can be represented by MDGs. In Section 5, the transformation from ASM into MDGs is given in two steps: Firstly, ASM are mapped into the intermediate language ASM-IL+ (Subsection 5.1). Secondly, ASM-IL+ is transformed into MDGs (Subsection 5.2). Section 6 relates our work to others' and Section 7 gives a summary.

2 Abstract State Machines

In this section we introduce some basic notions of ASM (see [Gur95] for the complete definition). We first describe the underlying computational model and

then the syntax and semantics of the subset of the ASM language needed in this paper.

2.1 Computational Model

Computations Abstract State Machines define a state-based computational model, where computations (*runs*) are finite or infinite sequences of states $\{S_i\}$, obtained from a given *initial state* S_0 by repeatedly executing *transitions* δ_i:

$$S_0 \xrightarrow{\delta_1} S_1 \xrightarrow{\delta_2} S_2 \ \dots \ \xrightarrow{\delta_n} S_n \ \dots$$

States The *states* are algebras over a given *signature* Σ (or Σ-*algebras* for short). A signature Σ consists of a set of *basic types* and a set of *function names*, each function name f coming with a fixed arity n and type $T_1 \dots T_n \to T$, where the T_i and T are basic types (written $f : T_1 \dots T_n \to T$, or simply $f : T$ if $n = 0$). for each function name $f : T_1 \dots T_n \to T$ in Σ (the *interpretation* of the function name f in S). Function names in Σ can be declared as:

- *static*: static function names have the same (fixed) interpretation in each computation state;
- *dynamic*: the interpretation of dynamic function names can be altered by transitions fired in a computation step (see below);
- *external*: the interpretation of external function names is determined by the environment (thus, external functions may change during the computation as a result of environmental influences, but are not controlled by the system).

Any signature Σ must contain at least a basic type $BOOL$, static nullary function names (constants) $\mathit{true} : BOOL$, $\mathit{false} : BOOL$, the usual boolean operations ($\wedge$, $\vee$, etc.), and the equality symbol $=$. We also assume that there is a (polymorphic) type $SET(T)$ of finite sets with the usual set operations. When no ambiguity arises we omit explicit mention of the state S (e.g., we write $\mathcal{T}$ instead of $\mathcal{T}^S$ for the carrier sets, and $\mathbf{f}$ instead of $\mathbf{f}_S$ for static functions, as they never change during a run).

Locations If $f : T_1 \dots T_n \to T$ is a dynamic or external function name, we call a pair $l = (f, \overline{x})$ with $\overline{x} \in \mathcal{T}_1 \times \dots \times \mathcal{T}_n$ a *location* (then, the *type* of l is T and the *value* of l in a state S is given by $\mathbf{f}_S(\overline{x})$). Note that, within a run, two states S_i and S_j are equal iff the values of all locations in S_i and S_j are equal (i.e., they coincide iff they coincide on all locations).

Transitions Transitions transform a state S into its successor state S' by changing the interpretation of some dynamic function names on a finite number of points (i.e., by updating the values of a finite number of *locations*).

More precisely, the transition transforming S into S' results from firing a finite *update set* Δ at S, where *updates* are of the form $((f, \overline{x}), y)$, with $(f, \overline{x})$ being the location to be updated and y the value. In the state S' resulting from firing Δ at S the carrier sets are unchanged and, for each function name f:

$$\mathbf{f}_{S'}(\overline{x}) \ = \ \begin{cases} y & \text{if } ((f, \overline{x}), y) \in \Delta \\ \mathbf{f}_S(\overline{x}) & \text{otherwise.} \end{cases}$$

Note that the above definition is only applicable if Δ does not contain *conflicting updates*, i.e., any updates $((f, \overline{x}), y)$ and $((f, \overline{x}), y')$ with $y \neq y'$.

The update set Δ—which depends on the state S—is determined by evaluating in S a distinguished closed *transition rule* P, called the *program*. The program consists usually of a set (block) of rules, describing system behavior under different—usually mutually exclusive—conditions.

2.2 The ASM Language

Terms Terms are defined as in first-order logic: *(i)* if $f : T_1 \ldots T_n \rightarrow T$ is a function name in Σ, and t_i are terms of type T_i (for $i = 1, \ldots, n$), then $f(t_1, \ldots, t_n)$ is a term of type T (written $t : T$) (if $n = 0$ the parentheses are omitted, i.e. we write f instead of $f()$); *(ii)* a variable v (of a given type T) is a term. The meaning of a term $t : T$ in a state S and environment ρ is a value $S_\rho(t) \in \mathcal{T}$ defined by:[1]

$$S_\rho(t) \;=\; \begin{cases} \mathbf{f}_S(S_\rho(t_1), \ldots, S_\rho(t_n)) & \text{if } t \equiv f(t_1, \ldots, t_n) \\ \rho(v) & \text{if } t \equiv v. \end{cases}$$

As opposed to first-order logic, there is no notion of formula: boolean terms are used instead. Finite quantifications of the form "$(Q\ v\ \texttt{in}\ A : G)$", where Q is $\forall$ or $\exists$, $v : T$, $A : \mathit{SET}(T)$, and $G : \mathit{BOOL}$, are also valid boolean terms.[2]

Transition rules While terms denote values, transition rules (*rules* for short) denote *update sets*, and are used to define the dynamic behavior of an ASM: the meaning of a rule R in a state S and environment ρ is an update set $\Delta_{S,\rho}(R)$.

ASM runs starting in a given initial state S_0 are determined by the program P: each state S_{i+1} ($i \geq 0$) is obtained by firing the update set $\Delta_{S_i}(P)$ at S_i:

$$S_0 \;\overset{\Delta_{S_0}(P)}{\longrightarrow}\; S_1 \;\overset{\Delta_{S_1}(P)}{\longrightarrow}\; S_2 \;\ldots\; \overset{\Delta_{S_{n-1}}(P)}{\longrightarrow}\; S_n \;\ldots$$

Basic transition rules are the *skip*, *update*, *block*, and *conditional* rules. Additional rules are the *do-forall* (a generalized block rule) and *choose* rules (for non-deterministic choice).[3]

$$\begin{array}{rl} R ::= & \texttt{skip} \;\mid\; f(t_1, \ldots, t_n) := t \;\mid\; R_1 \ldots R_n \;\mid\; \texttt{if}\ G\ \texttt{then}\ R_T\ \texttt{else}\ R_F \\ & \mid\; \texttt{do forall}\ v\ \texttt{in}\ A\ \texttt{with}\ G\ \ R' \;\mid\; \texttt{choose}\ v\ \texttt{in}\ A\ \texttt{with}\ G\ \ R' \end{array}$$

The form "`if` G `then` R" is a shortcut for "`if` G `then` R `else skip`". Omitting "`with` G" in *do-forall* and *choose* rules corresponds to specifying "`with true`".

[1] Environments—denoted by the letter ρ—are finite maps containing bindings which associate (free) variables to their corresponding values. We adopt the following notation: $\rho[v \mapsto x]$ is the environment obtained by modifying ρ to bind v to x, while $\rho \setminus v$ is the environment with the binding of variable v removed from ρ. For closed terms and rules, we omit explicit mention of ρ (e.g., if t is a closed term, $S(t) = S_\emptyset(t)$).

[2] Also in the rest of this paper we use A for set-typed terms and G for boolean terms.

[3] The ASM Workbench support more rules, such as *let* and *case* rules with pattern matching: however, for reasons of space, we have to skip them here.

The semantics of transition rules is as follows:

$$\begin{aligned}
\Delta_{S,\rho}(\,\texttt{skip}\,) &= \{\ \} \\
\Delta_{S,\rho}(\,f(t_1,\ldots,t_n) := t\,) &= \{\,((f,(S_\rho(t_1),\ldots,S_\rho(t_n))),S_\rho(t))\,\} \\
\Delta_{S,\rho}(\,R_1\ \ldots\ R_n\,) &= \textstyle\bigcup_{i=1}^{n} \Delta_{S,\rho}(\,R_i\,) \\
\Delta_{S,\rho}(\,\texttt{if}\ G\ \texttt{then}\ R_T\ \texttt{else}\ R_F\,) &= \begin{cases} \Delta_{S,\rho}(\,R_T\,) \text{ if } S_\rho(G) = \textbf{true} \\ \Delta_{S,\rho}(\,R_F\,) \text{ otherwise} \end{cases}
\end{aligned}$$

$$\Delta_{S,\rho}(\,\texttt{do forall}\ v\ \texttt{in}\ A\ \texttt{with}\ G\ \ R'\,) \ = \ \textstyle\bigcup_{x \in X} \Delta_{S,\rho[v \to x]}(\,R'\,)$$
$$\text{where } X = \{x \mid x \in S_\rho(A) \wedge S_{\rho[v\to x]}(G) = \textbf{true}\}.$$

Note that executing a block (or a do-forall) rule corresponds to *simultaneous* execution of its subrules[4] and may lead to conflicts.

Choose rules are not directly supported by our transformation tool, but can always be replaced by external functions for arbitrary choice of a value (by a transformation similar to skolemization). For example, let A_i be terms of type $SET(T_i)$, $i = 1, 2, 3$, and $f_x : T_1$, $f_z : T_2 \to T_3$ external functions with $f_x \in A_1$ and $f_z(y) \in A_3$ for each $y \in A_2$. Then the following two rules are equivalent:

```
choose x in A1
  do forall y in A2                 ≅    do forall y in A2
    choose z in A3                         a(fx, y, fz(y)) := fx + y + fz(y)
      a(x, y, z) := x + y + z
```

Multi-Agent ASM Concurrent systems can be modelled in ASM by the notion of multi-agent ASM (called *distributed ASM* in [Gur95]). The basic idea is that the system consists of more *agents*, identified with the elements of a finite set $AGENT$ (which are actually sort of "agent identifiers"). Each agent $a \in AGENT$ executes its own program $prog(a)$ and can identify itself by means of a special nullary function $self : AGENT$, which is interpreted by each agent a as a.

As a semantics for multi-agent ASM we consider here a simple interleaving model, which allows us to model concurrent systems in the basic ASM formalism as described above. In particular, we consider *self* as an external function, whose interpretation $\mathbf{self}_{S_i}$ determines the agent which fires at state S_i. We assume that there is one program P, shared by all agents, possibly performing different actions for different agents, e.g.:

```
if self = a1 then prog(a1)
...
if self = an then prog(an)
```

where $\{a_1, \ldots, a_n\}$ are the agents and $prog(a_i)$ is the rule to be executed by agent a_i, i.e., the "program" of a_i.

[4] For example, a block rule `a` := `b`, `b` := `a` exchanges `a` and `b`.

3 Multiway Decision Graphs

Multiway Decision Graphs (MDGs) are a generalisation of Binary Decision Diagrams (BDDs). They are a data structure for canonically representing formulas of a many-sorted first-order logic, called *Directed Formulas* (DFs). A special feature of the underlying logic is the distinction between *concrete* and *abstract* sorts. Correspondingly, function symbols may be concrete, abstract (if the range is abstract), or cross-operators (if the range is concrete but the domain contains some abstract sort).

DFs are suitable for describing sets of states and transition relations of transition systems. They are formulas in disjunctive normal form (DNF) over simple equations of the following form: $f(B_1, \ldots, B_n) = a$ (where f is a cross-operator and a is a constant of concrete sort), $w = a$ (where w is a variable of concrete sort and a is a concrete constant), or $v = A$ (where v is a variable of abstract sort and A is a term of the same sort). Furthermore, in each disjunct of a DF, all left hand sides (LHSs) of the equations are pairwise distinct and every abstract variable that occurs as a LHS must occur in every disjunct of the DF. To be represented by MDGs, a DF has to be *concretely reduced*, i.e., all concrete variables that occur on the right-hand side (RHS) of an equation have to be substituted by a value.

An MDG is a finite graph G, whose non-terminal nodes are labelled by terms and whose edges are labelled by terms of the same sort as the node. Terminal nodes are labelled by formulas. Generally, a graph G represents a formula in the following way:

- If G consists of a single terminal node, then it represents the formula the node is labelled with.
- If G has a root node labelled with term A and edges labelled with terms $B_1, \ldots, B_n$ leading to subgraphs $G_1, \ldots, G_n$ that represent formulas $P_1, \ldots, P_n$, then G represents the formula

$$(A = B_1 \wedge P_1) \vee (A = B_2 \wedge P_2) \vee \ldots \vee (A = B_n \wedge P_n)$$

In Figure 1, we depict the formula that is given above as a graph G. To be a canonical representation, an MDG has to satisfy certain *well-formedness conditions* which involve an order on function symbols and variables that has to be provided by the user (the detailed list of conditions is defined in [CZS+97,CCL+97]).

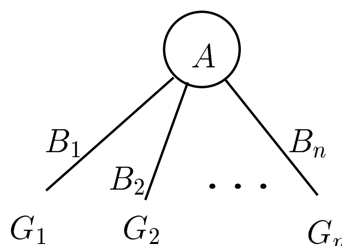

Fig. 1: The MDG G

A library of operations on MDGs is available that is sufficient for realising an *implicit* state enumeration, namely disjunction, *relational product*, and *pruning-by-subsumption* (see [Zho96]). They are defined for combining sets rather than only pairs of MDGs. This allows us to represent the transition relation as a set of several small graphs instead of one

big graph and benefits from a more efficient state enumeration.

The relational product operation computes the conjunction of a set of graphs under existential quantification of all variables in a given variable set E and the possible renaming η of variables, $((\exists v \in E)(\bigwedge_{1\leq i\leq n} P_i) \cdot \eta)$. This is used for computing the set of reachable states from a given state (in which case the MDGs P_i represent the transition relation and E the set of state variables). Pruning-by-subsumption approximates the difference of two sets. This enables us to check if an invariant (given as an MDG) is satisfied in a given set of states (also given as an MDG).

According to the well-formedness conditions on MDGs, the application of the operations is restricted. The relational product of two MDGs can only be computed if their nodes are not labelled with the same abstract variable. Disjunction, in contrast, is applicable only to MDGs that contain the same set of abstract variables as nodes labels (i.e., the same set of abstract variables that occur as LHSs in the equations of the corresponding DF).

4 Generating Abstract ASM

In order to make use of the concept of abstract sorts in MDGs, we introduce a syntactic feature into ASM-SL for indicating that a type is abstract. Given this, we are able to automatically indicate abstract functions and cross-terms as well. We get a simple tool for computing an *abstract ASM* once the user has chosen the types that are considered to be abstract.

In an abstract ASM, abstract functions and variables are treated as being *uninterpreted*, i.e., every interpretation is possible. We may describe the step of abstraction as stripping off semantics by giving up information about interpretation. Thus, the abstract specification is a model for all structures with the same signature. That is, an abstract ASM describes a set of concrete ASM with suitable (with respect to the signature) interpretations for abstract sorts, functions, variables, and cross-terms. Figure 2 depicts the abstraction step: We consider the sort Q in our concrete model as abstract and change all its occurrences into $Data_{abs}$. As a result, we get an abstract model of the same signature. For this abstract specification, all those interpretations are suitable that have a sort, a 2-ary function that maps arguments of the given sort to a value of the same sort, and a boolean predicate over the sort. In the figure, we give some examples for different interpretations for the sort $Data_{abs}$ and the functions that are possible.

The purpose of this abstraction step is to substitute infinite sorts, and functions over them, since these cannot be exhaustively explored. The use of cross-terms on abstract sorts and their complete case distinction naturally provides a partitioning of the infinite sort into finitely many equivalence classes. The state space of the abstract model is smaller in most cases. It can be canonically represented by MDGs and the corresponding model checking algorithm can be applied to check certain properties.

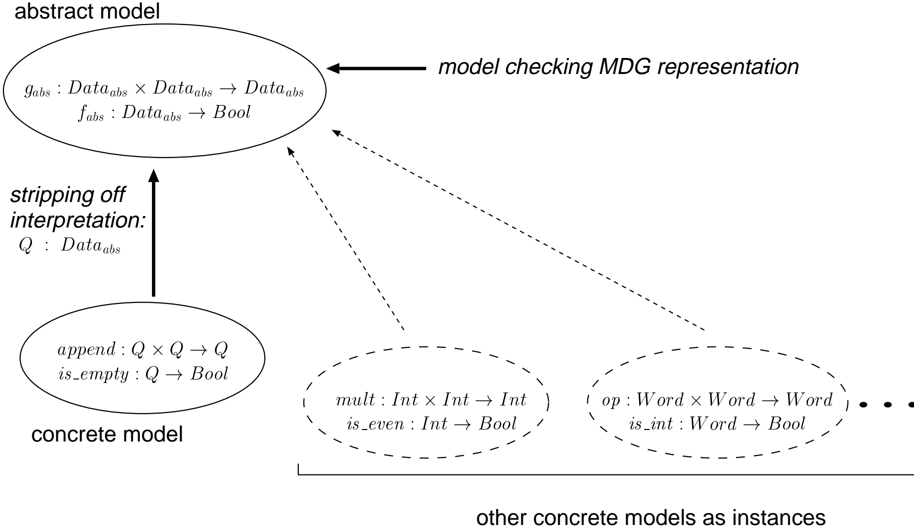

Fig. 2. Lifting a concrete model to an abstract model

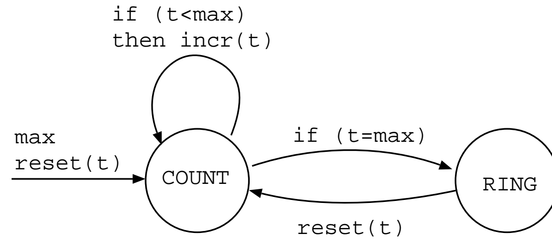

Fig. 3: Example of a generic timer

As a simple example, consider the specification of a generic timer in Figure 3: the system gets as an input value any natural number *max* that specifies the number the timer has to count to. The timer has two states, `COUNT` and `RING`. As long as the system is in state `COUNT` it increments the state variable t in every step. Once t has reached the limit max the system changes into state `RING`; a bell might ring to give a signal. In the next step, t is reset and the timer starts again counting to max.

This system specification can easily be abstracted by treating the natural numbers $\mathcal{N}$ as abstract sort $\mathcal{N}_{abs}$. We replace the equality relation on natural number by a new predicate *isEq* that compares two abstract values of sort $\mathcal{N}_{abs}$; $isEq(t, max)$ may evaluate to *true* if the (abstract) arguments are equal, otherwise it evaluates to *false*. The functions *incr* and *reset* turn into abstract functions that map any value of sort $\mathcal{N}_{abs}$ into a value of the same sort.

In the MDG approach, predicates like *isEq* are cross-term symbols; they are applied to abstract terms and evaluate to a value of concrete sort (in our example, boolean values). When model checking, we can use cross-terms since their range is a concrete type that can be enumerated. Without any knowledge of the value of the abstract parameter, we simply unfold the different (enumerate-able) cases and explore them when model checking. This may lead to exploring states that

do not occur in the concrete model and hence may yield wrong counterexamples. One means for addressing this problem is to add rewrite rules to the model which restrict the possible interpretations of abstract sorts and functions (for further investigation of this problem see also [ZST+96,MSC97]).

Lifting a model to a higher level of abstraction can be done automatically within our transformation step from ASM to MDG provided the developer has chosen the domains of the ASM that should be considered as abstract sorts. Functions in the ASM model that involve abstract data are automatically treated as abstract functions or cross-terms. In contrast to other approaches for generating abstraction, there is no extra effort necessary to change the model under investigation other than the change of the data type definition.

5 The Transformation of ASM into MDG

In order to benefit from the notion of abstract sorts for supporting abstraction, we have to extend the ASM language with a syntactic feature that allows a sort to be marked as abstract. This extension requires an extension of the basic transformation algorithm (see [CW00]) as well. The extended algorithm is introduced in the following two steps: Firstly, we detail the adaption of the basic transformation algorithm that transforms ASM models into the extended intermediate language ASM-IL$^+$. Secondly, we develop a transformation from ASM-IL$^+$ models into MDGs. We have to justify that ASM-IL$^+$ rules represent DFs which can be canonically represented by MDGs. The second transformation step completes the interface from the ASM Workbench to the MDG-Package.

5.1 The Adapted Transformation into ASM-IL$^+$

The original transformation algorithm maps all occurrences of dynamic and external functional terms $f(t_1, \ldots, t_n)$ into locations $f(a_1, \ldots, a_n)$, which can be renamed to simple state variables. This is done by simplifying and unfolding: The evaluation of parameter terms t_i to elements a_i may be state dependent. Each possible evaluation of these terms leads to a different location. The result of the unfolding procedure is a set of locations that are given by all possible evaluations of the dynamic parameter terms.

Functions and terms of abstract sort cannot be similarly evaluated; their interpretation is not specified. Unfolding a term involves evaluating it, i.e., adding the state dependent interpretation to a term. Therefore, an *uninterpreted* function cannot be unfolded in any term in which it appears. In order to implement this special treatment for abstract functions, we have to restrict the unfolding procedure to functions that are concrete.

Adapted simplification function. The transformation algorithm for simplification and unfolding follows an inductive schema. The treatment of abstract functions and cross-terms is easily introduced into this inductive schema by adding some case distinctions to the term simplification: abstract terms should not be

values, locations and variables are mapped to

$[\![a]\!]_\zeta = a$ values and $[\![\, loc \,]\!]_\zeta = loc$ locations

$$[\![\, v \,]\!]_\zeta = \begin{cases} a = \zeta(v) & \text{if } v \in \mathrm{dom}(\zeta) \text{ and of } \textbf{concrete} \text{ sort} \\ v^{abs} & \text{otherwise} \end{cases}$$

for applied functions we distinguish

$[\![t_i]\!]_\zeta = a_i$ for each $i \in \{1, \ldots, n\}$ $\Rightarrow$

$$[\![f(t_1, \ldots, t_n)]\!]_\zeta = \begin{cases} a = f^A(a_1, \ldots, a_n) & \text{if } f \text{ is a static function name of } \textbf{concrete} \text{ sort} \\ loc = (f, (a_1, \ldots, a_n)) & \text{if } f \text{ is a dynamic/external function name of } \textbf{concrete} \text{ sort} \\ f^{abs}(a_1, \ldots, a_n) & \text{if } f \text{ is a static function name of } \textbf{abstract} \text{ sort} \\ loc^{abs} = (f^{abs}, (a_1, \ldots, a_n)) & \text{if } f \text{ is a dynamic/external function name of } \textbf{abstract} \text{ sort} \end{cases}$$

if all arguments are of **constant** sort and
$[\![t_i]\!]_\zeta = loc$ or $[\![t_i]\!]_\zeta = f'(\bar{t}')$ for some $i \in \{1, \ldots, n\}$ $\Rightarrow$

$$[\![f(t_1, \ldots, t_n)]\!]_\zeta = f([\![t_1]\!]_\zeta, \ldots, [\![t_n]\!]_\zeta)$$

if some arguments t_i are of **abstract** sort and $[\![t_i]\!]_\zeta = t^{abs}$ $\Rightarrow$

if f is a **cross-term** operator or a **static** function name:

$$[\![f(t_1, \ldots, t_n)]\!]_\zeta = f([\![t_1]\!]_\zeta, \ldots, [\![t_n]\!]_\zeta)$$

if f is an **abstract** function name:

$$[\![f(t_1, \ldots, t_n)]\!]_\zeta = (f^{abs}, ([\![t_1]\!]_\zeta, \ldots, [\![t_n]\!]_\zeta))$$

Table 1. Adapted Term Transformations

unfolded to any evaluation. The modified definition for term simplification is summarised in Table 1.

The simplification function $[\![.]\!]_\zeta$ is related to the variable assignment $\zeta : V \rightarrow S^A$ since all concrete variables in V are mapped to the values of the corresponding concrete sort. The base of the induction schema is given by simple terms. We distinguish constant *values* a, *locations* loc, that may change their values at a transition step, and *variables* v, which can be evaluated to values a if they are **concrete** (according to ζ) or left as uninterpreted variables v^{abs} if they are **abstract**[5]. We extend this base by *functional terms*: Functional terms are **abstract** functions applied to terms t_i that can be abstract terms or concrete values. These terms cannot be unfolded further; the simplification terminates. Any dynamic or external function turns into a *locational term*. Locational terms can be simple, i.e., concrete locations loc, abstract functional terms loc^{abs}, or cross-terms $f(t_1, \ldots, t_n)$.

[5] The index *abs* is used in Table 1 for indicating that a variable or term is of abstract sort or is a cross-term.

Locational terms that are abstract functions (and not cross-terms) are mapped into simple variables by merging function symbol and parameter symbols into one new variable name. This is indicated through the pair (f^{abs},$(t_1, \ldots, t_n)$). In contrast, any static functional term, and also any cross-term, is kept as a function application: $f(t_1, \ldots, t_n)$. The term simplification $[\![.]\!]_\zeta$ (c.f., Table 1) works inductively over arbitrary function application: concrete parameters are unfolded, abstract parameters are left unchanged. We distinguish between terms whose parameters are all concrete values, terms whose parameters are concrete but have to be unfolded further, and terms that have some parameters of abstract sort:

1. If all parameters are (unfolded to) values, we distinguish between
   - static functions that are **concrete**; these have to be evaluated by means of applying the function interpretation f^A to the parameter values;
   - dynamic or external functions that are **concrete**; these turn into a location *loc* by means of merging function symbol and parameter symbols; any location *loc* is thus equivalent to a simple state variable;
   - static functions of **abstract** sort; these are left as uninterpreted functions f^{abs} which are not applied; they are denoted as functional terms $f^{abs}(a_1, \ldots a_n)$.
   - dynamic or external functions of **abstract** sort; we denote these terms as loc^{abs}, a locational term that can change its value, and proceed similar to concrete locations: we merge the function symbol and the parameter symbols and get an abstract state variable.
2. If all parameters are of **concrete** sort but some of them are non-values, i.e., locations *loc* or other function applications $f'(\bar{t}')$ that have to be unfolded further, we apply the unfolding function to the parameters first.
3. If some parameters are **abstract** terms t^{abs}, i.e., abstract variables v^{abs} or abstract functions $f^{abs}(t_1, \ldots, t_n)$, we simplify the parameters by $[\![.]\!]_\zeta$ first; two cases may occur:
   - the function is a cross-operator or a static abstract function; in this case we keep the function;
   - the function is an abstract function; we merge the function symbol and the parameter symbols into one abstract variable.

First-order terms of the form $(\exists v : g(v))\ s(v)$ or $(\forall v : g(v))\ s(v)$ are simplified as in the original transformation (see [Win01]) as long as their range is a concrete and finite set. If the range of a first-order term turns into an abstract range by means of applying abstraction to the domain S_i^A the range is not enumeratable any more. The simplification keeps the head variable v as an abstract variable and treats the body $s(v)$ as a cross-term. This is shown in Table 2 by applying $[\![.]\!]_\zeta$ to the body. For abstract head variables v, $[\![s(v)]\!]_\zeta$ implicitly models universal quantification over every $a \in S_i^A$. Existential quantification over an abstract variable is logically not expressible.

Note that uninterpreted, abstract functional terms $f^{abs}(t_1, \ldots t_n)$, where f^{abs} is a static abstract function and t_i is a value or another abstract term, or $f(t_1, \ldots t_n)$ where f is a cross-term operator, are not mapped into a state variable name. In contrast to the SMV approach, these functional terms may appear

quantified terms over finite ranges

$\{a \in S_i^A \mid B \models g(a)\}$ where S_i^A is a **concrete** sort:

$$[\![((\exists v : g(v))\ s(v))]\!]_\zeta \ = \ [\![s(v)]\!]_{\zeta[v \mapsto a_1]} \vee \ldots \vee [\![s(v)]\!]_{\zeta[v \mapsto a_n]}$$

$$[\![((\forall v : g(v))\ s(v))]\!]_\zeta \ = \ [\![s(v)]\!]_{\zeta[v \mapsto a_1]} \wedge \ldots \wedge [\![s(v)]\!]_{\zeta[v \mapsto a_n]}$$

quantified terms over **abstract** ranges

$\{a \in S_i^A \mid B \models g(a)\}$ where S_i^A is an **abstract** sort:

$$[\![((\forall v : g(v))\ s(v))]\!]_\zeta \ = \ [\![s(v)]\!]_\zeta$$

($[\![((\exists v : g(v))\ s(v))]\!]_\zeta$ is not defined; see remark below)

Table 2. Adapted Term Transformation of first-order Terms

as labels in the MDG structure. It is easy to see that abstraction saves alot of unfolding effort. The resulting set of guarded updates is much smaller.

Table 3 summarises the rule simplification that applies the simplification function $[\![.]\!]_\zeta$ (c.f., Table 1) to the terms in the rules. This rule simplification is not changed. We recall the definition here for the readers' convenience.

$$[\![\texttt{skip}]\!]_\zeta \ = \ \texttt{skip}$$

$$[\![f(\bar{t}) \ := \ t]\!]_\zeta \ = \ \texttt{if}\ \mathit{true}\ \texttt{then}\ [\![f(\bar{t})]\!]_\zeta \ := \ [\![t]\!]_\zeta$$

$$[\![\texttt{block}\ R_1 \ldots R_n\ \texttt{endblock}]\!]_\zeta \ = \ [\![R_1]\!]_\zeta \ldots [\![R_n]\!]_\zeta$$

$$[\![\texttt{if}\ g\ \texttt{then}\ R_1\ \texttt{else}\ R_2]\!]_\zeta \ = \ \begin{cases} \texttt{if}\ \ [\![g]\!]_\zeta\ \texttt{then}\ [\![R_1]\!]_\zeta \\ \texttt{if}\ \neg[\![g]\!]_\zeta\ \texttt{then}\ [\![R_2]\!]_\zeta \end{cases}$$

Table 3. Rule Simplifications

Adapted rule unfolding. Table 4 defines the extended rule unfolding. In this rule unfolding, simple update rules are left unchanged. More complex rules are unfolded according to the evaluation of **concrete** locations that can be found in the rule. For every possible evaluation a_i we introduce an instantiation of the rule $R[loc/a_i]$ that is guarded by the equation $(loc = a_i)$, where loc is the first concrete location occurring in the rule R. The substitution $R[loc/a_i]$ formalises that every occurrence of loc is substituted by the value a_i. Note that **abstract** locations are not unfolded; they are left uninterpreted.

Also, all locations that appear as a left-hand side (LHS) of an update or that are parameters of the so called *primary* operators (i.e., equality, conjunction, disjunction, and negation) are excluded from the unfolding procedure due to optimisation issues[6] (c.f., [Win00]).

[6] Note that in the MDG approach, we do not consider arithmetic operations as being primary.

If $R \ = \ loc_1 := a_1 \ \ldots \ loc_n := a_n$:

$$\mathcal{E}(R) = R$$

Otherwise:

$$\mathcal{E}(R) = \begin{cases} \texttt{if } loc = a_1 \texttt{ then } \mathcal{E}([\![R[loc/a_1]]\!]_\zeta) \\ \ldots \\ \texttt{if } loc = a_n \texttt{ then } \mathcal{E}([\![R[loc/a_n]]\!]_\zeta) \end{cases}$$

where
loc is the first location **of concrete sort** occurring in R but not as an LHS of an update rule and not as a parameter of a primary operation;
$\{a_1, \ldots, a_n\}$ is the range of location *loc*.

Table 4. Unfolding of concrete Locations in a Rule

ASM-IL$^+$ representation. As a result of the adapted term simplification and rule unfolding we get a list of pairs over locations and their guarded updates, which is the ASM-IL$^+$ representation for the ASM model.

$$(\ loc_term_i,\ [\ (guard_{i1}, upd_term_{i1}), (guard_{i2}, upd_term_{i2}), \ldots])$$

These pairs are called *location-update pairs* and each pair $(guard_{ij}, upd_term_{ij})$ is called a *guarded-update* pair. In contrast to ASM-IL representations of an ASM model, these pairs may contain abstract variables and cross-terms as LHSs and RHSs, as well as abstract functions as RHSs of equations or updates.

Limitations of the adapted term simplification. We identify two cases in which the adapted term simplification does not provide proper results and thus is not applicable:

1. Any dynamic or external abstract function is mapped into an abstract state variable. For any concrete parameter (that can be unfolded) we create several instances of this state variable. However, abstract parameters are not unfolded; instead of multiple instances we get only one abstract variable. For example, assume that in the concrete ASM we have a sort $A = \{a, b, c\}$ and a dynamic function $f : A \to B$. The location $f(x)$ is simplified into three state variables f_a, f_b, and f_c. Abstracting this model we may change the sorts A and B into abstract sorts. The same location $f(x)$ is now simplified into one variable f_x. The resulting abstract ASM-IL$^+$ model is not a correct abstraction of the concrete model. Note that the same problem does not occur for cross-terms: any function $g : A \to C$, where C is not an abstract sort, is kept as a function application rather than being mapped into a state variable.
2. Existentially quantified first-order terms $(\exists v : g(v))\ s(v)$ where the sort of variable v turns into an abstract sort cannot be simplified properly. The existence of a witness (necessary to evaluate existential quantification) is not decidable since an abstract sort has no concrete (interpreted) entities. A similar problem does not occur for universally quantified terms.

As a consequence of these limitations, we get guidance as to where abstraction should not be applied in our approach: Any sort that is used as a domain (or

part of a cross-product of domains) of an n-ary dynamic or external function should not be abstracted.

Since the adapted transformation algorithm is identical to the original transformation algorithm for terms without abstract sorts, its correctness is based on the proof given in [Win01]. In the case of terms with abstract sorts, the adapted transformation algorithm itself defines the semantics of these terms and therefore no correctness proof is required.

A set of location-update pairs represents an *abstract* transition system, which can be treated by a tool that is based on MDGs. In the next subsection we show that each location-update pair represents a DF in terms of the MDG approach.

ASM-IL$^+$ Models as Directed Formulas For mapping a set of location-update pairs into MDGs, we have to justify that the well-formedness conditions of MDGs are satisfied. In this subsection, we show that our transformation provides directed formulas (DFs) that can be canonically represented by well-formed MDGs (we follow the description of DFs and MDGs in Section 3).

Concretely reduced terms. Concretely reduced DFs are formulas in which all concrete terms on the RHS of an update or equation in such formulas are individual constants. Only these formulas can be canonically represented by well-formed MDGs. As introduced in the last section, the simplification function $[\![.]\!]_\zeta$ unfolds all concrete terms that appear in ASM rules. This way, all terms in location-update pairs are already *concretely reduced* in the sense of the well-formedness conditions of MDGs ([CZS$^+$97]), i.e., all concrete functions and variables are mapped into their values.

Partitioned transition relation. In model checking approaches that are based on decision diagrams (e.g., BDDs and MDGs), the transition relation of a transition system that is to be checked should be partitioned and represented by smaller graphs rather than represented and treated as one large graph (c.f., [BCMD90]). Partitioning helps prevent the (single) representing graph growing too big. Instead of working with one graph for representing the whole transition relation, the algorithms work on a set of smaller graphs, each representing a part of the transition relation only. This technique is adapted for the MDG approach as well (see [ZSC$^+$95]). All algorithms (e.g., relational_product, pruning_by_subsumption, etc.) expect lists of MDGs as input to operate on.

Since in ASM-IL$^+$ every location has an attached list of guarded-update pairs, this representation naturally provides a partitioning of the overall transition relation. Moreover, our partitioning naturally yields MDGs with disjoint sets of primary abstract variables (see below).

Correctness of the mapping The correctness of mapping location-updates pairs into MDGs is based on the following equivalence:

$$
\begin{aligned}
&(loc_term_i, [(guard_{i1}, upd_term_{i1}), \ldots]) \;\Leftrightarrow \\
&\quad \bigvee_j (loc_term_i' = upd_term_{ij} \wedge guard_{ij}) \\
&\quad \vee \left(loc_term_i' = loc_term_i \wedge \bigwedge_j \neg guard_{ij}\right)
\end{aligned} \tag{1}
$$

where loc_term_i' denotes the locational term loc_term_i in the next state. The first part of the disjunction conjoins all guards and corresponding updates. The second part specifies the "else-case"; that is, if none of the guards are true then the location should keep its old value. It can be shown that according to the adapted transformation algorithm introduced in the last subsection, this DNF is a well-formed DF (for a full proof see [Win01]). Each of these DFs has at most one abstract variable on the LHS of an equation: loc_term_i'. As a consequence each location-update pair can be represented as an MDG. These MDGs have pairwise disjoint sets of primary abstract variables. This property is a necessary precondition for applying the algorithms for computing the relational product (see Section 3). In the next section, we introduce our algorithm that interfaces ASM-IL$^+$ with the MDG-Package.

5.2 Transformation of ASM-IL$^+$ into MDGs

In the second step of our transformation, an algorithm maps an ASM-IL$^+$ model into a set of MDGs. In order to do this, we have to treat the updates, the guards of the updates, and the else cases, which specify that the location is not changed if none of the guards are satisfied. Moreover, a variable ordering that satisfies the well-formedness conditions should be suggested.

Representing a location-update pair $(loc_i, [(guard_{i1}, val_{i1}), \ldots])$ as an MDG is straightforward if we consider the equivalence (1). The next state variable loc_i' labels the root node of the graph. Each edge starting at the root is labelled with one of the specified values in the next state val_{ij} and leads to the subgraph G_{ij} that represents the corresponding guard of the update $guard_{ij}$. Figure 4 sketches a graph for a location-update pair.

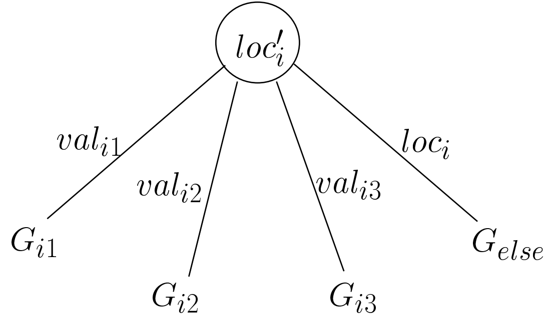

Fig. 4: Location-update pair represented as MDG

Figure 4 shows also that one edge of the MDG is labelled by the current value of the location loc_i and leads to a graph G_{else}. This branch specifies the else-case of the location-update pair: if none of the guards $guard_{ij}$ are satisfied,

and therefore none of the updates can fire, the location should keep its value in the next state. For model checking, this semantics has to be specified explicitly. Otherwise, the checker investigates every possible case.

For generating a branch that represents this else-case behaviour, we need to keep in mind that an edge cannot be labelled with a concrete variable. If loc_i is of abstract sort, then we may simply use the abstract state variable loc_i as an edge label. However, if loc_i is of concrete sort, then it has to be substituted by its current value val_{loc_i}. In this case, we have to generate a graph that comprises branches for all possible evaluations for the state variable loc_i. Thus, each branch represents the formula $loc'_i = val_i \wedge loc_i = val_i$, which obviously specifies that the location keeps its value. Figure 5 depicts an MDG that represents the else-case for a concrete location that ranges over three values. This (sub-) graph is disjoined with the MDG that represents the location-update pair (without else-case).

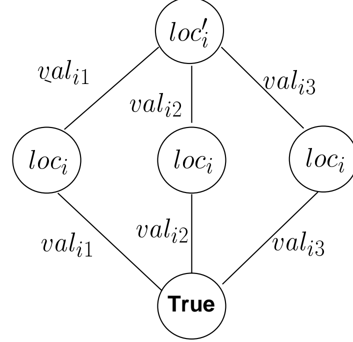

Fig. 5: The else-case for concrete locations

For generating the MDG G_{ij} that represents a guard, we use four basic functions (*and*, *or*, *negation*, and *equality*) as basic boolean operators that may appear in the guard (referred to as primary operators in the last section). Every other boolean operator can be mapped to these basic predicates.

- $and(opd_1, opd_2)$ is transformed into the **conjunct** of opd_1 and opd_2, assuming that both operands are given as MDGs.
- $or(opd_1, opd_2)$ is transformed into the **disjunct** of opd_1 and opd_2, if both operands are given as MDGs.
- $not(opd)$ is transformed into the **negation** of opd, if the operand is given as an MDG that does not contain node labels of abstract sort.
- $eq(lhs, rhs)$ assembles a new MDG. Its root is labelled by lhs, and it has a single edge labelled with rhs which leads to the leaf `True`.

Except for equations, the boolean operations take MDGs as operands. Thus, we recursively call the guard transformation function for the operands. The base of this recursion is the equality operator which operates on simple terms and constants and yields a simple graph.

Note that in most cases, the parameters of *eq* cannot be complex terms. Any parameter t_i of concrete sort will be simplified into a simple term $[\![t_i]\!]_\zeta$ during the first step of the transformation that is introduced in the last section. These simple terms can be treated as labels. If one parameter is of abstract sort, the equation is mapped into a cross-term $isEq(lhs, rhs)$ and does not cause problems either (cross-terms can be used as labels as well).

The only terms that are not simplified by $[\![.]\!]_\zeta$ are the primary operators for equality, conjunction and disjunction. If the operands of an equation are boolean expressions with non-simplified operators, we have to replace the equa-

tion $eq(lhs, rhs)$ with the expression $(lhs \wedge rhs) \vee (\neg lhs \wedge \neg rhs)$ which is logically equivalent[7].

Basic functions for conjoining and disjoining MDGs are given in the MDG-Package (see [Zho96]). Additionally, we implemented the algorithm for negation. This negation of an MDG is possible if no abstract variable appears as an *lhs* in any involved equations. As argued above, this is the case for guards $guard_{ij}$.

Our algorithm for negation of an MDG M assumes that the root is not labelled by a variable of abstract sort (otherwise an error will be output). It works inductively on the structure of MDGs:

- if $M = \texttt{true}$ then $not(\texttt{true}) = \texttt{false}$
- if $M = \bigvee_{j \in J}(loc = val_j \wedge G_j)$

  where $dom(loc) = \{val_i \mid i \in I\}$, and the index set I, and the index subsets J and $\bar{J}$ of occurring and non-occurring indices are such that $J \subseteq I, \bar{J} \subseteq I, J \cap \bar{J} = \emptyset$, and $J \cup \bar{J} = I$ then the negation of M can be reduced to

  $not(M) = \bigvee_{k \in \bar{J}}(loc = val_k \wedge \texttt{true}) \vee \bigvee_{j \in J}(loc = val_j \wedge not(G_j))$.

  Figure 6 shows the sketch of the corresponding graphs for $I = \{val_1, val_2, val_3\}$.

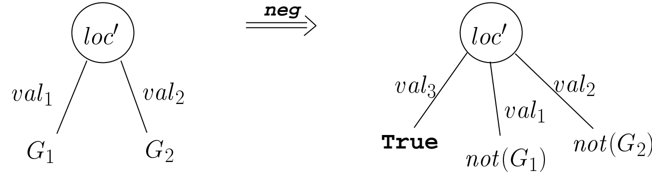

Fig. 6. Computing negation

The initial state of an ASM model is defined together with the declaration of functions and their domains. Every dynamic or external function may have a defined initial value. For our transformation we gather all initialisation information and put it into a formula that determines the set of initial states: $init_state = \bigwedge_i(loc_i = init_val_i)$. This conjunction need not be exhaustive since initialisation for some functions may be omitted. This formula is transformed into a simple MDG that consists of a single branch comprising all conjuncts. The domains of an ASM model are defined as enumerations or by set comprehensions in the case of concrete sorts. Set comprehensions are unfolded by our transformation algorithm into enumerated sorts.

Each location and each cross-term operator has an attached number in order to define a variable ordering. This ordering determines the shape of the corresponding MDG later on and heavily influences the graph size. We do not have a

[7] Since guard expressions do not contain abstract variables on a LHS of an equation, negation is applicable.

proper heuristic for a good ordering implemented yet[8]. Our implementation so far orders the locations and cross-term operators such that cross-term operators have a greater number than locations (in most cases the former depend on some of the latter and thus must have a greater ordering number). To support the user, we automatically generate a function which prints the list of locations and cross-term operators that are contributing to the current model. The order of this variable list can changed manually.

Given this interface from ASM-IL$^+$ to the MDG-Package, and with respect to certain limitations for applying abstraction (as discussed in the last section), we are able to represent ASM models with abstract sorts by means of MDGs. This MDG representation of our model is thus treatable by MDG-based checking algorithms.

6 Related Work

Uninterpreted functions are addressed elsewhere: In [BD94] data values and operations within the specification of the DLX architecture are modelled by means of uninterpreted functions. However, this approach allows only validity checking, no temporal properties can be checked. [CN94] introduce a new logic, called GTL, which also allows uninterpreted functions to be represented. The decidable fragment of GTL can be treated by an automatic validity checker (based on PVS). The thesis of Xu ([Xu99]) that introduces the logic $\mathcal{L}_{MDG}$ and the corresponding model checking algorithm based on MDGs shows that the decidable fragment of GTL is a subset of $\mathcal{L}_{MDG}$. Moreover, the MDG model checking approach goes beyond validity checking.

Due to the support for abstract sorts in MDGs, the computation of the abstract model appears to be much simpler than the mechanisms suggested in, e.g., [GS97] and [BPR00]. Instead of providing an abstraction function and proving that properties are preserved, we generate with less effort an abstract model that includes the intended model and more. This may result in *false negatives*, that is counter-examples that are not related to the particular instance of the model we want to check (in this case, it may be possible to add rewriting rules to exclude the non-intended interpretations), but if no counter-example can be found then *all* instances are correct.

Closer to our approach from the language point of view is the work in [Spi99]. It investigates automatic verification of ASM with unbounded input by representing an ASM model by means of a logic for computation graphs (called CGL*). The resulting formula is combined with a CTL*-like formula which specifies properties and checked by means of deciding its finite validity. This approach addresses the problem of checking systems with infinitely many inputs but it is only applicable to ASM with 0-ary dynamic functions and input that is

[8] There is some ongoing work at the University of Montreal investigating heuristics for a good variable ordering for MDGs. The results are not published yet.

restricted to relations. Spielmann proves that the decision procedure is PSPACE-complete and optimal. In his framework, the verification of generalised nullary programs which have functions in their input (instead of relations only) becomes undecidable.

7 Conclusion and Future Work

Multiway Decision Graphs (MDGs) are a graph structure for canonically representing transition systems that include abstract sorts and functions. A library of MDG functions is available to implement symbolic model checking algorithms for abstract transition systems. In this work, we introduced an interface from the ASM Workbench to MDGs. In order to exploit the expressibility of MDGs, we introduce a notion of abstract types into ASM-SL. This provides a simple means for generating abstract ASM models that can be represented by MDGs and model checked.

An implementation of the transformation algorithm is available. Our interface is tested only by using our own re-implementation of the reachability algorithm based on functions that are provided by the MDG library. A complete model checking tool for ASM based on MDGs is not available yet. However, we are planning to adapt the MDG-HDL model checker (see [Xu99,XCS$^+$98]) for our needs. This tool implements algorithms for model checking $\mathcal{L}_{MDG}$ formulas on hardware designs. $\mathcal{L}_{MDG}$ is the universal fragment of abstract CTL*, a derivate of CTL* that is tailored for MDGs (see [XCS$^+$98]).

We are still lacking experience with the applicability of our approach. Only the application to various case studies will show whether the suggested abstraction mechanism is too coarse or not to provide reasonable model checking results. If it is feasible, it provides a light-weight approach that is easy to use for checking infinite or too large ASM models.

References

[BCMD90] J. Burch, E. Clarke, K. McMillan, and D. Dill. Sequential circuits verification using symbolic model checking. In *Proc. of 27th ASM/IEEE Design Autiomation Conference*, 1990.

[BD94] J. Burch and D. Dill. Automatic verification of pipelined microprocessor control. In D.L. Dill, editor, *Proc. of Int. Conf. on Computer Aided Verification, CAV'94*, volume 818 of *LNCS*, pages 68–80. Springer-Verlag, 1994.

[BH99] R. Bharadwaj and C. Heitmeyer. Model checking complete requirements specifications using abstraction. *Automated Software Engineering*, 6(1):37–68, 1999.

[BPR00] T. Ball, A. Podelski, and S.K. Rajamani. Boolean and cartesian abstraction for model checking C programs. Technical Report MSR-TR-2000-115, Microsoft Research, 2000. To appear in TACAS'2001.

[Cas98] G. Del Castillo. The ASM Workbench: an Open and Extensible Environment for Abstract State Machines. In U. Glässer and P. Schmitt, editors, *Procs. of the 28th Annual Conference of the German Society of Computer Science*, TR, pages 139–155, Magdeburg University, 1998.

[Cas00] G. Del Castillo. *The ASM Workbench.* PhD thesis, Department of Mathematics and Computer Science, Paderborn University, 2000.

[CCL+97] E. Cerny, F. Corella, M. Langevin, X. Song, S. Tahar, and Z. Zhou. Automated verification with abstract state machines using Multiway Decision Graphs. In T. Kropf, editor, *Formal Hardware Verification: Methods and Systems in Comparison*, volume 1287 of *LNCS*, pages 79–113. Springer Verlag, 1997.

[CN94] D Cyrluk and P. Narendran. Ground temporal logic: A logic for hardware verification. In D. Dill, editor, *Proc. of Int. Conf. on Computer Aided Verification, CAV'94*, volume 818 of *LNCS*, pages 247–259. Springer-Verlag, 1994.

[CW00] G. Del Castillo and K. Winter. Model checking support for the ASM high-level language. In S. Graf and M. Schwartzbach, editors, *Proc. of 6th Int. Conference for Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2000*, volume 1785 of *LNCS*, pages 331–346. Springer-Verlag, 2000.

[CZS+97] F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny. Multiway Decision Graphs for automated hardware verification. *Formal Methods in System Design*, 10(1), 1997.

[Dur98] A. Durand. Modeling cache coherence protocol - a case study with FLASH. In U. Glässer and P. Schmitt, editors, *Proc. of the 28th Annual Conference of the German Society of Computer Science*, Technical Report, Magdeburg University, Germany, 1998.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. of Int. Conf. on Computer Aided Verification, CAV'97*, volume 1254 of *LNCS*, pages 72–83. Springer-Verlag, 1997.

[Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods.* Oxford University Press, 1995.

[MSC97] O. Ait Mohamed, X. Song, and E. Cerny. On the Non-Termination of MDG-based abstract state enumeration. In *Proc. of the IFIP Advanced Research Working Conference on Correct Hardware Design and Verification Methods (Charme'97)*, pages 218–235, 1997.

[Spi99] M. Spielmann. Automatic verification of abstract state machines. In N. Halbwachs and D. Peled, editors, *Proc. of Int. Conf. on Computer Aided Verification, CAV '99*, volume 1633 of *LNCS*, pages 431–442. Springer-Verlag, 1999.

[SS99] H. Saïdi and N. Shankar. Abstract and model check while you prove. In N. Halbwachs and D. Peled, editors, *Proc. of Int. Conf. on Computer Aided Verification, CAV'99*, volume 1633 of *LNCS*, pages 443–453. Springer-Verlag, 1999.

[Win97] K. Winter. Model Checking for Abstract State Machines. *J.UCS Journal for Universal Computer Science (special issue)*, 3(5):689–702, 1997.

[Win00] K. Winter. Towards a Methodology for Model Checking ASM: Lessons learned from the FLASH Case Study. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines - Theory and Applications, Int. Workshop ASM'2000, Selected papers*, volume 1912 of *LNCS*, pages 341–360. Springer-Verlag, 2000.

[Win01] K. Winter. *Model Checking Abstract State Machines.* PhD thesis, Technical University of Berlin, Germany, http://edocs.tu-berlin.de/diss/2001/winter_kirsten.htm, 2001.

[XCS+98] Y. Xu, E. Cerny, X. Song, F. Corella, and O. Ait Mohamed. Model checking for a first-order temporal logic using Multiway Decision Graphs. In *Proc. of Int. Conf. on Computer Aided Verification (CAV'98)*, volume 1427 of *LNCS*, pages 219–231. Springer-Verlag, 1998.

[Xu99] Y. Xu. *Model Checking for a First-order Temporal Logic Using Multiway Decision Graphs.* PhD thesis, University of Montreal, 1999.

[Zho96] Z. Zhou. *MDG Tools (V1.0) Developer's Manual.* IRO University of Montreal, Montreal, Canada, June 1996.

[ZSC+95] Z. Zhou, X. Song, F. Corella, E. Cerny, and M. Langevin. Partitioning transition relations automatically and efficiently. In *IEEE Proceedings of Fifth Great Lakes Symposium on VLSI (GLSVLSI'95)*, March 1995.

[ZST+96] Z. Zhou, X. Song, S. Tahar, F. Corella E. Cerny, and M. Langevin. Verification of the island tunnel controller using Multiway Decision Graphs. In M. Srivas and A. Camilleri, editors, *Proc. of Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *LNCS*, pages 233–246. Springer-Verlag, 1996.