Model Checking Support for the ASM High-Level Language

Giuseppe Del Castillo[1]* and Kirsten Winter[2]

[1] Heinz Nixdorf Institut, Universität-GH Paderborn
Fürstenallee 11, D-33102 Paderborn, Germany
giusp@uni-paderborn.de
[2] GMD FIRST
Kekuléstr.7, D-12489 Berlin, Germany
kirsten@first.gmd.de

Abstract Gurevich's Abstract State Machines (ASM) constitute a high-level specification language for a wide range of applications. The existing tool support for ASM–currently including type-checking, simulation and debugging–should be extended to support computer-aided verification, in particular by model checking. In this paper we introduce an interface from our existing tool environment to the model checker SMV, based on a transformation which maps a large subset of ASM into the SMV language. Through a case study we show how the proposed approach can ease the validation process.

1 Introduction

Gurevich's Abstract State Machines (ASM) [7] constitute a simple but powerful method for specifying and modeling software and hardware systems. Existing case studies include specifications of distributed protocols, architectures, embedded systems, programming languages, etc. (see [1] and [8]).

The advantage of ASMs is in the simple language and its intuitive understanding. The method is based on general mathematics which allows to naturally model systems on a suitable level of abstraction. Traditionally, the verification task is done by means of hand-written mathematical proofs. Tool support for the verification process is obviously needed for a broader acceptance.

Our contribution to this task is the development of an interface between the *ASM Workbench* [2] and the SMV model checker [11]. The ASM Workbench is a tool environment, based on a typed version of ASM, which includes a type checker and a simulator for ASMs. SMV has been chosen as a typical representative of a class of model checkers based on transition systems and could be easily replaced by any other similar model checker, e.g., SVE [5] or VIS [6].

On the other hand our transformation tool supplies SMV with a higher level modeling language, namely ASMs. This facilitates the specification task by allowing the use of more complex data types and of *n-ary dynamic functions* for

* Partially supported by the DFG Schwerpunktprogramm "Softwarespezifikation".

S. Graf and M. Schwartzbach (Eds.): TACAS/ETAPS 2000, LNCS 1785, pp. 331–346, 2000.
© Springer-Verlag Berlin Heidelberg 2000

parameterization (a peculiar feature of the ASM language, which generalizes the classical notion of state variables).

Since model checking is only applicable to finite-state systems, we have to put restrictions on the ASM model to be checked in order to make it finite: all function ranges have to be restricted to a fixed finite set of values. To cope with a broader subset of the ASM language, we extend the basic work of [14], which introduced a simple transformation schema, to support the transformation of *n-ary dynamic functions* for $n > 0$. To ease the transition from infinite or large models to finite and feasible ones, we introduce a language feature for adjusting the function ranges in the declaration part of the system specification. Thus, such changes can be done locally and are not spread over the whole model.

From a methodical point of view, model checking can *support the early design phase*: checking properties of the system behavior may yield counterexamples which help to "debug" the system specification. The simulator provided by the ASM Workbench can be fed with the counterexamples in order to illustrate the erroneous behavior. After locating and correcting the error that causes the counterexample, the transformation and model checking should be repeated. This debugging process gives a deeper insight into the model at hand. Errors become visible that can be easily over seen when carrying out mathematical proofs which are not mechanically checked, borderline cases become visible that are mostly not found when simulating isolated test cases.

We are not claiming that model checking can replace, in general, mathematical proofs (developed with or without the help of theorem provers), as the range of applicability of model checking techniques is restricted to the verification of finite instances of the problem at hand and is in most cases insufficient to prove correctness of a system or protocol in general. However, we argue that using tool support in the way we suggest helps to find errors with small additional effort.

This paper is structured as follows: after introducing the main features of ASM (Sect. 2), we show how the transformation from ASM into the SMV language is performed (Sect. 3). Sect. 4 presents results from applying our approach to a case study, an ASM specification of the FLASH cache coherence protocol. Sect. 5 outlines related work. We conclude in Sect. 6 with an outlook to further possible improvements of our tool.

2 Basic Notions of Abstract State Machines

In this section we introduce some basic notions of ASM (see [7] 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:

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

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 $T_1 \ldots T_n \to T$, where the T_i and T are basic types (written $f : T_1 \ldots T_n \to T$, or simply $f : T$ if $n = 0$). for each function name $f : T_1 \ldots T_n \to T$ 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) *true* : *BOOL*, *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 $f : T_1 \ldots T_n \to T$ is a dynamic or external function name, we call a pair $l = (f, \overline{x})$ with $\overline{x} \in \mathcal{T}_1 \times \ldots \times \mathcal{T}_n$ 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.[1]

[1] See, for instance, the example in Sect. 4, containing a rule for each message type.

2.2 The ASM Language

Terms Terms are defined as in first-order logic: *(i)* if $f : T_1 \dots T_n \to T$ is a function name in Σ, and t_i are terms of type T_i (for $i = 1, \dots, n$), then $f(t_1, \dots, t_n)$ 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:[2]

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

As opposed to first-order logic, there is no notion of formula: boolean terms are used instead. Finite quantifications of the form "$(Q\ v$ `in` $A : G)$", where Q is $\forall$ or $\exists$, $v : T$, $A : SET(T)$, and $G : BOOL$, are also valid boolean terms.[3]

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:

$$S_0 \xrightarrow{\Delta_{S_0}(P)} S_1 \xrightarrow{\Delta_{S_1}(P)} S_2 \;\dots\; \xrightarrow{\Delta_{S_{n-1}}(P)} S_n \;\dots$$

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).[4]

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

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`". The semantics of transition rules is as follows:

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

[2] 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 $\rho \backslash v$ 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, $S(t) = S_\emptyset(t)$).

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

[4] 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.

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

Note that executing a block (or a do-forall) rule corresponds to *simultaneous* execution of its subrules[5] 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(f_x, y, f_z(y)) := f_x + y + f_z(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 [7]). 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. (The FLASH model presented in Sect. 4 is an example of this style of modelling, except that all agents execute exactly the same program, but on different data.)

The ASM-SL Notation The ASM language, including all constructs above, is supported by the "ASM Workbench" tool environment [2], which provides syntax- and type-checking of ASM specifications as well as their simulation and debugging. The source language for the ASM Workbench, called ASM-SL, includes some additional features which are necessary for practical modelling tasks: constructs for defining types, functions, and named transition rules ("macros"), as well as a set of predefined data types (booleans, integers, tuples, lists, finite sets, etc.): as the ASM-SL notation is quite close to usual mathematical notation, no further explanation of ASM-SL will be needed.

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

3 Translating Abstract State Machines into SMV

In this section, after a brief comparison of the ASM and SMV specification languages, we describe the transformation from ASM to SMV in two stages. First we recall the translation scheme introduced in [14], defined for a subset of ASM called ASM_0 in this paper (Sect. 3.1). Then we define a transformation technique to reduce any ASM specification to ASM_0, such that the first translation scheme can then be applied (Sect. 3.2).

ASM versus SMV While the computational model underlying both SMV and ASM is essentially the well-known model of *transition systems*, there are some significant differences: **(1.)** Abstract State Machines define, in general, systems with a possibly infinite number of states (as both the number of locations and the location ranges may be infinite); **(2.)** the way of specifying transitions in ASM and SMV is different: in SMV transitions are specified by **next**-expressions, which completely define the value which a state variable assumes in the next state, while updates of dynamic functions in ASM may be scattered troughout the program; **(3.)** the ASM notions of *dynamic function* and *external function* generalize the notion of *state variable* typical of basic transition systems (state variables correspond to nullary dynamic/external functions of ASM).

The first issue is solved by introducing *finiteness constraints*, the second and third are addressed by the transformations of Sect. 3.1 and 3.2, respectively.

Finiteness constraints In order to ensure that the ASM programs to be translated into SMV define finite-state systems, the user has to specify, for each dynamic or external function $f : T_1 \dots T_n \to T$, a finiteness constraint of the form $f(x_1, \dots, x_n) \in t[x_1, \dots, x_n]$, where $t : SET(T)$ is a term denoting a finite set, possibly depending on the arguments of f (see Fig. 1 for an example). For external functions, finiteness constraints correspond to environment assumptions, expressed in the resulting SMV model by the range of the generated state variables; for dynamic functions, it must be checked that the constraints are not violated by the rules, resulting in the SMV code in appropriate proof obligations, which we call *range conditions*.[6]

3.1 The Basic Translation Scheme

The translation scheme introduced in [14] can be applied to transform into SMV a subset ASM_0 of ASM, where: *(i)* only **nullary** dynamic and external functions are allowed; *(ii)* the only available data types are integers, booleans and enumerated types; *(iii)* the only defined static functions are those corresponding to predefined operations in SMV (boolean operations, +, -, etc.).

As the semantic models for ASM_0 are essentially basic transition systems, the translation of ASM into SMV is very close:

[6] Note, however, that the range conditions can often be discarded by a simple static analysis of the rules, which prevents their expensive proof by model-checking.

- non-static functions (i.e., dynamic and external functions) are identified with locations and thus mapped one-to-one to SMV state variables;
- values of the ASM data types are mapped one-to-one to SMV constants;
- applications of static functions are translated to applications of the corresponding built-in operators of SMV.

What remains to be done is to restructure the ASM program into a form where updates of the same location, together with their guards, are collected together. This is done in two steps. First, we transform an ASM program P into an equivalent ASM program P' consisting only of a block of guarded updates (i.e., rules of the form `if` G `then` $f(\overline{t}) := t$) by means of a "flattening" transformation:

$[\![\texttt{skip}]\!] = \textit{(empty block)}$
$[\![f(\overline{t}) := t]\!] = \texttt{if } \textit{true} \texttt{ then } f(\overline{t}) := t$
$[\![R_1 \ldots R_n]\!] = [\![R_1]\!] \ldots [\![R_n]\!]$

$$[\![R_T]\!] = \begin{cases} \texttt{if } G_T^1 \texttt{ then } R_T^1 \\ \ldots \\ \texttt{if } G_T^n \texttt{ then } R_T^n \end{cases} \quad [\![R_F]\!] = \begin{cases} \texttt{if } G_F^1 \texttt{ then } R_F^1 \\ \ldots \\ \texttt{if } G_F^m \texttt{ then } R_F^m \end{cases} \Rightarrow [\![\texttt{if } G \texttt{ then } R_T \texttt{ else } R_F]\!] = \begin{cases} \texttt{if } G \wedge G_T^1 \texttt{ then } R_T^1 \\ \ldots \\ \texttt{if } G \wedge G_T^n \texttt{ then } R_T^n \\ \texttt{if } \neg G \wedge G_F^1 \texttt{ then } R_F^1 \\ \ldots \\ \texttt{if } \neg G \wedge G_F^m \texttt{ then } R_F^m \end{cases}$$

Second, we collect all guarded updates of the same location, thus obtaining, for each location *loc* occurring on the left-hand side of an update in P', a pair $(loc, \{(G_1, t_1), \ldots, (G_n, t_n)\})$ which maps *loc* to a set of pairs (guard, right-hand side). Such a pair is translated into the following SMV assignment:[7]

```
ASSIGN next(C⟦loc⟧) :=
    case C⟦G1⟧ : C⟦t1⟧ ;  ...  C⟦Gn⟧ : C⟦tn⟧ ;  1 : C⟦loc⟧ esac;
```

where $\mathcal{C}[\![.]\!]$ denotes here the ASM $\rightarrow$ SMV compiling function for terms, which is straightforward for ASM_0. For each location l of a dynamic function f, in addition to the `next` assignment above, the transformation also generates the location's initialization (an `init` assignment in SMV) as well as two proof obligations, a *range condition* (see discussion of finiteness constraints above) and a *no-conflict condition*, which ensures that no conflicts arise on this location. In fact, due to the semantics of `case` in SMV, the translation scheme is correct only if for all i, j with $i \neq j$, $S \models \neg(G_i \wedge G_j) \vee (t_i = t_j)$ in any reachable state S: if, in some state S, this condition is not satisfied, the ASM transition produces a conflict (i.e., an error), while the corresponding SMV transition simply picks one of the updates (the first one in the `case` whose guard is satisfied).[8]

3.2 The Extended Translation Scheme

In this section we show how to reduce an arbitrary (finite-state) ASM to ASM_0. This transformation allows to deal with the complete ASM language as in [7],

[7] Note that we have to specify the default case explicitly (if none of the guards is true) which is given implicitly in ASM rules (see ASM semantics above).

[8] Like range conditions, no-conflict conditions can be often discarded statically.

with the exception of `import` rules (rules which allow the dynamic creation of elements at run-time) and `choose` rules. (However, one can deal with `choose` as explained in Sect. 2.2.) Arbitrary data types and operations (in particular, lists, finite sets, finite maps and user-definable freely generated types, as provided by ASM-SL) can be used without any restriction. Finite quantifications are also supported.

The main problem here, as opposed to ASM_0, is that in general we do not know which location is updated by an update rule $f(t_1, \ldots, t_n) := t$ (if $n > 0$): the updated location may differ from state to state if some t_i contains non-static function names. However, if all terms t_i contain only static function names, they can be evaluated statically to values x_i, and the term $f(t_1, \ldots, t_n)$ to the location $l = (f, \overline{x})$. Thus, the basic idea of the transformation is to iteratively unfold and simplify rules until all terms can be reduced to values or locations.

To formally define the transformation, we extend the syntactic category of terms to "partially evaluated terms" (simply called "terms" in the sequel) by adding values and locations:

$$t \ ::= \ f(t_1, \ldots, t_n) \ \mid \ v \ \mid \ (Q \ v \ \texttt{in} \ A : G) \ \mid \ x \ \mid \ l$$

(We adopt the convention that x stands for a value and l for a location).

Terms can be simplified by means of the transformation $[\![.]\!]_\rho$ defined in Table 1, which is then extended to rules in a canonical way. Note that, whenever ρ contains bindings for all free variables occurring in t: *(i)* if t is a static term, then $[\![t]\!]_\rho$ is a value x (coinciding with $S_\rho(t)$ in every state S); *(ii)* if $t \equiv f(t_1, \ldots, t_n)$ is a term where f is a dynamic or external function name and all the subterms t_i are static (we call such a term a *locational* term), then $[\![t]\!]_\rho$ is a location l.[9]

The rule-unfolding transformation $\mathcal{E}$, which operates on closed rules such as the program P, is formally defined in Table 2. It works as follows:

- if the rule R consists of a block of update rules of the form *location* := *value*, it terminates and yields R as result (there is nothing left to unfold);
- otherwise, it looks for the first location l occurring in R (but not as left-hand side of some update rule) and unfolds R according to the possible values[10] of l. In turn, the unfolding has to be applied to the subrules $[\![R[l/x_i]]\!]$ obtained by substituting the values x_i for l in R and simplifying.

Applying $\mathcal{E}$ to the (simplified) ASM program $[\![P]\!]_\emptyset$ yields a program $P' = \mathcal{E}([\![P]\!]_\emptyset)$ which is essentially an ASM_0 program (formally, the locations have still to be replaced by nullary dynamic or external function names and the values by nullary static function names, i.e. by constants).[11]

[9] A simple consequence of this fact is that every closed static term simplifies to a value and every closed locational term to a location.

[10] The finite range of location $l = (f, \overline{x})$ is derived from the finiteness constraint for f.

[11] The unfolding transformation often results in very large decision trees (`case`-structures in SMV): however, this does not have a negative influence on the efficiency of verification with SMV, as the verification costs depend on the size of the BDDs representing the transition relation and not on the size of the SMV source code (and BDDs, for a given variable ordering, are a canonical representation).

Table 1. Term and Rule Simplification

Term Simplification

$[\![x]\!]_\rho = x \qquad [\![\, l \,]\!]_\rho = l$

$$[\![\, v \,]\!]_\rho = \begin{cases} x = \rho(v) & \text{if } v \in \mathrm{dom}(\rho) \\ v & \text{otherwise} \end{cases}$$

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

$$[\![f(t_1, \ldots, t_n)]\!]_\rho = \begin{cases} x = \mathbf{f}(x_1, \ldots, x_n) & \text{if } f \text{ static function name} \\ l = (f, (x_1, \ldots, x_n)) & \text{if } f \text{ dynamic/external function name} \end{cases}$$

$[\![t_i]\!]_\rho = l$ or $[\![t_i]\!]_\rho = f'(\overline{t}')$ for some $i \in \{1, \ldots, n\} \Rightarrow$
$[\![f(t_1, \ldots, t_n)]\!]_\rho = f([\![t_1]\!]_\rho, \ldots, [\![t_n]\!]_\rho)$

$$[\![(Q\ v\ \texttt{in}\ A : G)]\!]_\rho = \begin{cases} [\![G]\!]_{\rho[v \mapsto x_1]}\ op\ \ldots\ op\ [\![G]\!]_{\rho[v \mapsto x_n]} \\ \quad \text{if } [\![A]\!]_\rho = x = \{x_1, \ldots, x_n\} \quad \text{(i.e., if } [\![A]\!]_\rho \text{ is a value)} \\ (Q\ v\ \texttt{in}\ [\![A]\!]_\rho : [\![G]\!]_{(\rho \setminus v)}) \\ \quad \text{otherwise.} \end{cases}$$

(where $op \equiv \wedge$ if $Q \equiv$ `forall`, $op \equiv \vee$ if $Q \equiv$ `exists`).

Rule Simplification

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

$[\![t_L := t_R]\!]_\rho = [\![t_L]\!]_\rho := [\![t_R]\!]_\rho$

$[\![R_1 \ldots R_n]\!]_\rho = [\![R_1]\!]_\rho \ldots [\![R_n]\!]_\rho$

$$[\![\texttt{if}\ G\ \texttt{then}\ R_T\ \texttt{else}\ R_F]\!]_\rho = \begin{cases} [\![R_T]\!]_\rho & \text{if } [\![G]\!]_\rho = \mathbf{true} \\ [\![R_F]\!]_\rho & \text{if } [\![G]\!]_\rho = \mathbf{false} \\ \texttt{if}\ [\![G]\!]_\rho\ \texttt{then}\ [\![R_T]\!]_\rho\ \texttt{else}\ [\![R_F]\!]_\rho & \text{otherwise.} \end{cases}$$

$$[\![\texttt{do forall}\ v\ \texttt{in}\ A\ \texttt{with}\ G\ \ R']\!]_\rho = \\ = \begin{cases} [\![\texttt{if}\ G\ \texttt{then}\ R']\!]_{\rho[v \mapsto x_1]}\ \ \ldots\ \ [\![\texttt{if}\ G\ \texttt{then}\ R']\!]_{\rho[v \mapsto x_n]} \\ \quad \text{if } [\![A]\!]_\rho = x = \{x_1, \ldots, x_n\} \quad \text{(i.e., if } [\![A]\!]_\rho \text{ is a value)} \\ \texttt{do forall}\ v\ \texttt{in}\ [\![A]\!]_\rho\ \texttt{with}\ [\![G]\!]_{(\rho \setminus v)}\ \ [\![R']\!]_{(\rho \setminus v)} \\ \quad \text{otherwise.} \end{cases}$$

Table 2. Rule Unfolding

Rule Unfolding

If R has the form $l_1 := x_1 \ldots l_n := x_n$, then $\mathcal{E}(R) = R$.
Otherwise:

$\mathcal{E}(R) =$ `if` $l = x_1$ `then` $\mathcal{E}([\![R[l/x_1]]\!]_\emptyset)$
`else if` $l = x_2$ `then` $\mathcal{E}([\![R[l/x_1]]\!]_\emptyset)$
$\ldots$
`else if` $l = x_n$ `then` $\mathcal{E}([\![R[l/x_n]]\!]_\emptyset)$

where l is the first location occurring in R (but not as lhs of an update rule) and $\{x_1, \ldots, x_n\}$ is the range of location l.

Fig. 1 illustrates graphically the transformation technique (for simplicity, we consider a rule without variables, such that we can omit mentioning environments). The root of the tree–enclosed in the dashed box–is the (simplified) ASM program $[\![P]\!]$ to be transformed. The successors of each node in the tree are obtained as result of an unfolding step (under the given finiteness constraints): for instance, the successors of the root node are the rules $[\![[\![P]\!][a/1]]\!]$, $[\![[\![P]\!][a/2]]\!]$, and $[\![[\![P]\!][a/3]]\!]$, respectively. Locations are emphasized by enclosing them in boxes: note that, at the leaves, locations occur only as left-hand side of updates, thus they cause no further unfolding. The dashed box on the right contains the ASM_0 program produced by the transformation: note that the locations actually affected by the ASM program–which are revealed by the unfolding–are mapped to nullary functions ("state variables"), whose ranges are derived from the finiteness constraints (see box at the top right corner).

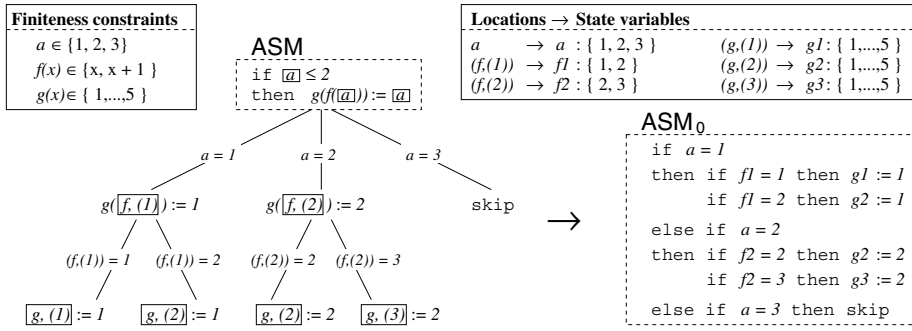

Fig. 1. Rule Transformation Example

4 Case Study: The FLASH Cache Coherence Protocol

As an example for our model checking approach we chose a formalization of the FLASH protocol [9] with ASM. Our model is based on the work of Durand [4]. In Sect. 4.1, after a short introduction to FLASH, we describe an ASM model derived from [4] and motivate our refinements. Then we sketch the debugging process supported by the transformation and checking with SMV (in Sect. 4.2).

4.1 FLASH Cache Coherence Protocol

The Stanford FLASH multiprocessor integrates support for cache coherent shared memory for a large number of interconnected processing nodes. Each line-sized block of the distributed memory is associated with a *home* node containing the part of the physical memory where that line resides. Every read or write miss concerning a remote memory line triggers a line request to its *home* node that in turn initiates the corresponding part of the protocol. The request

may ask for *shared* or *exclusive* access depending on whether reading or writing access is wanted.

The ASM description of the protocol is based on agents. A set of transition rules describes the behavior of a single agent. The behavior is determined by the currently processed message type – a notion that yields the clear model structure that is sketched in Fig. 2.

Message Structure. A message is modeled as a quintuple consisting of the type of the message, the addressed agent, sender agent, agent initiating the request and requested line[12]. Message types related to **shared** access are:

`get:`	requesting a line from its *home*
`put:`	granting a line to the requester (*source* of the request)
`fwdget:`	forwarding the request to an exclusive owner of the line
`swb:`	requesting a write-back of an owned line that is to be shared
`nack, nackc:`	negatively acknowledging the request or forwarded request respectively, if it cannot be performed now.

In analogy, message types related to **exclusive** access are:

`getx`, `putx`, `fwdgetx`, and also

`inv:`	requesting a current *sharer* of the line to invalidate its local copy
`invAck:`	acknowledging the invalidation of the line
`fwdAck:`	owner's granting according to a forwarded shared request.

Additionally, for releasing a shared or exclusive copy from its cache an agent sends a write_back (`wb`) and a replace message (`rpl`) to *home*, respectively. A read or write miss of a line, or the end of accessing, is simulated with the help of an oracle function which non-deterministically triggers an agent to send `get`/`getx` or `rpl`/`wb` messages.

State Functions. Besides the message type, the agent's behavior depends on several local state variables: *curPhase(line)* (phase of the current request), *State(line)* (state of the local line copy in use), and *pending(line)* (flag for currently processed request). *Owner(line)* and the set of *Sharers* of a line are also taken into account.

Adjustable Parameters. The transition rules are parameterized by *self*, the agent that is currently active (this is implicit in Fig. 2), and the requested line. The domains of these parameters, *Agents* and *Lines*, and their extent are easily adjustable in the declaration part of the specification.

Necessary Refinements. Sending a message is given as a macro definition. In the abstract model of [4] *SendMsg* adds a message to a (possibly infinite) set of messages in transit. The strategy for receiving a message from this set is not specified. For the proof it is just assumed that the messages are received in the right order. In order to keep the model finite and to formalize the assumption on the model behavior we have to refine the model. We replace the set of messages in transit by a finite queue for each agent, and we extend the overall behavior by means of a sub-step for synchronization. In the synchronization step the messages are passed through to the addressed agent in the proper order.

[12] In our adaptation of the model the parts related to data are discarded.

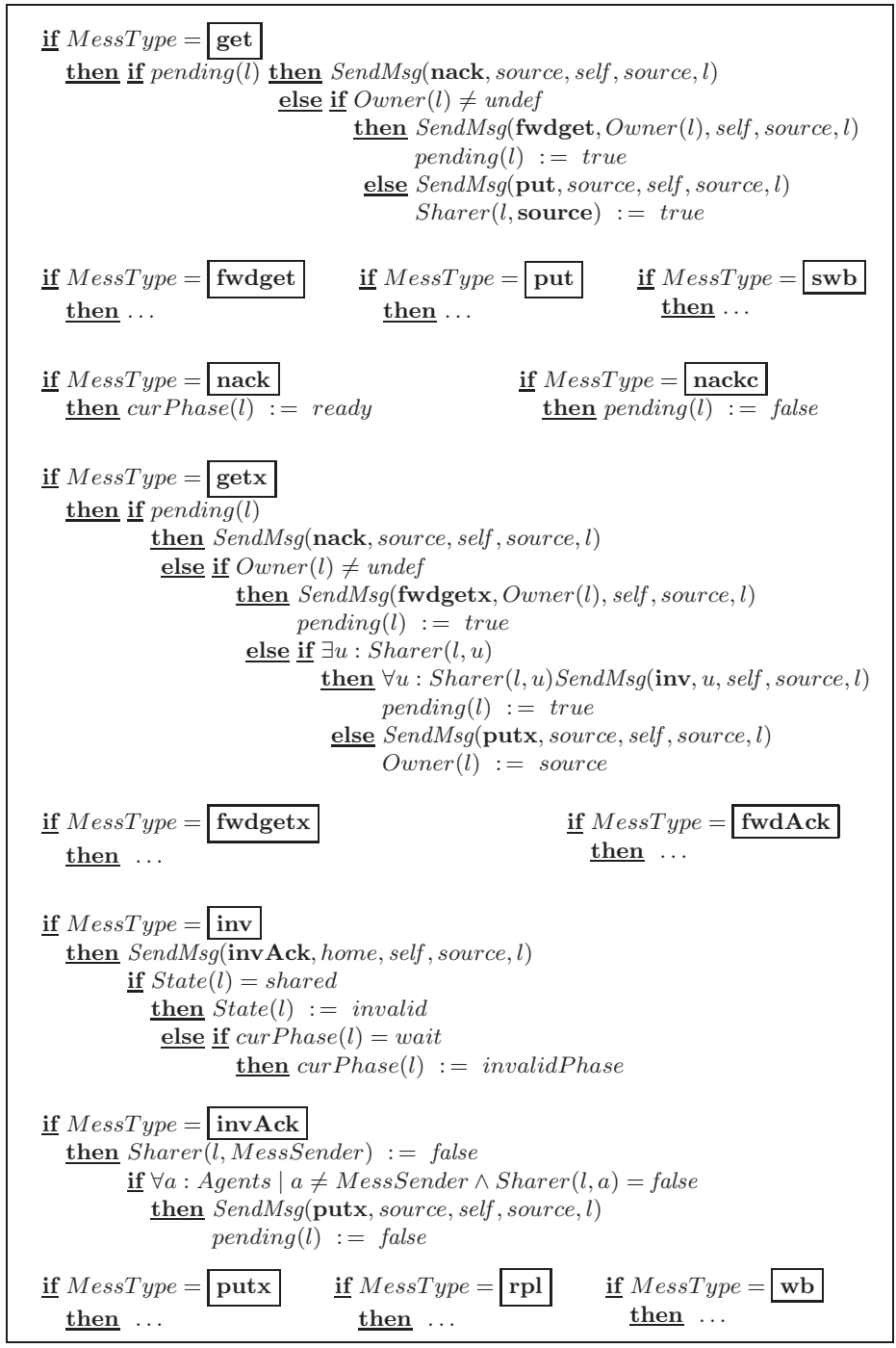

Fig. 2. Agent behavior modeled by ASM transition rules

In an ASM model, introducing a sub-step is structure preserving: in addition to the ASM for message computation (explained above) we specify an ASM for the message passing through. An overall ASM invokes both "sub-ASMs" in turn. Taking this, we benefit from the clear and understandable structure of the abstract model. The entire refined ASM-model is available on the web at `http://www.first.gmd.de/~kirsten/publications/flash_param.asm`.

4.2 Model Checking the Transformed System Specification

We take model checking of the transformed ASM model as an evolutionary process of debugging: we edit the ASM model, transform it automatically into an SMV model, run SMV to check the properties under investigation, investigate the resulting counterexample (if any) within the ASM model, and debug the ASM model. Since there are no restrictions on the behavior of the environment (producing requests on a line), we do not suffer from "wrong" counterexamples that are not suitable for debugging the ordinary system behavior. (We call counterexamples wrong, if they are caused by non-reasonable environment behavior that should be excluded. They obstruct the debugging process, since only one counterexample will be produced.)

As the debugging process is more efficient if the model checking terminates in a reasonable span of time, we keep our model as small as possible. We find that, even when the model is restricted to few agents and lines, we detect errors in the abstract model as well as in our refinement. In the following we describe two of them as examples. We check the model for **safety** and **liveness**, i.e.:

- No two agents have exclusive access on the same line simultaneously.
- Each request will eventually be acknowledged.
- Whenever an agent gets shared access, home will note it as a sharer.

We formalize these requirements in CTL, e.g.[13]:

$\bigwedge_{i \neq j}$[`AG (!(State(a(i),l)=exclusive & State(a(j),l)=exclusive))`]

$\bigwedge_{i}$[`AG (curPhase(a(i),l) = wait -> AF (curPhase(a(i),l) = ready))`]

$\bigwedge_{i}$[`AG (State(a(j),l)=shared -> AX (Sharer(l,a(i)) = true))`]

Our first counterexample shows simultaneous exclusive access (for reasons of space we have to omit the listing here). The error that caused the counterexample can also be found in the abstract ASM model of [4]:

> Whenever a `putx`-message is sent to grant exclusive access the addressed requester has to be noted as owner of the line. This is specified in the `getx`-rule but it is missing in the `invAck`-rule that might also cause a `putx`-message to be send (see also Fig. 2). The protocol is unsafe since simultaneous exclusive access may occur, and written data may be lost.

[13] Though the third specification is rather weak, it yields helpful counterexamples.

The following counterexamples are dedicated to the problem of racing (i.e., conflicts) on the finite message queue. Although our data space is limited to a very short queue, we can derive more general remarks, e.g.:

> Each sharer of a requested line has to process the rule for invalidation (`inv`-rule). It sends an `invAck`-message to *home* for acknowledging the invalidation. When receiving an `invAck`-message, *home* deletes the sender from the list of sharers. If *home* is sharer too,[14] a deadlock may occur if the number of sharers is greater or equal than the length of the message queue: *home* may fail to complete with the `inv`-rule when the queue is full and sending a message is not possible (since every other sharer may have sent before); *home* stays busy and can not process the incoming `invAck`-rule to clear the queue. In general, we found out that the message queue must be larger or equal than the number of agents since in the worst case each agent is a sharer and will send simultaneously an `invAck`-message to the home node.

The examples show that helpful borderline cases can be detected more easily by a model checker than by pure simulation. The computational effort for the automated transformation of our models ranges from three to five seconds. The size of the resulting SMV models is given below.[15] The variable ordering is determined by the automatic reordering facility that is given by the SMV.

resources used:	2 agents, 1 line	3 agents, 1 line	2 agents, 2 lines
user time/system time:	4.69 s/0.13 s	5687.52 s/0.6 s	17263.2 s/0.86 s
BDD nodes allocated:	70587	1612740	2975127
Bytes allocated:	4849664	37748736	54657024
BDD nodes repr. transition relation:	19261 + 78	288986 + 82	78365 + 96

Although checking our model of the FLASH protocol is only feasible for a small number of agents and lines, the results show that the counterexamples yield extremely helpful scenarios for locating errors.

5 Related Work

Extending tool environments for high-level specification languages with an interface to a model checker is an upcoming topic. One can find approaches that are quite similar to ours but work on a different language: [3] suggests a transformation from Statecharts into SMV, in [10] Controller Specification (CSL) models are transformed and model checked by SVE, [12] equips the multi-language environment SYNCHRONIE with an interface to the VIS model checker, etc.

Closer to our approach from the language point of view, [13] also investigates automatic verification of ASM. Spielmann represents an ASM model independently of its possible input by means of a logic for computation graphs (called

[14] This is possible if we allow intra-node communication.

[15] The experiments were carried out on an UltraSPARC-II station with 296MHz and 2048 Mb memory, the operating system is Solaris 2.6.

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 only 0-ary dynamic functions (i.e. ASM_0 programs) and relational input, which is the second result of [13].

6 Conclusions

We presented an interface from the ASM Workbench to SMV, based on a transformation from ASM to the SMV language extending the one defined in [14] by the treatment of dynamic functions of arity $n > 0$. This is essential, as most ASM specifications benefit from the abundant use of parametric dynamic functions.

The practicability of our approach is demonstrated by a non-trivial case study: the ASM model of the FLASH protocol. By example we show that errors can be found in the ASM model that will hardly be detected by pure mathematical proofs, and deduce more general constraints for the model at hand from the counterexamples.

We support the exploitation of the model checking facility by means of introducing *finiteness constraints* into the ASM specification language for easy control of the function ranges in order to restrict the state space of the model. Additionally, the developer benefits from the automatically generated proof obligations to be checked by SMV: the *no-conflict* conditions and the *range* conditions.

Some improvements of our tool, which are still to be implemented in order to make the transition between ASM and SMV smoother and thus ease the validation process, include the automatic translation of the counterexamples into a form which can be immediately read and simulated by the Workbench and the embedding of CTL operators into the ASM-SL language.

References

1. E. Börger. *Specification and Validation Methods*. Oxford University Press, 1995. 331
2. G. Del Castillo. Towards comprehensive tool support for Abstract State Machines: The ASM Workbench tool environment and architecture. In D. Hutter et al., eds., *Applied Formal Methods – FM-Trends 98*, LNCS 1641, pp. 311–325. Springer, 1999. 331, 335
3. N. Day. A model checker for Statecharts (linking case tools with formal methods). TR 93-35, CS Dept., Univ. of British Columbia, Vancouver, B.C., Canada, 1993. 344
4. A. Durand. Modeling cache coherence protocol - a case study with FLASH. In U. Glässer and P. Schmitt, editors, *Procs. of the 5th International ASM Workshop*, pages 111–126, Magdeburg University, 1998. 340, 341, 343
5. T. Filkorn et. al. *SVE Users' Guide*. Siemens AG, München, 1996. 331
6. The VIS Group. Vis: A system for verification and synthesis. In T. Henzinger and R. Alur, editors, *8th Int. Conf. on Computer Aided Verification, CAV'96*, number 1102 in LNCS, pages 428–432, July 1996. 331

7. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995. 331, 332, 335, 337
8. J.K. Huggins. Abstract State Machines home page. EECS Department, University of Michigan. `http://www.eecs.umich.edu/gasm/`. 331
9. J. Kuskin et al. The Stanford FLASH multiprocessor. In *21th Int. Symp. on Computer Architecture*. Chicago, IL, 1994. 340
10. P. Liggesmeyer and M. Rothfelder. Towards automated proof of fail-safe behavior. In W. Ehrenberger, editor, *Computer Safety, Reliablity and Security, SAFECOMP'98*, LNCS 1516, pages 169–184, 1998. 344
11. K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993. 331
12. A. Merceron, M. Müllerburg, and G.M. Pinna. Verifying a time-triggered protocol in a multi-language environment. In W. Ehrenberger, editor, *Computer Safety, Reliablity and Security, SAFECOMP'98*, LNCS 1516, pages 185–195, 1998. 344
13. M. Spielmann. Automatic verification of Abstract State Machines. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification, CAV '99*, number 1633 in LNCS, pages 431–442, Trento, Italy, 1999. 344, 345
14. K. Winter. Model checking for abstract state machines. *J.UCS Journal for Universal Computer Science (special issue)*, 3(5):689–702, 1997. 332, 336, 345