Automatic Review of Abstract State Machines by Meta-Property Verification*

Paolo Arcaini
University of Milan
paolo.arcaini@unimi.it

Angelo Gargantini
University of Bergamo
angelo.gargantini@unibg.it

Elvinia Riccobene
University of Milan
elvinia.riccobene@unimi.it

Abstract

A model review is a validation technique aimed at determining if a model is of sufficient quality and allows defects to be identified early in the system development, reducing the cost of fixing them. In this paper we propose a technique to perform *automatic* review of Abstract State Machine (ASM) formal specifications. We first detect a family of typical vulnerabilities and defects a developer can introduce during the modeling activity using the ASMs and we express such faults as the violation of meta-properties that guarantee certain quality attributes of the specification. These meta-properties are then mapped to temporal logic formulas and model checked for their violation. As a proof of concept, we also report the result of applying this ASM review process to several specifications.

1 Introduction

Using formal methods, based on rigorous mathematical foundations, for system design and development is of extreme importance, especially for high-integrity systems where safety and security are important. By means of abstract models, faults in the specification can be detected as early as possible with limited effort. Validation should precede the application of more expensive and accurate verification methods, that should be applied only when a designer has enough confidence that the specification really reflects the user perceptions. Otherwise (right) properties could be proved true for a wrong specification.

Model review, also called "model walk-through" or "model inspection", is a validation technique in which modeling efforts are critically examined to determine if a model not only fulfills the intended requirements, but also are of sufficient quality to be easy to develop, maintain, and enhance. This process should, therefore, assure a certain degree of quality. The assurance of quality, namely ensuring readability and avoiding error-prone constructs, is one of the most essential aspects in the development of safety-critical reactive systems, since the failure of such systems – often attributable to modeling and, therefore, coding flaws – can cause loss of property or even human life [13]. When model reviews are performed properly, they can have a big payoff because they allow defects to be detected early in the system development, reducing the cost of fixing them.

Usually model review, which comes from the code-review idea, is performed by a group of external qualified people. However, this review process, if done by hand, requires a great effort that might be tremendously reduced if performed in an automatic way – as allowed by using formal notations – by systematically checking specifications for known vulnerabilities or defects. The question is *what* to check on and *how* to automatically check the model. In other words, it is necessary to identify classes of faults and defects to check, and to establish a process by which to detect such deficiencies in the underlying model. If these faults are expressed in terms of formal statements, these can be assumed as a sort of measure of the *model quality assurance*.

In this paper, we tackle the problem of automatically reviewing formal specifications given in terms of Abstract State Machines (ASMs) [4]. We first detect a family of typical vulnerabilities and defects a developer can introduce during the modeling activity using the ASMs and we express such faults as the violation of formal properties. These properties refer to model *attributes* and characteristics that should hold in any ASM model, independently from the particular model to analyze. For this reason they are

*This work was partially supported by the Italian Government under the project PRIN 2007 D-ASAP (2007XKEHFA)

called *meta-properties*. They should be true in order for an ASM model to have the required quality attributes. Therefore, they can be assumed as measures of model quality assurance. Depending on the meta-property, its violation indicates the presence of actual faults, or only of potential faults.

These meta-properties are defined in terms of temporal logic formulas that use two operators, *Always* and *Sometime*, to capture properties that must be true in every state or eventually true in at least one state of the ASM under analysis. Then, we translate these temporal formulas to Computational Tree Logic (CTL) formulas and we exploit the model checking facilities of AsmetaSMV [2, 1], a model checker for ASM models based on NuSMV [5], to check the meta-property violation.

The choice of defining a model review process for the ASM formal method is due to several reasons. First, the ASMs are powerful extensions of the Finite State Machines (FSMs), and it has been shown [4] that they capture the principal models of computation and specification in the literature. Therefore, the results obtained for the ASMs can be adapted to other state-transition based formal approaches. Furthermore, the ASMs are endowed with a set of tools [6, 1] (among which a model checker) which makes it possible to handle and to automate our approach. Finally, ASMs have been widely applied as a formal method for system specification and development, which makes available a certain number of nontrivial specifications on which to test our process.

The Abstract State Machine formal method is briefly presented in Section 2. Section 3 defines a function, later used in the meta-properties definition, that statically computes the firing condition of a transition rule occurring in the model. Meta-properties that are able to guarantee certain quality attributes of a specification are introduced in Section 4. In Section 5, we describe how it is possible to automate our model review process by exploiting the use of a model checker to check the possible violation of meta-properties. As a proof of concept, in Section 6 we report the results of applying our ASM review process to a certain number of specifications, going from benchmark models to test the meta-properties, to ASM models of real case studies of various degree of complexity. In Section 7, we present other works related to the model review process. Section 8 concludes the paper and indicates some future directions of this work.

2 Abstract State Machines

Abstract State Machines (ASMs), whose complete presentation can be found in [4], are an extension of FSMs [3], where *states* are multi-sorted first-order structures, i.e. domains of objects with functions and predicates (boolean functions) defined on them, and the *transition relation* is specified by "rules" describing how functions change from one state to the next.

Basically, a transition rule has the form of *guarded update* "**if** *Condition* **then** *Updates*" where *Updates* are a set of function updates of the form $f(t_1,\dots,t_n) := t$ which are simultaneously executed when *Condition* is true. f is an arbitrary n-ary function and $t_1,\dots,t_n,t$ are first-order terms.

To fire this rule in a state s_i, $i \geq 0$, all terms $t_1,\dots,t_n,t$ are evaluated at s_i to their values, say $v_1,\dots,v_n,v$, then the value of $f(v_1,\dots,v_n)$ is updated to v, which represents the value of $f(v_1,\dots,v_n)$ in the next state s_{i+1}. Such pairs of a function name f, which is fixed by the signature, and an optional argument $(v_1,\dots,v_n)$, which is formed by a list of dynamic parameter values v_i of whatever type, are called *locations*. They represent the abstract ASM concept of basic object containers (memory units), which abstracts from particular memory addressing and object referencing mechanisms. Location-value pairs (loc,v) are called *updates* and represent the basic units of state change.

There is a limited but powerful set of *rule constructors* that allow to express simultaneous parallel actions (`par`) or sequential actions (`seq`). Appropriate rule constructors also allow non-determinism (existential quantification `choose`) and unrestricted synchronous parallelism (universal quantification `forall`).

A *computation* of an ASM is a finite or infinite sequence $s_0,s_1,\dots,s_n,\dots$ of states of the machine, where s_0 is an initial state and each s_{n+1} is obtained from s_n by firing simultaneously all of the transition rules which are enabled in s_n. The (unique) *main rule* is a transition rule and represents the starting point of the computation. An ASM can have more than one *initial state*. A state s which belongs to a computation starting from an initial state s_0, is said to be *reachable* from s_0.

For our purposes, it is important to recall how functions are classified in an ASM model. A first distinction is between *basic* functions which can be *static* (never change during any run of the machine) or *dynamic* (may be changed by the environment or by machine *updates*), and *derived* functions, i.e. those coming with a specification or computation mechanism given in terms of other functions. Dynamic functions are further classified into: *monitored* (only read, as events provided by the environment), *controlled* (read and write (i.e. updated by transaction rules)), *shared* and *output* (only write) functions.

The ASMETA tool set [1] is a set of tools around the ASMs. Among them, the tools involved in our model review process are: the textual notation *AsmetaL*, used to encode fragments of ASM models, and the model checker *AsmetaSMV* [2], which is based on the NuSMV model checker [5] to prove temporal properties on ASM models.

3 Rule Firing Condition

In the following we introduce a method to compute, for each rule of the specification under review, the firing condition under which the rule is executed. We introduce a function *Rule Firing Condition* which returns this condition.

$$RFC : Rules \rightarrow Conditions$$

where *Rules* is the set of the rules of the ASM M under review and *Conditions* are boolean predicates over the state of M. *RFC* can be statically computed as follows. First we build a static directed graph, similar to a program control flow graph. Every node of the graph is a rule of the ASM and every edge has label $[u]c$ representing the conditions under which the target rule is executed. c is a boolean predicate and $[u]$ is a sequence of logical assignments of the form $v = t$, being v a variable and t a term. The condition c must be evaluated under every logical assignment $v = t$ listed in u. Figure 1 reports how to incrementally build the graph, together with the labels for the edges. By starting from the main rule, the entire graph is built, except for the rules that are never used or are not reachable from the main rule and for which the *RFC* evaluates to *false*. We assume that there are no recursive calls of ASM rules, so the graph is *acyclic*. In general, an ASM rule can call itself (directly or indirectly), but rule recursion is seldom used. However, recursion is still supported in derived functions, which are often used in ASM specifications.

Parallel rule R R_1 **par** R_2	$\begin{tikzcd} R \arrow[r, "{[]true}"] \arrow[dr, "{[]true}"'] & R_1 \\ & R_2 \end{tikzcd}$	Conditional rule R **if** c **then** R_1 **else** R_2 **endif**	$\begin{tikzcd} R \arrow[r, "{[]c}"] \arrow[dr, "{[]\neg c}"'] & R_1 \\ & R_2 \end{tikzcd}$
Let rule R **let** $x = t$ **in** R_1	$\begin{tikzcd} R \arrow[r, "{[x=t]true}"] & R_1 \end{tikzcd}$	Forall rule R **forall** x **in** D **with** a **do** R_x	$\begin{tikzcd} R \arrow[r, bend left, "{[x=d_1]a}"] \arrow[r, bend right, "{[x=d_n]a}"'] & R_x \end{tikzcd}$
Macro call rule R $R_m[t_1,..,t_n]$	$\begin{tikzcd} R \arrow[r, "{[x_1=t_1,\ldots,x_n=t_n]true}"] & R_m \end{tikzcd}$	Choose rule R **choose** x **in** D **with** a **do** R_x	$\begin{tikzcd} R \arrow[r, bend left, "{[x=d_1]a}"] \arrow[r, bend right, "{[x=d_n]a}"'] & R_x \end{tikzcd}$

Figure 1: Schemas for building the graph for *RFC*

For this reason the lack of recursive rules does not prevent to write realistic specifications.

To compute the *RFC* for a rule R, one should start from the rule R and visit the graph backward until the main rule is reached. The condition $RFC(R)$ is obtained by applying the following three steps. Initially, $R_x = R$ holds.

1. Expand every occurrence of $RFC(R_x)$ by substituting it with the conditions under which R_x is reached, i.e. the labels of the edges entering the node of R_x. If the graph has the schema shown besides, one must substitute $RFC(R_x)$ with $[u_1](RFC(R_1) \wedge c_1) \vee \cdots \vee [u_n](RFC(R_n) \wedge c_n)$

$$\begin{tikzcd}
R_1 \arrow[dr, "{[u_1]c_1}"'] & R_2 \arrow[d, "{[u_2]c_2}"] & \cdots\cdots & R_n \arrow[dll, "{[u_n]c_n}"] \\
& R_x & &
\end{tikzcd}$$

2. Eliminate every logical assignment by applying the following rules:
   - Distribute the $\vee$ (or) over the $\wedge$ (and):
     $$([u_1]A_1 \vee \cdots \vee [u_n]A_n) \wedge B \equiv [u_1](A_1 \wedge B) \vee \cdots \vee [u_n](A_n \wedge B)$$
   - Distribute the assignments: $[u](A \wedge B) \equiv [u]A \wedge [u]B$
   - Apply the assignments: $[u, x = t]A \equiv [u]A[x \leftarrow t]$

3. Apply again 1 until you reach a rule with no entering edges (main rule).

Example Consider the following example in which y and z are nullary functions of the machine and $\$x$ is a logical variable. The inner rules are labeled for their concise representation in the graph.

```
main rule R =
 par
  r1: forall $x in {0,2} with $x < 2 do
  r2:     if y < $x then
  r3:             z := y endif
  r4: skip
 endpar
```

$$\begin{tikzcd}
R \arrow[r, "true"] \arrow[dr, "true"'] & r1 \arrow[r, bend left, "{[\$x=0]\$x<2}"] \arrow[r, bend right, "{[\$x=2]\$x<2}"'] & r2 \arrow[r, "{[]y<\$x}"] & r3 \\
& r4 & &
\end{tikzcd}$$

To compute the condition under which rule `r3` fires, $RFC(r3)$, one must perform the following steps:

1. Apply the expansion of $RFC(r3)$: $RFC(r3) \equiv RFC(r2) \wedge y < \$x$
2. No assignment to eliminate, expand $RFC(r2)$:
   $$RFC(r3) \equiv ([\$x = 0](RFC(r1) \wedge \$x < 2) \vee [\$x = 2](RFC(r1) \wedge \$x < 2)) \wedge y < \$x$$
3. Distribute the $\vee$ over the $\wedge$:
   $$RFC(r3) \equiv [\$x = 0](RFC(r1) \wedge \$x < 2 \wedge y < \$x) \vee [\$x = 2](RFC(r1) \wedge \$x < 2 \wedge y < \$x)$$
4. Apply the assignments:
   $$RFC(r3) \equiv (RFC(r1) \wedge 0 < 2 \wedge y < 0) \vee (RFC(r1) \wedge 2 < 2 \wedge y < 2)$$
   $$RFC(r3) \equiv (RFC(r1) \wedge y < 0) \vee false$$
5. Expand the definition of $RFC(r1)$ which is *true*:
   $$RFC(r3) \equiv y < 0$$

4 Meta-properties

In this section we introduce some properties that should be proved in order to ensure that an ASM specification has some quality attributes. These properties refer to attributes that are defined independently from the particular ASM specification to be analyzed and they should be true in order to guarantee certain degree of quality for the ASM model. For this reason we call them *meta-properties*, and they are formally defined in the following.

The violation of a meta-property always means that a quality attribute is not met and may indicate a potential/actual fault in the model. The severity of such violation depends on the meta-property, each of which measures the degree of model adequacy to the guidelines of ASM modeling style we introduce in this paper in order to use the ASM method for safety critical systems. We have identified the following categories of model quality attributes.

Consistency guarantees that locations (memory units) are never simultaneously updated to different values (MP1). This fault is known as *inconsistent updates* and must be removed in order to have a correct model.

Completeness requires that every behavior of the system is explicitly modeled. This enforces explicit listing of all the possible conditions in conditional rules (MP2) and the actual updating of controlled locations (MP7).

Minimality guarantees that the specification does not contain elements – i.e. transition rules, domain elements, locations, etc. – defined or declared in the model but never used (MP3, MP4, MP5, MP6). Minimality of the state requires that only the necessary state functions are introduced (MP7). These defects are also known as *over-specification*.

4.1 Meta-property definition

To formally specify the above attributes in terms of meta-properties we have identified properties that must be true in every state and properties that must be eventually true in at least one state of the ASM under analysis. Given an ASM M and a predicate ϕ over a state of M, we define the operators *Always* and *Sometime* as follows:

$$\begin{aligned} M \models Always(\phi) &= \forall s_0 \in S_0\ \forall s \in \mathscr{R}(s_0):\ \phi(s) \\ M \models Sometime(\phi) &= \exists s_0 \in S_0\ \exists s \in \mathscr{R}(s_0):\ \phi(s) \end{aligned}$$

where S_0 is the set of initial states of M, and $\mathscr{R}(s_0)$ is the set of all the states reachable from s_0. In the following we present the meta-properties we have introduced, currently support, and use for automatic review of ASM models.

MP1. No inconsistent update is ever performed

An inconsistent update occurs when *two updates clash, i.e. they refer to the same location but are distinct* [7]. If a location is updated by only one rule, no inconsistent update occurs. Otherwise an inconsistent update is possible. Let's see these two examples:

```
main rule r_inc0 =
  par
    l := 1
    l := 2
  endpar
```

```
main rule r_inc1 =
  par
    if cond1 then l(a1) := t1 endif
    if cond2 then l(a2) := t2 endif
  endpar
```

In the first example, the same location l is updated to two different values (1 and 2) in two rules having both conditions *RFC* equal to *true*; in this case, the inconsistent update is apparent. In the second example, instead, to prove that the two updates are consistent, one should prove:

$$Always((cond1 \land cond2 \land a1 = a2) \rightarrow t1 = t2)$$

In general, for every pair of rules R_1 and R_2 that update two locations (f, a_1) and (f, a_2) to the values t_1 and t_2 respectively, the property:

$$Always((RFC(R_1) \land RFC(R_2) \land a_1 = a_2) \rightarrow t_1 = t_2) \qquad (1)$$

states that the two updates are never inconsistent. The violation of property (1) means that there exists a state in which R_1 and R_2 fire, $a_1 = a_2$, and $t_1 \neq t_2$.

```
if x > 0 then skip
else
  if x <= 0 then skip endif
endif
```

```
if a and b then skip
else
  if not a then skip endif
endif
```

Figure 2: Complete and incomplete if

MP2. Every conditional rule must be complete

In a conditional rule R = **if** c **then** R_{then} **endif**, without *else*, the condition c must be true if R is evaluated. Therefore, in a nested conditional rule, if one does not use the else branch, the last condition must be true. In Fig. 2 the inner conditional rule is complete in the left-hand code, incomplete in the right-hand one, since if a is *true* but b is false, then no branch in the conditional statements is chosen. Property

$$Always(RFC(R) \rightarrow c) \tag{2}$$

states that, when the conditional rule R is executed, its condition c is evaluated to true. A violation of property 2 means that there exists a behavior of the system that satisfies $RFC(R) \wedge \neg c$ but it is not explicitly captured by the model.

Corollary 1: Every Case Rule without otherwise must be complete Since the case rule can be reduced, by definition [4], to a series of conditional rules, the computation of RFC is straightforward. The meta-property *MP2* is applied to case rules as follows. Let $R =$ **switch** t **case** $t_1 : R_1 \quad \dots \quad$ **case** $t_n : R_n$ **endswitch** be a case rule. Its completeness is given by the following property:

$$Always(RFC(R) \rightarrow c_1 \vee c_2 \cdots \vee c_n) \tag{3}$$

where c_j is $t = t_j$ for each $j = 1 \dots n$. The violation of the property (3) means that there is a state in which the case rule R is executed and none of its conditions is true.

MP3. Every rule can eventually fire

Let R be a rule of our ASM model (forall, choose, conditional, update, ...); to verify that R is eventually executed, we must prove the following property:

$$Sometime(RFC(R)) \tag{4}$$

If the property is proved false, it means that rule R is contained in an unreachable model fragment.

Corollary 2: Every condition in a conditional rule is eventually evaluated to true (and false if the else branch is given) For every conditional rule, MP3 requires that there exists a path in which its guard is eventually true and, if the else is given, also a path in which its guard is eventually false. In the following example the guard of the inner conditional rule is never true.

```
if x > 0 then
    if x < 0 then skip endif
endif
```

Let $Q =$ **if** c **then** R_{then} [**else** R_{else}] **endif** be a conditional rule. The property 4 becomes, for the **then** and **else** part, respectively:

$$Sometime(RFC(Q) \wedge c) \quad (5) \qquad\qquad Sometime(RFC(Q) \wedge \neg c) \quad (6)$$

```
  enum domain State = { AWAITCARD | AWAITPIN | CHOOSE | OUTOFSERVICE | OUTOFMONEY}
  dynamic controlled atmState: State
  dynamic controlled atmInitState: State
  dynamic controlled atmErrState: State
  dynamic monitored pinCode: Integer
  main rule r_Main =
    par
      if(atmState = atmInitState) then atmState := AWAITPIN endif
      if(atmState=AWAITPIN)       then atmState := CHOOSE endif
      if(atmState=CHOOSE)         then atmState := AWAITCARD endif
    endpar

default init s0:
  function atmInitState = AWAITCARD
  function atmErrState = OUTOFSERVICE
  function atmState = atmInitState
```

Figure 3: Over-specified ATM

MP4. No assignment is always trivial

An update $l := t$ is trivial [7] if l is already equal to t, even before the update is applied. This property requires that each assignment which is eventually performed, will not be always trivial. Let $R = \ l := t$ be an update rule. Property

$$Sometime(RFC(R)) \rightarrow Sometime(RFC(R) \wedge l \neq t) \tag{7}$$

states that, if eventually updated, the location l will be updated to a new value at least in one state. The more simple property $Sometime(RFC(R) \wedge l \neq t)$ would be false if the update is never performed.

MP5. For every domain element e there exists a location which has value e

Every domain element should be used at least once as location value. In the example of Fig. 3, the element `OUTOFMONEY` of the domain `State` is never used. To check that a domain element $e_j \in D$ is used as location value, if $l_1, \dots, l_n$ are all the locations (possibly defined by different function names) taking value in the domain D, the property

$$Sometime(l_1 = e_j \vee l_2 = e_j \vee \dots \vee l_n = e_j) \tag{8}$$

states that at least a location once takes the value e_j. Note that this property must be restricted to domains that are only function co-domains: if the domain D is used as domain of an n-ary function with $n > 0$, all its elements have to be considered useful, even if property 8 would be false for some $e_j \in D$. Otherwise, if property 8 is false, the element e_j may be removed from the domain.

MP6. Every controlled function can take any value in its co-domain

Every controlled function is assigned at least once to each element in its co-domain; otherwise it could be declared over a smaller co-domain. Let $l_1 \dots l_m$ be the locations of a controlled function f with co-domain $D = \{e_1, \dots, e_n\}$. Property

$$Sometime(l_1 = e_1 \vee \cdots \vee l_m = e_1) \wedge \dots \wedge Sometime(l_1 = e_n \vee \dots \vee l_m = e_n) \tag{9}$$

states that f takes all the values of its co-domain D.

MP7. Every controlled location is updated and every location is read

This property is obtained combining the results of the previous meta-properties and a static inspection of the model. It is defined by the following table, which also shows what actions the various results suggest.

controlled[1]	**initialized**[2]	**updated**[3]	**always trivial update**[4]	**read**[5]	**Possible actions**
false	N/A	N/A	N/A	false	remove
true	-	false	N/A	false	remove
true	true	false	N/A	true	declare static/add an update
true	true	true	true	-	declare static

In the example in Fig. 3 the monitored location *pinCode* is never read; it could be removed. The controlled *atmErrState* location is initialized, but never updated nor read; it could be removed. The controlled *atmInitState* location is initialized, read, but never updated; it could be declared static. Note that if a controlled location is never updated, that may suggest that the specification is incomplete and it misses an update to a part of the controlled state.

5 Meta-Property Verification by Model Checking

To verify (or falsify) the meta-properties introduced in the previous section, we use the AsmetaSMV tool [2] which is able to prove temporal properties of ASM specifications by using the model checker NuSMV [5]. The ASM specification M is translated to a NuSMV machine M_{NuSMV} (as explained in [2]) representing the Kripke structure which is model checked to verify a given temporal property. In this paper we use the CTL (Computation Tree Logic) language to express the properties to be verified by NuSMV. In NuSMV, a CTL property ψ is true if and only if ψ is true in every initial state of the machine M_{NuSMV}, i.e. $M_{NuSMV} \models \psi$ iff $(M_{NuSMV}, s_0) \models \psi, \forall s_0 \in S_0$, where S_0 is the set of initial states of M_{NuSMV}. Since the Kripke structure obtained from the ASM may have many initial states, the translation of the meta-properties as defined in Sect. 4.1 into CTL formulas is not straightforward. The translation of $Always(\phi)$ is simply $\texttt{AG}(\phi)$, since $M_{NuSMV} \models \texttt{AG}(\phi)$ means that along all paths starting from each initial state, ϕ is true in every state (globally), which corresponds to the definition of *Always*. However, the translation of $Sometime(\phi)$ is not $\texttt{EF}(\phi)$, since $M_{NuSMV} \models \texttt{EF}(\phi)$ means that there exists at least one path starting from *each* initial state containing a state in which ϕ is true, while *Sometime* requires only that there exists *at least* an initial state from which ϕ will eventually hold. This means that there are cases in which $\texttt{EF}(\phi)$ is false, since not from every initial state ϕ will eventually be true, while $Sometime(\phi)$ is true. To prove $Sometime(\phi)$ we use the following equivalence:

$$M \models Sometime(\phi) \Leftrightarrow M_{NuSMV} \not\models \texttt{AG}(\neg\phi)$$

that means that $Sometime(\phi)$ is true if and only if $\texttt{AG}(\neg\phi)$ is false. We run the model checker with the property $P = \texttt{AG}(\neg\phi)$ and if a counter example of P is found, then $Sometime(\phi)$ holds, while if P is proved true, then $Sometime(\phi)$ is false.

6 Experimental results

We have implemented a prototype tool, available at [1], that has allowed us to apply our model review process to three different sets of ASM specifications. The first set `Bench` contains only the benchmarks we have explicitly designed to expose the violations of the introduced meta-properties. The set `AsmRep` contains models taken from the ASMETA repository which are also available at [1]. Many ASM case studies of various degree of complexity and several specifications of classical software engineering systems (like ATM, Dining Philosophers, Lift, etc.) are included in `AsmRep`. The last set, `Stu`, contains the

[1] *true* if it is a controlled location, *false* if it is a monitored/static/derived location. Statically checked.
[2] *true* if the location is initialized. It is applicable only to controlled locations. Statically checked.
[3] *true* if the location is updated. It is applicable only to controlled locations. Checked by MP3.
[4] *true* if the update is always trivial. It is applicable only to controlled locations. Checked by MP4.
[5] *true* if the location is read at least in one state. Checked by MP3.

Spec Set	# spec.	# rules	# violations	violated MPs (# violations)
`Bench`	21	384	61	All
`AsmRep`	18	506	29	MP4(11), MP6(8), MP5(5), MP7(4), MP3(1)
`Stu`	6	172	38	MP7(11), MP5(9), MP6(9), MP1(3), MP3(3), MP4(3)

Table 1: Experimental results and violations found

models written by our students in a master course in which the ASM method is taught. The results of our experiments are reported in Table 1 which shows the name of the set, the number of models in it, the total number of rules in those models, the number of violations we detected, and the violations found in terms of meta-properties.

As expected our tool was able to detect all the violations in the benchmarks. The student projects contained several faults, most regarding the model minimality but also some inconsistencies which were not detected by model simulation. We found also several violations in the models of `AsmRep`, all of them regarding model minimality. Note that not all the models in `AsmRep` could be analyzed, since AsmetaSMV does not support all the AsmetaL constructs and it can analyze only finite models. We plan to use SMT solvers and Bounded Model Checking to analyze ASMs with infinite states.

7 Related work

Typical automatic reviews of formal specifications include simple syntax checking and type checking. This kind of analysis is performed by simple algorithms which are able to immediately detect faults like wrong use of types, misspelled variables, and so on. Some complex type systems may require proving of real theorems, like the non-emptiness of PVS types [11]. The review we propose in this paper is more similar to the kind of reviews proposed by Parnas and his colleagues. In a report about the certification of a nuclear plant, he observed that "reviewers spent too much of their time and energy checking for simple, application-independent properties" (like our meta-properties) which distracted them from the more difficult, safety-relevant issues." [12]. Tools that automatically perform such checks can save reviewers considerable time and effort, liberating them to do more creative work.

Our approach has been greatly influenced by the work done by the group lead by Heitmeyer with the Software Cost Reduction (SCR) method. SCR features a tabular notation which can be checked for *completeness* and *consistency* [8]: completeness guarantees that each function is totally defined by its table and consistency guarantees that every value of controlled and internal variables is uniquely defined at every step. In [9] is described a method, similar to ours, to automatically verify the consistency of a software requirements specification (SRS) written in an SCR-style; properties that describe the consistency of the model are defined *structural properties*. The SRS document is translated into a PVS model where, for each structural property, a PVS theorem is declared. The verification of structural properties is carried out through the proof of PVS theorems and, for one property, through the model checking of a CTL property.

Other approaches try to apply analyses similar to those performed in SCR to non-tabular notations. In [13], the authors present a set of robustness rules (like UML well-formedness rules) that seek to avoid common types of errors by ruling out certain modeling constructs for UML state machines or Statecharts. Structural checks over Statecharts models can be formulated by OCL constraints which, if complex, must be proved by theorem proving. Their work and ours extend the use of meta-properties not only to guarantee correctness but also to assure high quality standards in case the models are to be used for safety critical applications.

8 Conclusions and Future work

We have presented a method to perform automatic model review of ASM specifications. This process has the aim of guarantee certain quality attributes of models. A given quality attribute is captured by a meta-property expressed in terms of a CTL formula. The AsmetaSMV model checker for ASMs is used to detect a possible violation of this meta-property and, therefore, the presence of a possible defect in the model. These meta-properties can be assumed as measures of model quality assurance.

In the future we plan to improve our process in the following directions. One of the typical shortcomings introduced by a not ASM-expert when modeling with ASMs is the use of the `seq` rule constructor when `par` could be used, instead. This is usually due by a wrong understanding of the simultaneous parallel execution of function updates. The correct use of a `par` instead of a `seq` can improve the quality of a model in terms of abstraction and minimality. So we plan to investigate this kind of defect and define suitable meta-properties able to detect it. Another future plan regards the *vacuity detection* [10] of (temporal) properties which can be specified for an ASM model. We plan to investigate if it is possible to detect property vacuity before proving properties. To this purpose, an integration of the AsmetaSMV system with existing tools able to detect vacuity could be possible.

References

[1] The ASMETA website. `http://asmeta.sourceforge.net/`, 2010.

[2] P. Arcaini, A. Gargantini, and E. Riccobene. AsmetaSMV: a way to link high-level ASM models to low-level NuSMV specifications. In M. Frappier, U. Glässer, S. Khurshid, R. Laleau, and S. Reeves, editors, *Abstract State Machines, Alloy, B and Z, Second Inter. Conference, ABZ 2010*, volume 5977 of *Lecture Notes in Computer Science*, pages 61–74. Springer, 2010.

[3] E. Börger. The ASM method for system design and analysis. A tutorial introduction. In B. Gramlich, editor, *Frontiers of Combining Systems, 5th International Workshop, FroCoS 2005, Vienna, Austria, September 19-21, 2005, Proceedings*, volume 3717 of *Lecture Notes in Computer Science*, pages 264–283. Springer, 2005.

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

[5] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*. Springer, July 2002.

[6] A. Gargantini, E. Riccobene, and P. Scandurra. Model-driven language engineering: The ASMETA case study. In *International Conference on Software Engineering Advances, ICSEA*, pages 373–378, 2008.

[7] Y. Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Trans. Comput. Logic*, 1(1):77–111, 2000.

[8] C. Heitmeyer, R. Jeffords, and B. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.

[9] T. Kim and S. D. Cha. Automated structural analysis of SCR-style software requirements specifications using PVS. *Softw. Test, Verif. Reliab*, 11(3):143–163, 2001.

[10] O. Kupferman and M. Y. Vardi. Vacuity detection in temporal model checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(2):224–233, 2003.

[11] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE-11)*, pages 748–752, London, UK, 1992. Springer-Verlag.

[12] D. L. Parnas. Some theorems we should prove. In *HUG '93: 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 155–162, London, UK, 1994. Springer-Verlag.

[13] S. Prochnow, G. Schaefer, K. Bell, and R. von Hanxleden. Analyzing robustness of UML state machines. In *Workshop on Modeling and Analysis of Real-Time and Embedded Systems (MARTES 06)*, 2006.