Electronic Communications of the EASST
Volume 70 (2014)

Proceedings of the 14th International Workshop on Automated Verification of Critical Systems (AVoCS 2014)

Using SMT for dealing with nondeterminism in ASM-based runtime verification

Paolo Arcaini, Angelo Gargantini, Elvinia Riccobene

15 pages

Guest Editors: Marieke Huisman, Jaco van de Pol
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Using SMT for dealing with nondeterminism in ASM-based runtime verification

Paolo Arcaini[1], **Angelo Gargantini**[1], **Elvinia Riccobene**[2]

[1]Dipartimento di Ingegneria, Università degli Studi di Bergamo, Italy
[2]Dipartimento di Informatica, Università degli Studi di Milano, Italy

Abstract: In runtime verification, operational models describing the expected system behavior offer some advantages with respect to declarative specifications of properties, especially when designers are more accustomed to them. However, nondeterminism in the specification usually affects performances of those operational methods that explicitly represent all the possible conformant states. In this paper, we tackle the problem of dealing with nondeterminism in an operational runtime verification approach based on the use of Abstract State Machines (ASMs). We propose an SMT-based technique in which ASM computations are symbolically represented and conformance verification is performed by means of satisfability checking. Experiments show that, in most of the cases, the symbolic approach performs better than a technique for ASM-based runtime verification explicitly representing the conformant states.

Keywords: runtime verification, nondeterminism, Abstract State Machines, SMT

1 Introduction

Runtime verification in the form of conformance monitoring consists in checking during runtime that the monitored system behaves like specified. Monitors [FHR13] are used to assess the correctness of a system behavior by checking whether the *observed* state of the implemented system is conformant to the *expected* state provided by an abstract specification of the system.

In most approaches dealing with runtime verification of software, the required behavior of the system is specified by means of correctness properties [DGR04]. Temporal logic-based formalisms are very popular in runtime verification [CDR04], especially variants of linear temporal logic [BLS11]. However, *operational* notations can be used in runtime verification as well [AGR12, BS03a, LDSW09]. Operational specifications offer some advantages with respect to declarative specifications of properties, especially when designers are more accustomed to them. Since operational models may be executable and easier to write and understand, they can be used starting from the first stages of the software development both for documentation purposes and validation activities, as simulation and model-based testing [BS03a]. Moreover, operational approaches usually permit to trace, by means of a step-wise model refinement [BS03b], the relation between the specification and the implementation.

The Abstract State Machine (ASM) operational method [BS03b] has been used in [AGR12], where we developed a framework (called CoMA) for runtime verification of Java programs. The technique proposed for conformance monitoring makes use of Java annotations to link the concrete implementation to its ASM formal model.

In case of not fully predictable systems, models specifying system behaviors are nondeterministic. Nondeterminism in the specification can also be due to underspecification, when some implementation choices are left abstract, or model abstraction used to reduce complexity. ASMs permit to model nondeterministic behaviors in a concise and natural way, thanks to the built-in `choose` operator that allows to nondeterministically select a computation path among all possible ones. However, nondeterminism poses further challenges to the ASM-based runtime monitoring approach. In [AGR13], we extended CoMA in order to deal with nondeterminism by explicitly building and keeping track of all the possible next states: that approach becomes very inefficient when the number of next states is high. Although this limitation is unavoidable if one needs to keep track of all the nondeterministic *choices* performed, techniques can be developed to mitigate this shortcoming. For example, symbolic representation can help to specify in a concise way the relation between a state and the set of reachable next states.

Following our ongoing research on ASM-based runtime verification of Java programs, to overcome the limits of CoMA, we here present an approach exploiting a symbolic representation of a machine computation [VBGS09], similarly to what is done in bounded model checking where a program trace is represented by means of a propositional formula [Bie09].

Starting from the theoretical framework we presented in [AGR12, AGR13], we here provide a novel definition of conformance in the presence of nondeterminism, and propose CoMA-SMT, an SMT-based technique in which ASM computations are symbolically represented and the conformance verification is performed by means of satisfability checking.

The devised approach requires the designer to write an ASM specification of the system and to link it to the implementation to be monitored. The specification is automatically translated into an SMT logical context. During runtime, the SMT context is step by step extended with the current transitions, and the values observed in the monitored system are asserted. The implementation behaves correctly as long as the SMT context stays satisfiable: if the context becomes unsatisfiable, a runtime fault is observed.

We have compared CoMA-SMT with CoMA, and our preliminary experiments show that, in most of the cases, the symbolic approach performs better than techniques based on explicit state representation. Moreover, other experiments show that CoMA-SMT can handle a more general notion of conformance, not supported by CoMA, that is much more difficult to check.

Section 2 presents a background about ASMs, nondeterminism in runtime monitoring, and CoMA. In Section 3, to deal with conformance in case of nondeterminism, we improve the theoretical framework CoMA is based on. How to symbolically represent an ASM is described in Section 4. Section 5 presents the proposed runtime monitoring approach. Preliminary experiments are reported in Section 6. Some related work is presented in Section 7, and Section 8 concludes the paper.

2 Background

2.1 ASMs

Abstract State Machines (ASMs), whose complete presentation can be found in [BS03b], are an extension of FSMs, where unstructured control states are replaced by states with arbitrary complex data. The *states* of an ASM are multi-sorted first-order structures, i.e., domains of

objects with functions and predicates defined on them. ASM states are modified by *transition relations* specified by "rules" describing the modification of the function interpretations from one state to the next one. There is a limited but powerful set of *rule constructors* that allow to express guarded actions (`if-then`), simultaneous parallel actions (`par`) or sequential actions (`seq`). Appropriate rule constructors also allow modeling nondeterminism (existential quantification `choose`) and unrestricted synchronous parallelism (universal quantification `forall`).

An ASM state s is represented by a set of couples (*location*, *value*). ASM *locations*, namely pairs (*function-name*, *list-of-parameter-values*), represent the abstract ASM concept of basic object containers (memory units). Location *updates* represent the basic units of state change and are given as assignments, each of the form $loc := v$, where loc is a location and v its new value.

Functions are classified as *derived*, i.e., those coming with a specification or computation mechanism given in terms of other functions, and *basic* which can be *static* (never change during any run of the machine) or *dynamic* (may change as a consequence of agent actions or *updates*). Dynamic functions are distinguished between *monitored* (only read by the machine and modified by the environment), and *controlled* (read and written by the machine).

A *computation* of an ASM is a finite or infinite sequence $s_0, s_1, \ldots, s_n, \ldots$ of states of the machine, where s_0 is an initial state and each s_{n+1} is obtained from s_n by simultaneously firing all 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*. It is possible to specify state *invariants* which are checked during simulation.

For our purposes, we use a definition of ASM adapted from [BS03b]: $ASM = <$ *signature*, *funcDefs*, *funcInit*, *r_main* $>$, where *signature* contains the function declarations, *funcDefs* the derived functions definitions, *funcInit* the definitions of initials values for the controlled functions, and *r_main* is the main rule.

2.2 Runtime monitoring and nondeterminism

Runtime verification of nondeterministic behaviors using state-based specifications, like ASMs, is particularly complex since the specification takes into account all the possible correct system evolutions. The nondeterminism due to monitored quantities (e.g., the system inputs or external actions), called *external*, is still easy to monitor: once these quantities are fixed by the environment, the system behaves deterministically. However, in most cases, the specification is *internally* nondeterministic, sometimes even when the system is deterministic. The following scenarios can be identified:

- The system has a nondeterministic behavior (for instance a Java program containing a call to a method in the class `java.util.Random`), as well as the abstract specification.
- The system has a deterministic behavior, while the specification is nondeterministic. This situation arises when the model is more abstract (with less implementation details) than the corresponding system, and is frequent in the object oriented context. Bekaert and Steegmans [BS01] have shown that nondeterminism in the behavioral specifications of object oriented conceptual models can simplify the representation of complex functionalities and achieve a better separation of concerns.

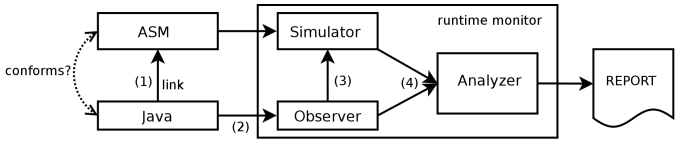


Figure 2: CoMA: Conformance monitoring through ASM

A simple example of nondeterministic system As running case study, we consider a tank that can be either filled or emptied (see Fig. 1). At every instant, the level cannot be increased/decreased more than fifty units of product. The tank is full when it contains one thousand units of product. Such tank can be modeled by a simple ASM, as shown in Code 1.

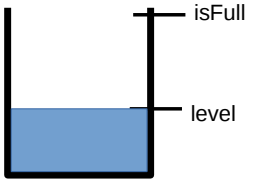


Figure 1: A tank

```
asm tank
signature:
    controlled level: Integer
    derived full: Boolean
definitions:
    function full = (level = 1000)

    main rule r_main =
        choose $x in {−50..50} with level + $x >= 0 and level + $x <= 1000 do
            level := level + $x

default init s0:
    function level = 0
```

Code 1: ASM model of the Tank case study

The controlled function level records the number of units in the tank; in the initial state the tank is empty. The boolean function full signals whether the tank is full: it is a derived function because its value depends on the value of function level. In the main rule, a choose rule nondeterministically increments/decrements the level of the tank of at most fifty units at a time, not exceeding the maximum capacity.

2.3 CoMA: Conformance Monitoring through ASMs

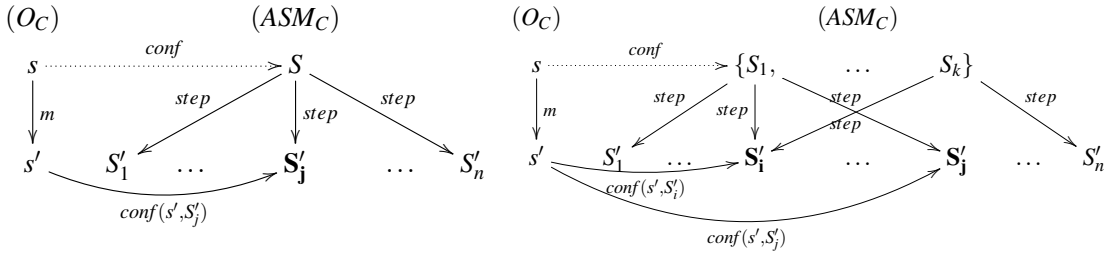
CoMA [AGR12] is a technique for runtime monitoring of Java programs through ASMs; Fig. 2 shows the structure of the framework. The runtime monitor observes the behavior of a Java code and determines its correctness w.r.t. the ASM specification working as an oracle of the expected behavior: while the software system is executed, the monitor checks conformance between the values of the observed elements and the expected state. A *link* between a Java class and an ASM is established using a set of Java annotations (1). The *Observer* detects when the Java object *observed* state is changed (2), and leads the corresponding ASM to perform a machine step (3). The *Analyzer* evaluates the conformance between the Java execution and the ASM behavior (4).

A complete description of CoMA can be found in [AGR12]. Here only some basic definitions are reported.

Let C be a Java class, O_C an object of C, and ASM_C an ASM model of the expected behavior of any object of C.

$OS(C)$ is the set of *observed elements*, i.e., all public fields, and *pure*[1] public methods of the class C the user wants to observe. Observable elements are linked to ASM functions by the function $link : OS(C) \rightarrow Funcs$.

[1] A method is *pure* when its execution does not affect the program state.



(a) Univocal

(b) Multiple

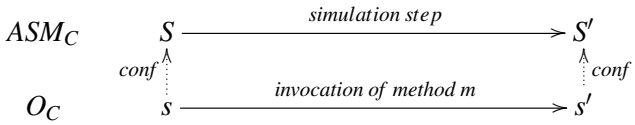
Figure 3: Runtime conformance

*Changing method*s, $CM(C)$, are the non-pure methods whose execution can change the element values of $OS(C)$ and that the user wants to monitor. A *changing step* is defined by the triple (s,m,s'), being m a method in $CM(C)$, and s and s' the states of an object O_C before and after the method execution.

Definition 1 (State and step conformance) A *state* s of O_C *conforms to a state* S of ASM_C if all observed elements of C have values in O_C conforming to the values of the functions in ASM_C linked to them, i.e.,

$$conf(s,S) \equiv \quad \forall e \in OS(C) : val_{Java}(e,s) \overset{conf}{=} val_{ASM}(link(e),S)$$

A *change step* (s,m,s') of an instance O_C, with m a method of $CM(C)$, *conforms with a step* (S,S') of ASM_C if $conf(s,S) \wedge conf(s',S')$.



Definition 2 **Univocal runtime conformance** A class C is *univocally* runtime conforming to its specification ASM_C if the following conditions hold:

1) the initial state s_0 of the computation of O_C conforms to *one and only one* initial state S_0 of the computation of ASM_C, i.e., $\exists! S_0$ initial state of ASM_C such that $conf(s_0,S_0)$;
2) for every change step (s,m,s') with s the current state of O_C, $\exists! (S,S')$ step of ASM_C with S the current state of ASM_C, such that (s,m,s') is *step conforming* with (S,S').

Univocal runtime conformance requires that the next step of O_C is state-conforming with *one and only one* of the next states of the specification. Therefore, in case of nondeterminism, during the runtime monitoring CoMA chooses, among the next states of the ASM, the unique state that conforms to the Java state. Fig. 3a depicts this situation.

Since CoMA represents the states in an explicit way, as in explicit state model checkers (e.g., SPIN), we will refer to it as an *explicit state* monitoring approach.

Example Let's consider the Tank case study. If the conformance between the ASM specification and the Java program is based on the value of the level, then the conformance is univocal. At each step, the ASM has between fifty and a hundred one possible next states; such states, however, are uniquely identified by the value of level. So, if the implementation is correct, only one of the possible next ASM states is conformant.

3 Dealing with multiple conformance

Definition 2 assumes that, at every time, there is only one possible current ASM state which is conformant to the current Java state. However, the implementation should be considered conformant also when there exists *at least one* conformant state.

Definition 3 Conformant set Given a Java object O_C, let s_n be the state obtained after n executions of changing methods. We call $confSet(s_n)$ the set of ASM states reachable in n steps and conformant with s_n.

Definition 4 Multiple runtime conformance A class C is *multiply* runtime conforming to its specification ASM_C if the following conditions hold:

1) the initial state s_0 of the computation of O_C conforms to *at least one* initial state S_0 of the computation of ASM_C, i.e., $\exists S_0$ initial state of ASM_C such that $conf(s_0, S_0)$;
2) for every change step (s, m, s') with s the current state of O_C, $\exists (S, S')$ step of ASM_C, $S \in confSet(s)$, such that (s, m, s') is *step conforming* with (S, S').

Multiple runtime conformance can be depicted as in Fig. 3b: The current Java state s is conformant with the ASM states $confSet(s) = \{S_1, \ldots, S_k\}$; the Java state s' is produced by the execution of the method m at the state s; ASM states $S'_1, \ldots, S'_n$ are reachable in one step from $confSet(s)$; the Java state s' is conformant with ASM states $\{S'_i, \ldots, S'_j\} \subseteq \{S'_1, \ldots, S'_n\}$.

Example Let's consider the Tank case study. If the conformance between the ASM specification and the Java program is only based on the value of function full, then the conformance is multiple. As seen before, at each step the ASM has between fifty and a hundred one possible next states; at most one next state can have value *true* for full. Therefore, if the implementation is correct and the value of full is *false*, more than one of the possible next ASM states can be conformant with the implementation.

Supporting multiple runtime conformance in monitoring Dealing with multiple conformance in an explicit state runtime monitoring approach as CoMA, would require to keep track of all the possible states to which the monitored system can be conformant. At the ith step of monitoring, the framework should record in the set $confSet(s_i)$ (see Def. 3) the ASM states reachable in i steps of simulation that are conformant with the current Java state (as proposed in [FHR13]). If $confSet(s_i)$ becomes empty, then an error is found. In our approach, instead, we would like to represent such reachable and conformant states in a symbolic way.

In order to do this, we describe how to symbolically represent the set of states RS_i reachable in i steps of the ASM execution, and the transition relation induced by the ASM transition rules

between states in RS_i and their successor states in RS_{i+1}. These formulae establish a logical *context*.

At runtime, in order to perform the conformance checking, we extend the context by asserting a set of formulae stating the values of the observed elements in the implementation current state. A Satisfiability Modulo Theories (SMT)[2] solver can be used to check the satisfiability of the obtained context. If the context becomes unsatisfiable, then the implementation is not conformant. As SMT solver we use Yices [DM06].

Note that we could use BDDs to symbolically represent ASM states; however, it has been shown that for bounded model checking a SAT/SMT approach scales better [Bie09, Kur08]. So, since the approach we are proposing has several commonalities with BMC, we adopt the SMT approach.

Example Let's consider the Tank case study. The ASM states reachable in one step can be symbolically described as $\mathsf{level} = 0 \wedge (\mathsf{level} - 50 \leq \mathsf{level}' \leq \mathsf{level} + 50)$, where level' represents the updated version of level.

4 ASM symbolic representation

This section describes how to build the logical context, symbolically representing any computation of length n of a given ASM. Section 4.1 describes how to represent the set of states RS_i at level i of the ASM computation – i.e., those reachable by the machine in i steps – and the transition relation induced by the ASM transition rules between states in RS_i and their successor states in RS_{i+1}. Some mapping functions permit to obtain, from an ASM model, a sequence of Yices definitions and assertions (commands) parameterized with the index i. In order to symbolically represent ASM computations of depth n, the Yices commands must be instantiated n times with concrete values for i (i.e., $i = 0, \ldots, n-1$), as described in Section 4.2.

4.1 Mapping from ASM to Yices

For the lack of space, we do not report the mapping of ASM domains to Yices types, that, however, is not relevant for the understanding of the work. The signature symbols are **define**d by applying the mapping reported in Table 1 ($\mathtt{T_f}$), being i the current level; note that, for each ASM function, a fresh Yices constant is created at every step. Function definitions are **assert**ed by applying the correspondence given in Table 2 ($\mathtt{T_d}$), at level i. Transition rules are symbolically represented by a formula describing the transition relation between states in RS_i and those in RS_{i+1}. This formula is **assert**ed by recursively applying the mapping reported in Table 3 ($\mathtt{T_r}$), starting from the main rule.

Both function definitions and transition rules contain terms: the symbolic representation of terms in states of RS_i is given in Table 4 ($\mathtt{T_t}$).

Note that in an update rule (first row of Table 3) the location term on the left-hand side of the rule refers to states in RS_{i+1}, while the term on the right-hand side of the rule refers to states in

[2] An SMT problem is a decision problem for logical formulae with respect to combinations of background theories expressed in classical first-order logic with equality. An SMT instance is a generalization of a boolean SAT instance in which various sets of variables are replaced by predicates from a variety of underlying theories.

ASM function declaration funcType ∈ {controlled, monitored, derived, static}	**Yices**
funcType f: Dom	(**define** f^i :: Dom)
funcType f: D_1 −> D_2	(**define** f^i :: (−> D_1 D_2))
funcType f: Prod(D_1, …, D_n) −> D with $n \geq 2$	(**define** f^i :: (−> D_1 … D_n D))

Table 1: $\mathtt{T_f}$: Mapping schema of the ASM function declarations to Yices at step i

ASM function definition	**Yices**
function f = fd	(**assert** (= f^i $\mathtt{T_t}$(fd, i)))
function f(x_1 **in** D_1, …, x_n **in** D_n) = fd[x_1, …, x_n] with $n \geq 1$ and D_1={d_1^1, …, $d_{m_1}^1$} … D_n={d_1^n, …, $d_{m_n}^n$}	(**assert** (**and** (= $\mathtt{T_t}$(f(d_1^1, …, d_1^n), i) $\mathtt{T_t}$(fd[$x_1 \mapsto d_1^1$, …, $x_n \mapsto d_1^n$], i)) … (= $\mathtt{T_t}$(f($d_{m_1}^1$, …, $d_{m_n}^n$), i) $\mathtt{T_t}$(fd[$x_1 \mapsto d_{m_1}^1$, …, $x_n \mapsto d_{m_n}^n$], i))))

Table 2: $\mathtt{T_d}$: Mapping schema of ASM function definitions to Yices at step i

ASM transition rule	**Yices**
updLoc := updTer	$\mathtt{T_t}$(updLoc, $i+1$) = $\mathtt{T_t}$(updTer, i)
par R_1 … R_n **endpar**	(**and** $\mathtt{T_r}$(R_1, i) … $\mathtt{T_r}$(R_n, i))
if guard **then** Rthen **else** Relse **endif**	(**if** $\mathtt{T_t}$(guard, i) $\mathtt{T_r}$(Rthen, i) $\mathtt{T_r}$(Relse, i))
if guard **then** Rthen **endif**	(=> $\mathtt{T_t}$(guard, i) $\mathtt{T_r}$(Rthen, i))
forall x_1 **in** D_1, …, x_n **in** D_n **with** guard[x_1, …, x_n] **do** R[x_1, …, x_n]	(**and** r_1 … r_m) with $m = \prod_{j=1}^{n} \lvert D_j \rvert$ where for each $\overline{d}_k$ = (d_1, …, d_n) ∈ $D_1 \times \ldots \times D_n$ r_k = (=> $\mathtt{T_t}$(guard[$x_1 \mapsto d_1$, …, $x_n \mapsto d_n$], i) $\mathtt{T_r}$(R[$x_1 \mapsto d_1$, …, $x_n \mapsto d_n$], i))
choose x_1 **in** D_1, …, x_n **in** D_n **with** guard[x_1, …, x_n] **do** R[x_1, …, x_n]	for each x_j: (**define** cv_j^i :: D_j) (=> (**exists** (d_1:: D_1 … d_n:: D_n) $\mathtt{T_t}$(guard[$x_1 \mapsto d_1$, …, $x_n \mapsto d_n$], i)) (**and** $\mathtt{T_t}$(guard[$x_1 \mapsto cv_1^i$, …, $x_n \mapsto cv_n^i$], i) $\mathtt{T_r}$(R[$x_1 \mapsto cv_1^i$, …, $x_n \mapsto cv_n^i$], i)))
main rule r_main = mainBody	(**assert** $\mathtt{T_r}$(mainBody, i))

Table 3: $\mathtt{T_r}$: Mapping schema of ASM transition rules to Yices at step i

RS_i. For this reason, the term on left-hand side is mapped with parameter $i+1$, whereas the term on the right-hand side is mapped with parameter i.

In order to guarantee the semantics of ASM steps, we must impose that locations that are not updated by the computation step keep their value unchanged. The formula obtained by applying the mapping function $\mathtt{T_r}$ to the transition rules (Table 3) does not guarantee such condition. Therefore, for each controlled location f^i, we need to add to the context the following formula

(=> (not (**or** $guard_1$ … $guard_n$)) (= f^{i+1} f^i))

ASM term	**Yices**
Location term: f	f^i
Location term: f(a_1, …, a_n) with $n \geq 1$	(f^i $T_t(a_1, i)$ … $T_t(a_n, i)$)
Boolean term: b with $b \in$ {true, false}	b
Integer term: h with $h \in \mathbb{Z}$	h
Natural term: hn with $h \in \mathbb{N}$	h
Enumeration term: E	E
if guard **then** Tthen **else** Telse **endif**	(**if** T_t(guard, i) T_t(Tthen, i) T_t(Telse, i))
(**forall** x_1 **in** D_1, …, x_n **in** D_n **with** cond[x_1, …, x_n])	(**and** c_1 … c_m) with $m = \prod_{j=1}^{n} \lvert D_j \rvert$ where for each $\bar{d}_k = (d_1, \ldots, d_n) \in D_1 \times \ldots \times D_n$: c_k = T_t(cond[$x_1 \mapsto d_1, \ldots, x_n \mapsto d_n$], i)
(**exists** x_1 **in** D_1, …, x_n **in** D_n **with** cond[x_1, …, x_n])	(**or** c_1 … c_m) with $m = \prod_{j=1}^{n} \lvert D_j \rvert$ where for each $\bar{d}_k = (d_1, \ldots, d_n) \in D_1 \times \ldots \times D_n$: c_k = T_t(cond[$x_1 \mapsto d_1$, …, $x_n \mapsto d_n$], i)

Table 4: T_t: Mapping schema of ASM terms to Yices at step i

being $guard_1, \ldots, guard_n$ the conditions upon which f^i is updated, and f^{i+1} the location in the next state. The conditions are statically derived from the transition rules leading to the updates of the location. Let $unchLocs_i$ be the assertion of the set of formulae obtained by instantiating the previous formula for all the controlled locations of the ASM model at level i.

4.2 Building the logical context

The logical context is initialized and then extended at every level i along the ASM computation.

Context initialization consists of a set of formulae symbolically representing the ASM initial state(s), i.e., signature, and function definitions and initializations. Context initialization $contInit$ is built as follows

$$contInit = T_f(signature, 0), T_d(funcDefs, 0), T_d(funcInit, 0)$$

where the comma is to be intended as a sequential concatenation operator.

Context extension consists of a set of formulae representing the transition relation between states in RS_i and their successor states in RS_{i+1}. This requires to define the signature at level $i+1$. Context extension $contExt_i$ at level i is built as follows

$$contExt_i = T_f(signature, i+1), T_d(funcDefs, i+1), T_r(r\_main, i), unchLocs_i$$

At every step, $T_f(signature, i+1)$ represents a fresh copy of the signature and it must be added to the context before the assertion of the rules $T_r(r\_main, i)$, because $T_r(r\_main, i)$ describes the relation between states in RS_i and states in RS_{i+1}.

Code 2 reports the context initialization and context extension at level 0 of the Tank case study.

```
;; T_f(signature, 0): Functions declarations − state 0
(define level0::int)
(define full0::bool)
;; T_d(funcDefs, 0): Derived functions definitions − state 0
(assert (= full0 (= level0 1000)))
;; T_d(funcInit, 0): Initial state definition
(assert (= level0 0))

;; T_f(signature, 1): Functions declarations − state 1
(define level1::int)
(define full1::int)
;; T_d(funcDefs, 1): Derived functions definitions − state 1
(assert (= full1 (= level1 1000)))
;; T_r(r_main, 0): Transition rules − from state 0 to state 1
(define cv0::(subrange −50 50)) ;; Declaration of a variable for the choose rule
(assert (=> (exists (x::(subrange −50 50)) (and (>= (+ level0 x) 0) (<= (+ level0 x) 1000)) )
            (and (and (>= (+ level0 cv0) 0) (<= (+ level0 cv0) 1000)) (= level1 (+ level0 cv0)) ) ) )
;; unchLocs_0: Unchanged controlled locations − from state 0 to state 1
(assert (=> (not (and (>= (+ level0 cv0) 0) (<= (+ level0 cv0) 1000))) (= level0 level1) ) )
```

Code 2: Tank case study – Context initialization and context extension at level 0

5 SMT-based Runtime Verification

We here describe an approach to perform runtime monitoring of Java programs, in which the expected behavior is given in terms of ASMs; the approach, called CoMA-SMT, exploits the symbolic representation of ASMs introduced in Section 4. We keep the overall architecture of the CoMA framework (Fig. 2). The technique, described in Section 2.3, for linking an ASM specification to a Java class is the same. However, the way to simulate the ASM (the *simulator* in Fig. 2) and to perform the conformance checking (the *analyzer* in Fig. 2) differ. The approach is also able to check multiple runtime conformance (see Def. 4), not supported by CoMA.

Alg. 1 depicts the monitoring procedure of the proposed approach. In order to ease the description, Alg. 1 reports both the execution of the monitored Java program (reported in a frame in the algorithm) and the execution of the monitoring framework. When a Java object of the monitored class is created (line 1), the framework creates a logical context (line 2) and add the context initialization to it (line 3). The monitoring consists in a never ending loop in which, when a Java changing method m is executed (line 6), the following actions are executed:

- the context is extended for describing the transition relation between ASM states at the current level i and the possible next states at level $i+1$ (line 7);
- from the Java state s_{Java}, obtained after the changing method execution (line 8), and from the linking between the specification and the code, the framework builds the formula *javaValuesConstr* in which the linked ASM locations are forced to assume the actual values of the corresponding Java elements (line 9). Let $f_1^{i+1}, \ldots, f_g^{i+1}$ be the locations linked to Java fields or methods (i.e., the observed elements) and $v_1, \ldots, v_g$ the values of the linked fields and methods at state $i+1$. Formula *javaValuesConstr* is built as follows:
$$(\textbf{assert } (\textbf{and } (= \mathsf{f}_1^{i+1}\ v_1) \ldots (= \mathsf{f}_g^{i+1}\ v_g)))$$
- formula *javaValuesConstr* is asserted in the logical context (line 10);
- the logical context is checked for satisfiability (line 11):

Algorithm 1 CoMA-SMT: monitoring procedure

```
1:  o_C ← new C()                                  ▷ Monitored program: Java object instantiation
2:  ctx ← mk_context()                             ▷ Logical context creation
3:  add_to_context(contInit, ctx)                  ▷ Context initialization
4:  i ← 0
5:  while true do
6:      o_C.m()                                    ▷ Monitored program: execution of a changing method m
7:      add_to_context(contExt_i, ctx)             ▷ Context extension at level i
8:      s_Java ← [|o_C|]                           ▷ Observed Java state after step i
9:      javaValuesConstr ← getValues(s_Java)       ▷ Observed elements values
10:     add_to_context(javaValuesConstr, ctx)      ▷ Assertion of the observed values
11:     if check(ctx) = UNSAT then                 ▷ Is SAT?
12:         return NotConformantException          ▷ The Java state is not conformant
13:     end if
14:     i ← i + 1
15: end while
```

- if the context is unsatisfiable, it means that the implementation is not conformant with the specification. In this case, the monitoring is interrupted by throwing an error message (line 12);
- otherwise, if the context is still satisfiable, it means that the implementation is conformant and the monitoring can continue.

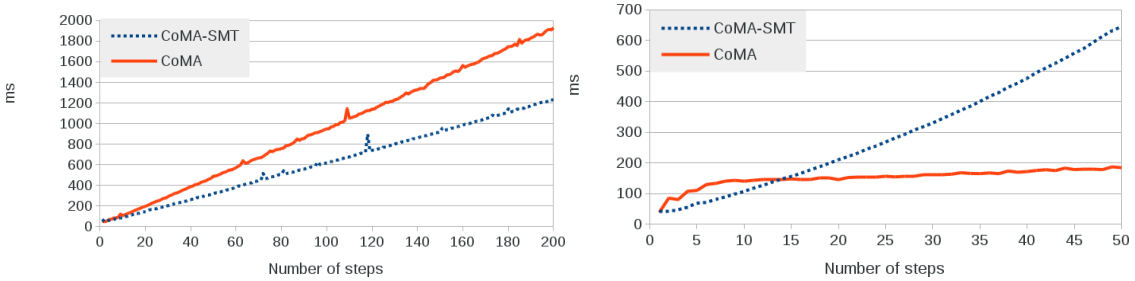
Example Let's consider a Java implementation of the Tank case study, having a pure method getLevel() returning the level of the tank, and a boolean pure method isFull () reporting whether the tank is full; the two methods are respectively linked (by means of the Java annotation `@MethodToFunction`) with ASM functions level and full. The implementation has also a changing method add(**int** quantity) (annotated with `@RunStep`) that permits to increase/decrease the tank level of a given quantity. After having detected the object instantiation and a call of method add, the monitoring framework has built the logical context as shown in Code 2. Let's suppose that the values returned by the observed elements getLevel and isFull are respectively 23 and *false*. The formula built by the framework for checking the conformance is (**and** (= level1 23)(= full1 false)).

6 Experiments

We run all the experiments on a Linux PC, Intel(R) Core(TM) i7, and 8 GB of RAM. The result of each experiment is the average of 20 runs.

6.1 Comparison with CoMA

As first experiment, we have successfully executed CoMA-SMT on all the case studies provided by the CoMA benchmarks. We have used univocal conformance, since it is the only kind of



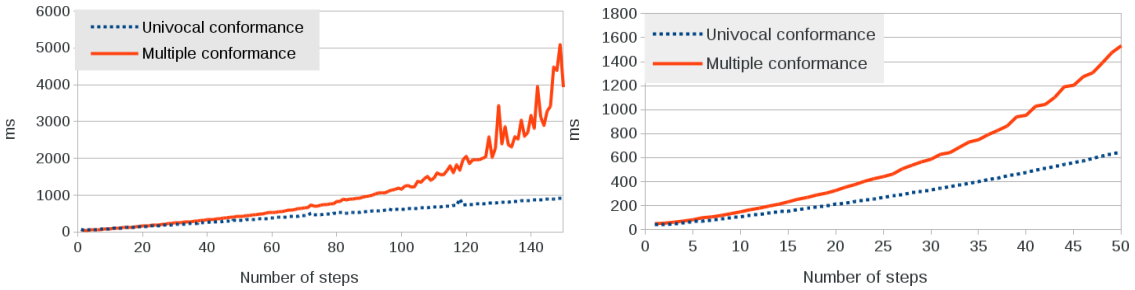
(a) Tank case study (b) Tic-tac-toe

Figure 4: Univocal conformance – CoMA-SMT and CoMA

conformance supported by CoMA.

Then, we have compared the two frameworks in terms of execution times. Fig. 4a shows the time taken for monitoring a Java implementation of the Tank case study in the presence of univocal conformance (i.e., linking both level and full functions) using the two frameworks. We have executed the Java code by calling the method add an increasing number of times. We can see that, in both frameworks, the execution time grows linearly with the number of steps. However, CoMA-SMT always performs better than CoMA, and the gain increases with the number of steps. We have observed that, for the case studies in which the number of possible next conformant states is always high, CoMA-SMT always performs better than CoMA. This means that, when the number of reachable states is high, a symbolic representation is more performant than an explicit representation.

Nonetheless, we found that, when the number of possible reachable states is low, CoMA may perform better than CoMA-SMT. We took the Tic-tac-toe case study introduced in [AGR13] as an example of nondeterministic system; it models the Tic-tac-toe game in which, at each step, the user and the computer alternatively make a move. The user makes a move by calling a given method (the actual parameters of the method represent the move coordinates); the computer makes a move by nondeterministically choosing an empty cell of the board. The linking between the implementation and the specification is based on the configuration of the board. Fig. 4b shows the comparison of the monitoring times in the two frameworks: CoMA-SMT performs better for less than around 15 moves, while, for more than 15 moves, CoMA is advantageous. Notice that, when a player has won or there is a tie, there is always only one possible next state because the game has terminated and any method invocation cannot change the board configuration; therefore, we can argue that after around 15 moves the game is very likely terminated. So, the results in Fig. 4b suggest that the explicit state representation is advantageous when the number of reachable states is low; using the symbolic representation, instead, always requires to check for satisfiability at each step, and so the monitoring time grows linearly in any case. As future work, we could devise some optimization techniques able to detect states in which the monitoring could be stopped (since no violation can be found in the future) or, at least, in which the logical context could be simplified.



(a) Tank case study (b) Tic-tac-toe

Figure 5: Univocal and multiple conformance – CoMA-SMT

6.2 Univocal and multiple conformance

We have then experimented the influence of multiple conformance in monitoring; since CoMA does not support multiple conformance, we have performed the experiments only using CoMA-SMT. We have monitored the Java implementation of the Tank case study in the presence of univocal conformance (i.e., linking both level and full functions) and in the presence of multiple conformance (i.e., linking only function full). We have executed the Java code by calling the method add an increasing number of times. Fig. 5a shows the results in terms of execution times. We have also executed the Tic-tac-toe implementation in the presence of univocal conformance (i.e., the linking is based on the board configuration) and in the presence of multiple conformance (i.e., the linking is only based on a boolean flag specifying whether the game is terminated).

As expected, the runtime monitoring is more computationally onerous in the presence of multiple conformance rather than in the presence of univocal conformance. Indeed, in univocal conformance only one state is selected when asserting the values of the Java values (line 10 in Alg. 1): therefore the solver, when checking for satisfiability, must handle a logical context in which most of the variables are fixed. In multiple conformance, instead, the solver must handle a logical context that is much more complex.

7 Related work

Extended literature exists about runtime verification [FHR13, DGR04]. Declarative specifications are more used in runtime monitoring [BLS11, CDR04, KLHN09] and they also deal with nondeterminism. Some attempts to use operational specifications exist and we relate to them; however, as far as we know, using SMT for doing runtime verification through operational specifications has never been proposed.

In [LDSW09], a *formal specification-based software monitoring system* is presented. In that approach the behavior of a concrete implementation (a Java code) is checked for compliance with a Z specification. The execution of a program is monitored by a *debugger* and the formal specification is executed in parallel with a *specification animator*. Although the approach is similar to ours in the use of an operational specification, it does not support nondeterminism.

Operational approaches based on Abstract State Machines can be found in [AGR12, BS03a].

CoMA [AGR12] has been already described in Section 2.3 and we have extensively compared to it. The approach in [BS03a] handles runtime verification of .NET programs. It uses AsmL, a dialect of ASMs, to describe the expected behavior. As CoMA, that approach only supports univocal conformance (the authors say that the choices must be *angelical*), whereas we also support multiple conformance. Moreover, the technique adopts an explicit state approach to do the conformance checking, while we use a symbolic one.

Several works handle the problem of runtime verification of concurrent programs, in which nondeterminism derives from threads' interleaving [BENS11]; instead, we focus on single thread programs that can be internally nondeterministic.

Symbolic representation of ASMs has already been presented in [VBGS09], for doing bounded model checking of AsmL models (a concrete syntax for ASMs). The main difference with our approach is that the transformation in [VBGS09] requires an intermediate representation as *model program* (i.e, a collection of guarded update rules), while we directly support ASM models.

8 Conclusions and Future work

We have proposed CoMA-SMT, an approach for runtime monitoring of nondeterministic Java programs using the ASM formal method. The approach improves an existing framework (CoMA), based on the explicit representation of ASM states. The new approach proposes a symbolic representation of the ASM computation, and exploits an SMT solver for checking the conformance between the monitored Java program and its ASM specification. While CoMA requires that, at each step, only one of the next ASM states is conformant with the observed Java state, CoMA-SMT is also able to monitor Java programs that, at a given step, are conformant with more than one ASM state. Preliminary experiments show that the symbolic approach can sometimes perform better than CoMA, but we plan to apply CoMA-SMT to more complex case studies.

As future work, we plan to devise some optimization techniques for dealing with the complexity of formulae in multiple conformance. A *context simplification* could be achieved by asserting, during monitoring, further constraints on the set of conformant states; in case of unsatisfiability, backtracking techniques should be necessary, although backtracking has never been used in runtime monitoring so far [FHR13]. Moreover, we could adapt some techniques of BMC [Bie09] for defining *stopping monitoring policies* able to detect situations in which any possible further system evolution cannot violate the conformance relation.

Bibliography

[AGR12] P. Arcaini, A. Gargantini, E. Riccobene. CoMA: Conformance Monitoring of Java Programs by Abstract State Machines. In Khurshid and Sen (eds.), *Runtime Verification*. LNCS 7186, pp. 223–238. Springer Berlin Heidelberg, 2012.

[AGR13] P. Arcaini, A. Gargantini, E. Riccobene. Combining Model-Based Testing and Runtime Monitoring for Program Testing in the Presence of Nondeterminism. In *The 9th Workshop on Advances in Model Based Testing (A-MOST 2013)*. ICSTW '13, pp. 178–187. IEEE Computer Society, Washington, DC, USA, 2013.

[BENS11] J. Burnim, T. Elmas, G. Necula, K. Sen. NDSeq: Runtime Checking for Nondeterministic Sequential Specifications of Parallel Correctness. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11, pp. 401–414. ACM, New York, NY, USA, 2011.

[Bie09] A. Biere. Bounded Model Checking. In Biere et al. (eds.), *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications 185, pp. 457–481. IOS Press, 2009.

[BLS11] A. Bauer, M. Leucker, C. Schallhart. Runtime Verification for LTL and TLTL. *ACM Transactions on Software and Methodology (TOSEM)* 20, 2011.

[BS01] P. Bekaert, E. Steegmans. Non-determinism in conceptual models. In *Proceedings of the Tenth OOPSLA Workshop on Behavioral Semantics*. Pp. 24 – 34. 2001.

[BS03a] M. Barnett, W. Schulte. Runtime verification of .NET contracts. *Journal of Systems and Software* 65(3):199–208, 2003.

[BS03b] E. Börger, R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.

[CDR04] F. Chen, M. D'Amorim, G. Roşu. A Formal Monitoring-Based Framework for Software Development and Analysis. In Davies et al. (eds.), *Formal Methods and Software Engineering*. LNCS 3308, pp. 357–372. Springer Berlin / Heidelberg, 2004.

[DGR04] N. Delgado, A. Q. Gates, S. Roach. A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools. *IEEE Transactions on Software Engineering* 30(12):859–872, 2004.

[DM06] B. Dutertre, L. de Moura. The Yices SMT solver. Technical report, SRI Available at http://yices.csl.sri.com/tool-paper.pdf, 2006.

[FHR13] Y. Falcone, K. Havelund, G. Reger. A Tutorial on Runtime Verification. In *Engineering Dependable Software Systems*. NATO Science for Peace and Security Series - D: Information and Communication Security 34, pp. 141–175. IOS Press, 2013.

[KLHN09] K. Kähkönen, J. Lampinen, K. Heljanko, I. Niemelä. The LIME Interface Specification Language and Runtime Monitoring Tool. In *Runtime Verification*. Volume 5779, chapter 7, pp. 93–100. Springer, Berlin, Heidelberg, 2009.

[Kur08] R. P. Kurshan. Verification Technology Transfer. In Grumberg and Veith (eds.), *25 Years of Model Checking*. LNCS 5000, pp. 46–64. Springer Berlin Heidelberg, 2008.

[LDSW09] H. Liang, J. Dong, J. Sun, W. Wong. Software monitoring through formal specification animation. *Innovations in Systems and Soft. Eng.* 5:231–241, 2009.

[VBGS09] M. Veanes, N. Bjørner, Y. Gurevich, W. Schulte. Symbolic Bounded Model Checking of Abstract State Machines. *Int. J. Software and Informatics* 3(2-3):149–170, 2009.