CoMA: Conformance Monitoring of Java programs by Abstract State Machines

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

[1] Dip. di Tecnologie dell'Informazione, Università degli Studi di Milano, Italy
{paolo.arcaini,elvinia.riccobene}@unimi.it
[2] Dip. di Ing. dell'Informazione e Metodi Matematici, Università di Bergamo, Italy
angelo.gargantini@unibg.it

Abstract. We present *CoMA* (Conformance Monitoring by Abstract State Machines), a specification-based approach and its supporting tool for runtime monitoring of Java software. Based on the information obtained from code execution and model simulation, the conformance of the concrete implementation is checked with respect to its formal specification given in terms of Abstract State Machines. At runtime, undesirable behaviors of the implementation, as well as incorrect specifications of the system behavior are recognized.
The technique we propose makes use of Java annotations, which link the concrete implementation to its formal model, without enriching the code with behavioral information contained only in the abstract specification. The approach fosters the separation between implementation and specification, and allows the reuse of specifications for other purposes (formal verification, simulation, model-based testing, etc.).

1 Introduction

Runtime software monitoring has been used for software fault-detection and recovery, as well as for profiling, optimization, performance analysis. Software fault detection provides evidence whether program behavior conforms with its desired or specified behavior during program execution. While other formal verification techniques, such as model checking and theorem proving, aim to ensure universal correctness of programs, the intention of runtime software-fault monitoring is to determine whether the current execution behaves correctly; thus, monitoring aims to be a lightweight verification technique that can be used to provide additional defense against failures and confidence of the system correctness.

In most approaches dealing with runtime monitoring of software, the required behavior of the system is formalized by means of correctness properties [11] (often given as temporal logic formulae) which are then translated into *monitors*. The monitor is then used to check if the properties are violated during the execution of a system. The properties specify all admissible individual executions of a system and may be expressed using a great variety of different formalisms. Some of these approaches are, for example, language oriented formalisms like extended regular expressions or tracematches by Allan et al. [1]. Temporal logic-based formalisms, which are well-known from model checking, are also very popular in

runtime verification, especially variants of linear temporal logic, such as LTL, as seen for example in [13,5].

Our approach requires a shift from a *declarative* style of monitoring to an *operational* style. Declarative specifications are used to state the desired properties of a software system by using a descriptive language. Examples of such notations are logic formulae, JML [16] or the LTL temporal logic. An *operational* specification describes the desired system behavior by providing a model implementation or model program of the system, generally executable. Examples of operational specifications are abstract automata and state machines. In [19], for instance, the specification is given in the Z language and it describes the system state and the ways in which it changes.

Specification styles (and languages) may differ in their expressiveness and very often their use depends on the preference and taste of the specifier, the availability of supporting tools, and so forth. Up to now, descriptive languages have been preferred for runtime software monitoring, while the use of operational languages has not been investigated with the same strength. Section 2 presents the current state of the art.

In this paper, we assume that the desired system behavior is given in an *operational* way by means of an Abstract State Machine (ASM), whose notation is presented in Section 3. We also assume that the implementation is a Java program and the technique we propose makes use of Java annotations. However, annotations do not contain the specification of the correct behavior (like in JML [16]) but they are used only to link the concrete implementation to its formal model, keeping separated the implementation of the system and its high-level specification. The approach has, therefore, the advantage of allowing the reuse of abstract formal specifications for other purposes, like formal verification, model simulation, model-based testing, and so forth. Indeed, the result of this work has to be also viewed towards the goal of engineering and building an environment able to support the major software life cycle activities by means of the integration of several tools that can be used for different purposes on the base of the same specification model. We are trying to achieve this goal through the open project ASMETA (ASM mETAmodeling) [3], which permits the integrated use of different tools for ASM model development and manipulation. Currently, the ASMETA tool-set allows creation, storage, interchange, Java representation, simulation, testing, scenario-based validation, model checking, and model review of ASM models for software systems[3].

In Section 4, we present the theoretical framework of *CoMA* (Conformance Monitoring by ASMs), in which we explain the relationship between the Java implementation and its ASM specification. This relationship defines syntactical links or mappings between Java and ASM elements and a semantical relation which represents the conformance. In Section 5, we introduce the actual implementation of our conformance monitoring approach which is based on Java annotations and AspectJ. A particular form of non-determinism is dealt with in Section 6. In Section 7, we discuss some advantages and limits of our approach;

[3] See the Asmeta web site http://asmeta.sourceforge.net, 2011

by means of diverse examples, we evaluate performance, expressiveness and usability of different ways (*compiled* vs *built-in*) of using CoMA, as well as w.r.t. other approaches for runtime monitoring, while Section 8 concludes the paper.

2 Related Work

Complete surveys about runtime verification can be found in [9,18,11].

Our work has been inspired by the work presented in [19], in which the authors describe a *formal specification-based software monitoring system*. In their system they check that the behavior of a concrete implementation (a Java code) complies with its formal specification (a Z model). We share with their work the fact that the concrete implementation is separated from the specification. In their monitoring system, a user of the Java program must use a specific tool to define the sequence of methods to execute. Therefore, their monitoring system is useful at testing and debugging time, but can not be used in the deployed system in which the monitoring system should be hidden to the final user. The final user, indeed, could be different from the developer of the code: he could be a normal user who wants to execute the code or another developer who wants to reuse the code. In both cases the user should be unaware of the formal specification; he could only be aware that some kind of monitoring is performed. In our system, instead, a developer can deploy a Java code linked with its formal specification. The final user can use the monitored code without knowing anything about the formal specification; the only thing that he must know is that, if he wants to enable the monitoring to the code, he must execute it with AspectJ.

Monitored-oriented programming (MOP) [7] permits to execute runtime monitoring by means of annotating the code with formal property specifications. The specifications can be written in any formalism for which a logic plug-in has been developed (LTL, ERE, JML, ...). The formal specifications are translated (in two steps) in the target programming language. The obtained monitoring code can be used in an *in-line* mode in which the monitoring code is placed in the monitored program, and in an *out-line* mode in which it is used to check traces recorded by adequate probes. Similarly to us, they use AspectJ to weave the monitoring code into the monitored code; in particular AspectJ gives them the ability to execute the monitoring code before or after some methods invocations. A similar approach is taken by *Lime* [15]. This tool permits to monitor the invocations of the methods of an interface by defining *pre* and *post* conditions, called *call specifications* (CS) and *return specifications* (RS). Specifications can be written as past/future LTL formulas, as regular expressions and as nondeterministic finite automata. The specifications are then translated into deterministic finite state automata encoded in Java that function as observers. AspectJ is used to weave the observer code into the original program that is being tested.

Another approach that uses ASMs as formal specification for system monitoring purpose is presented in [4]. That approach shares with ours many common features as using operational specifications (called model programs) and dealing with method calls ordering. However, the approach is mainly applied to specify all of the traditional design-by-contract concepts of pre- and post-conditions

and invariants. The technological framework is completely different, since .NET components are considered.

Different approaches exist for system monitoring that are based on runtime verification of temporal properties. In [5], traces of programs are examined in order to check if they satisfy some temporal properties expressed in LTL_3, a linear-time temporal logic designed for runtime verification.

3 Abstract State Machines

Abstract State Machines (ASMs), whose complete presentation can be found in [6], 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 non-determinism (existential quantification `choose`) and unrestricted synchronous parallelism (universal quantification `forall`).

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

Functions may 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 executing its (unique) *main rule*. An ASM can have more than one *initial state*. It is possible to specify state *invariants*. Because of the non-determinism of the choose rule and of moves of the environment, an ASM can have several different runs starting in the same initial state.

Code in Fig. 1 reports the ASM specification of a counter limited to 10 (according to the invariant) and initialized to the monitored value *initValue*; *counter* and *initValue* are both 0-ary functions.

The ASMETA tool set is a set of tools around the ASMs [3]. They can assist the user in developing specifications and proving model correctness by checking state invariants and temporal logic properties. For instance, the invariant in Fig. 1 can be proved invalid if *initValue* is greater than 10 by model checking.

Among the ASMETA tools, those involved in our conformance analysis process are: the textual notation *AsmetaL*, used to encode fragments of ASM models, and the simulator *AsmetaS*, used to execute ASM models.

```
asm counterMax10
signature:        dynamic controlled counter: Integer
                  dynamic monitored initValue: Integer

definitions:      invariant inv_a over counter: counter <= 10

  main rule r_Main = if  counter < 10  then    counter := counter + 1  endif

// initizialize   counter
default init s0:      function counter = initValue
```

Fig. 1. ASM Counter in AsmetaL

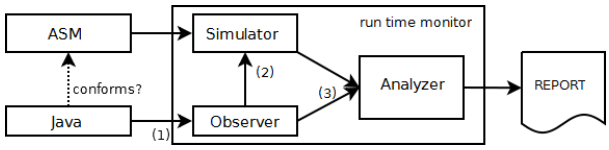


Fig. 2. The CoMA runtime monitor for Java

4 Runtime conformance monitoring based on ASMs

A *runtime software-fault monitor*, or simply a *monitor*, is a system that observes and analyzes the states of an executing software system. The monitor checks the correctness of the system behavior by comparing an *observed* state of the system with an *expected* state. The expected behavior is generally provided in terms of a formal specification. We here intend runtime monitoring as conformance analysis at runtime and we propose *CoMA*, runtime *Conformance Monitoring* of Java code *by ASM specifications*.

The CoMA monitor allows *online* monitoring, namely it considers executions in an incremental fashion. It takes as input an executing Java software system and an ASM formal model. The monitor observes the behavior of the Java system and determines its correctness w.r.t. the ASM specification working as an oracle of the expected behavior. While the software system is executing, the monitor checks conformance between the observed state and the expected state.

As shown in Fig. 2, the monitor is, therefore, composed of: an *observer* that evaluates when the Java (observed) state is changed (1), and leads the abstract ASM to perform a machine step (2), and an *analyzer* that evaluates the step conformance between the Java execution and the ASM behavior (3). When the monitor detects a violation of conformance, it reports the error. It can also produce a trace in form of couterexample, which may be useful for debugging. Note that the use of CoMA can be twofold: also faults in the specification can be discovered by monitoring software. For instance, by analysing and re-executing counterexamples, faults in the model can be exposed.

In the following sections, we introduce the theoretical basis of our monitoring system. We, therefore, formally define what is an observed Java state, how to establish a conformance relation between Java and ASM states, and, therefore, step conformance and runtime conformance between Java and ASM executions.

4.1 Observable Java elements and their link with ASM entities

In order to mathematically represent a class and the state of its objects, we introduce the following definitions.

Definition 1. ***Class*** *A* class C *is a tuple* $\langle c, f, m\rangle$ *where* c *denotes the non-empty set of constructors,* f *is the set of all the fields,* m *is the set of methods.*

We denote the public fields of C as f^{pub} while the public methods are denoted as m^{pub}. Among the methods of a class, we distinguish also the *pure* methods:

Definition 2. ***Pure method*** *Pure methods* m_{pure} *are side effect free, with respect to the object/program state. They return a value but do not assign values to fields.* m^{pub}_{pure} *denotes the set of all pure public methods in* m*.*

Pure methods [10] are useful and common specification constructs. By marking a method as pure, the specifier indicates that it can be treated as a function of the state (as in JML [16]). We consider only pure methods without arguments.

Definition 3. ***Virtual State*** *Given a class* $C = \langle c, f, m\rangle$*, the* virtual state*,* $VS(C)$*, is given by* $VS(C) = f^{pub} \cup m^{pub}_{pure}$*.*

Definition 4. ***Observed State*** *We define* observed state*,* $OS(C) \subseteq VS(C)$*, as the subset of the virtual state consisting of all public fields, and pure public methods of the class* C *the user wants to observe.*

Therefore, $OS(C)$ is the set of Java elements monitored at runtime. For convenience, we can see $OS(C) = OF(C) \cup OM(C)$ to distinguish between the subset of *observed fields* $OF(C)$ and the subset of *observed methods* $OM(C)$ of $OS(C)$. Note that $OF(C) \subseteq f^{pub}$ and $OM(C) \subseteq m^{pub}_{pure}$. The (returned) values of the elements of $OS(C)$ can change by executing any not pure method (in $m_{\neg pure} = m - m_{pure}$).

Definition 5. ***Changing Method*** *Given a Java class* C*, we define* changing methods*,* $changingMethods(C) \subseteq m_{\neg pure}$*, all methods of* C *whose execution is responsible for changing an element of* $OS(C)$ *and that the user wants to observe.*

Linking observable Java elements to ASM entities. In order to be runtime monitored, a Java class $C = \langle c, f, m\rangle$ should have a corresponding ASM model, ASM_C, abstractly specifying the behavior of an instance of the class C.

Observable elements of a class C must be linked to the dynamic functions $Funcs\_ASM_C$ of the ASM model ASM_C. The function

$$link : OS(C) \rightarrow Funcs\_ASM_C \tag{1}$$

yields the set of the ASM dynamic functions linked to the observable Java elements of C. The function $link$ is not surjective because there are ASM dynamic functions that are not used in the conformance analysis.

Execution step in Java and ASM. In order to define a step of a Java class execution, we rely on the concept of *machine step* and *last state* of execution sequence defined in the Unifying Theories of Programming (UTP) [14]. A Java *state* of an instance of a class C is the set of the actual values of its fields.

Definition 6. ***Java Step*** *Let m be a method of a Java class. A Java step is defined as the relation (s,m,s') where s is the starting state of the execution of m and s' the last state of this execution.*

Definition 7. ***Change Step*** *Let C be a Java class. A* change step *is defined as a Java step for $m \in changingMethods(C)$.*

Note that, choosing the granularity of the Java step at the level of class method and not at the level of single assignment, allows the designer to tune the desired granularity of the monitoring.

ASM *state* and ASM computation *step* have been defined in Section 3.

4.2 State Conformance, Step Conformance and Run Conformance

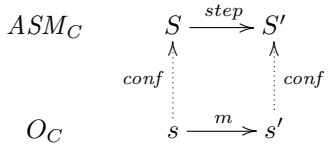
We have formally related a Java class and the execution of a Java class instance with the corresponding abstract ASM model and relative execution(s). In the following definitions, let C be a Java class, O_C any instance of C, and ASM_C its corresponding ASM abstract model.

We assume that the function $val_{Java}(e, s)$ yields the value of a Java element $e \in VS(C)$ of C in a given state s of O_C, while the value of an ASM function l in a state S is given by $val_{ASM}(l, S)$. Moreover we assume that there exists a conformance $\stackrel{conf}{=}$ relation among Java and ASM values [2].

Definition 8. ***State Conformance*** *We say that* 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 locations in ASM_C linked to them; i.e.*

$$conf(s, S) \equiv \forall e \in OS(C) : val_{Java}(e, s) \stackrel{conf}{=} val_{ASM}(link(e), S) \qquad (2)$$

Definition 9. ***Step Conformance*** *We say that* a change step (s, m, s') *of an instance O_C, with m a method of C,* conforms with a step (S, S') *of ASM_C if $conf(s, S) \wedge conf(s', S')$.*



Definition 10. ***Runtime Conformance*** *Given an* observed computation *of a Java instance O_C, we say that C is* runtime conforming *to its specification ASM_C if the following conditions hold:*

– *the initial state s_0 of the computation of O_C* conforms *to the initial state S_0 of the computation of ASM_C, i.e. it yields $conf(s_0, S_0)$;*

– *every observed change step (s, m, s') with s the current state of O_C,* conforms *with the step (S, S') of ASM_C with S the current state of ASM_C;*

– *no specification invariant of ASM_C is ever violated.*

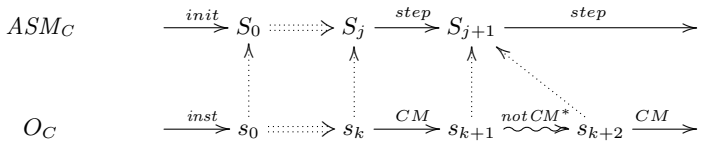


Fig. 3. Runtime conformance

Fig. 3 depicts the co-simulation of an instance O_C and its specification ASM_C. Def. 10 requires conformance between s_0 and S_0. If O_C is in state s_k, executes a change method CM, and moves to state s_{k+1}, then s_k must conform to the current ASM state S_j and s_{k+1} must conform to the next ASM state S_{j+1}. Then, no conformance check is performed until the next observed state s_{k+2} when a changing method is invoked again. Note that the final state of a Java change step and the initial state of the subsequent change step are both *state conforming* to the same abstract state of the ASM.

5 Monitor Implementation

We here describe how CoMA works. We provide technical details on how the runtime monitor is implemented by exploiting the mechanism of Java annotations to link observable Java elements to corresponding ASM entities, and AspectJ to observe code execution and establish conformance relation.

5.1 Using Java Annotations

Java annotations are meta-data tags that can be used to add some information to code elements as class declarations, field declarations, etc.

In addition to the standard ones, annotations can be defined by the user similarly as classes. For our purposes we have defined a set of annotations in order to link the Java code to its abstract specification. The retention policy, i.e. the way to signal how and when the annotation can be accessed, of all of our annotations is *runtime* – annotations can be read by the compiler and by the monitor at run-time through reflection.

In order to link a Java class C with its corresponding ASM model ASM_C, the class must be annotated with the `@Asm` annotation having the path of the ASM model as string attribute. Fig. 4 reports the Java class `Counter` linked to its ASM specification (see Fig. 1).

To establish the mapping defined by the function *link*, we annotate each observed field $f \in OF(C)$ by `@FieldToFunction`, and each observed method $m \in OM(C)$ by `@MethodToFunction`; both these annotations have a string attribute yielding the name of the corresponding ASM function. In the example, the Java field `counter` and the Java pure method `getCounter` are both linked to the *counter* ASM function.

All methods of $changingMethods(C)$ are annotated with the `@RunStep`. In the example, the observed method is `inc()` that simply increments the counter.

Finally, the user has to decide the starting point of the monitoring. The annotation `@StartMonitoring` is used to select a proper (not empty) subset

of constructors[4]. All or some constructor parameters (if any) can be annotated with the `@Init` annotation that permits to link a parameter with a monitored function (i.e. only read, as events provided by the environment) of the ASM model. This allows initializing the ASM model with the same values used to create the Java instance. In the example there is just one constructor whose parameter is linked with the ASM monitored function *initValue* which fixes the initial value of the counter (see the specification in Fig. 1).

Our use of the annotation mechanism requires a very limited code modification and differs from that usually exploited in other approaches for system monitoring. Usually annotations are used to enrich the code with extra formal specifications to obtain behavioral information about the target program [7,15]. This leads to the lack of separation between the implementation of the system and its high-level requirements specification. In our approach, the few annotations are only used to link the code to its specification, but keeping them separate. Furthermore, annotations are statically type checked and since the annotations are read reflectively at runtime, the monitoring setup can be carried out very easily. This is much more convenient than inserting special comments (like JML) and writing our own parser for them. Moreover, Java annotations make the links more robust when code refactoring is applied. Our approach fosters the reuse of specifications when code changes.

```
@Asm("counterMax10.asm")
class Counter {

@FieldToFunction("counter")
public int counter;

@StartMonitoring
Counter(@Init("initValue") int x){counter = x;}

@MethodToFunction("counter")
public int getCounter(){ return counter;}

@RunStep
public void inc(){ counter ++; }}
```

Fig. 4. Java Counter Annotated

5.2 Runtime monitor and AspectJ

The *runtime monitor* (see Fig. 2) is implemented through the facilities of AspectJ that permits to easily observe the execution of Java objects. AspectJ allows programmers to define special constructs called *aspects*.

(1) Observer By means of an aspect, AspectJ allows to specify different *pointcuts*, that are points of the program execution one wants to capture; for each pointcut it is possible to specify an *advice*, that is the actions that must be executed when a pointcut is reached. AspectJ permits to specify when to execute the *advice*: *before* or *after* the execution of the code specified by the pointcut.

The CoMA tool supports two different ways, *built-in* and *compiled*, of developing an aspect.

Built-in In this approach there is just one aspect that permits to monitor all the objects of the classes that must be monitored: (i) the pointcuts are general enough to capture the instantiations and the method executions of all the objects

[4] We do not consider the default constructor. If the class does not have any constructor, the user has to specify an empty constructor and annotate it with `@StartMonitoring`.

that must be monitored; (ii) the advices are able to dynamically inspect the Java and the ASM state in order to do the conformance checking.

The main advantage of this approach is that the developer does not have to care about building the aspect: after having written the Java class and the ASM specification, and after having linked them properly, he/she can execute the code immediately.

The main disadvantage of this approach is that, since the aspects are very general, they introduce an overhead in the pointcuts and in the advices that execute the conformance checking. For instance, the pointcuts to detect the creation of an observed object and to capture the execution of a *changing method* (we do not consider changing methods that are executed in the scope of other changing methods) are reported below.

```
pointcut objCreated(): call(@StartMonitoring *.new(..));
pointcut runStepCalled(): call(@RunStep * *.*(..))
                          && !cflowbelow(call(@RunStep * *.*(..)));
```

In order to read the values of the fields that are monitored, we have implemented two techniques: (i) reading them through reflection at the beginning and at the end of the execution of a changing method; (ii) using the AspectJ pointcut *set* in order to capture all their updates.

The main advantage of using reflection is that we can get their values just once for each changing method execution; using the *set* pointcut, instead, every time a monitored field is updated we collect its value: if a field is updated frequently (e.g. in a loop), using the *set* pointcut the performances of the monitoring module can get worse. However, the *set* pointcut can read private fields without programmatically changing their visibility.

Compiled In this approach, for each Java class that must be monitored, a suitable aspect is built. The main advantage of this approach is that the aspect definitions (pointcuts and advices) can be more precise (e.g. the pointcut that captures the execution of the changing methods can specify exactly the methods whose execution must be captured: in the *built-in* approach, instead, we must capture all the methods annotated with `@RunStep`). The main disadvantage is that the developer, before running his code, must build the aspect: if the Java code and/or the ASM specification change, the aspect may need to be rebuilt. For instance, the pointcuts for the `CounterDec` class are:

```
pointcut objCreated(): call(CountercDec.new(..));
pointcut methodCalled(): call(@RunStep public void CountercDec.inc()) ||
                         call(@RunStep public void CountercDec.dec());
pointcut runStepCalled(CountercDec target): methodCalled() &&
                                !cflowbelow(methodCalled()) && target(target);
```

(2) Simulator Upon a Java change step signaled by the observer, the *simulator* performs an ASM step by AsmetaS [12]. Before a change step, an advice reads the values of the *monitored* fields, sets the ASM monitored functions, and executes a state conformance check ($conf(s, S)$ in Def. 9). After a change step,

```
@Asm("CounterDec.asm")
class CounterDec {
@FieldToLocation("counter")
public int counter;

@RunStep(setFunction = "action", toValue = "dec")
public void dec() {
    counter --; }

@RunStep(setFunction = "action", toValue = "inc")
public void inc() {
    counter ++; } }
```

```
asm CounterDec
signature:
    controlled counter: Integer
    monitored action: String

definitions:
    main rule r_Main =
        if action = "inc" then
            counter := counter + 1
        else if action = "dec" then
            counter := counter -1
        endif endif
```

Fig. 5. Counter with decrement

another advice simulates a step of the ASM and forces the Analyzer to check again the state conformance ($conf(s', S')$ in Def. 9).

(3) Analyzer The *analyzer* compares the Java and the ASM state. To check state conformance (see Def. 8), we have implemented the conformance relation $\overset{conf}{=}$ among Java and ASM values as a string comparison. Therefore, the Java and the ASM values are both transformed into strings for comparison.

6 Dealing with multiple changing methods

Definition 10 is adequate for runs where the next state of a Java class C and of its specification ASM_C are unique. Thus, nondeterminism is limited to monitored quantities, which, once not deterministically fixed by the environment, make the evolution of the system deterministic. In this Section, we extend our conceptual framework to deal with a limited form of nondeterminism due to the presence of more than one changing method, each of which takes C to a possible different correct next state in a deterministic way; however, the choice of the changing method that causes a change step is non-deterministic.

In this case, the observer must signal to the ASM under simulation, which step has been chosen by the program. To this scope, we introduce two fields in the `@RunStep` annotation: *setFunction* permits to specify the name of a monitored function of the ASM model, and *toValue* the value to whom it must be set.

In the Java code in Fig. 5, the `@RunStep` annotations of the changing methods *dec()* and *inc()* specify that the monitored function *action* must be set, respectively, to *dec* and *inc*.

7 Evaluation

In order to assess the viability of our approach, we have taken several examples in literature and checked whether we were able to apply our approach to existing runtime case studies, including the Railroad Gate [9], the Initialization Fiasco problem [5], a robotic assembly system [19], the Knight's Tour problem [20]. We have written the Java code, if not available, and their ASM specifications (see [2] for details). We applied also CoMA to several Java programs borrowed from

	Java	JML	JavaMOP		CoMA			
					AsmetaS	compiled	set	reflection
Counter	4	280	(FSM)	109	4837	+ 783	+ 825	+ 898
Iterator	8	N/A	(FSM)	91	866306	+ 1439	+ 1812	+ 1820
Initialization Fiasco	7	N/A	(LTL)	72	870719	+1914	+ 2235	+ 2366

Table 1. Execution time in the experiments (in secs)

JavaMOP [7], like `Iterator` and `FileWriter`. Overall we found our approach applicable to all the considered case studies.

Execution time In order to evaluate the runtime overhead of our approach, we have considered three examples, the *Counter*, the *Iterator* and the *Initialization Fiasco*, and we have monitored them with CoMA, JavaMOP (FSM or LTL), and JML, when applicable. A comparison with [19] is not possible. They use, like CoMA, interpretation of formal specifications, but their tool is not available and no time data are published.

Table 1 reports the average of the time required for 20 runs with 100 instances running in parallel for 1000 steps. JML cannot be used with the *Iterator* and the *Initialization Fiasco*. For the CoMA, Table 1 reports the time for the three kinds of aspects described in Section 5.2; we have divided the overall time between the time taken by the simulator (column AsmetaS) and the time taken by the code under analysis and the monitor module.

It is apparent that most of the time is taken by the simulator, which is based on the Eclipse Modelling Framework, widely uses reflection and visitor design patterns, and has never been optimized for performance. On the average, using reflection or using the *set* pointcut is almost equivalent. However, *set* pointcuts may perform worst when an observed field is updated frequently. Instead, *compiled* pointcuts provide the best results.

Although our approach seems not competitive with others in terms of time overhead, we believe that it provides several advantages (explained below) and it can be used when performances are not critical. As a future work, we plan to decrease the running time of the simulation by translating the ASM machine directly into Java code (similarly of what is done in JavaMOP and in Lime). However, encoding ASM into Java would require the semantic correctness proof of the translation. Approaches translating to Java/AspectJ are more efficient but the preservation of the semantics by the translation may become an issue.

At the current development stage of our framework, we have been more interested in assessing the usability and expressiveness of our approach than its time performance.

Usability and expressiveness Although any comparison of our approach with others in terms of usability and expressiveness may be disputable, since it may depend on the expertise and taste of the user, some general considerations follow.

In comparison with JML, CoMA can be used to express the behavior of a single method call and also the interaction among calls, while JML concentrates on single methods. There exist JML extensions that allow the specification of temporal aspects of Java interfaces (like LIME [15] and trace assertions of Jass).

Another difference is that CoMA has a model separated from the implementation, while JML follows a unique model paradigm in which the code itself contains its specification. The advantage of CoMA is that the specification can exist even before its implementation and can be used for several preliminary activities (like model simulation, model review, and formal verification).

The expressiveness of CoMA is greater than approaches using plain FSMs, since ASMs can have infinite states and can be viewed as pseudo-code over abstract data type. In many approaches, like in JavaMOP and in JavaMAC (which uses automata with auxiliary variables) [17], FSMs are enriched with state variables. For instance, the FSM for the counter in JavaMOP becomes:

```
CheckCounter(Counter c) {
// counter value
int count = 0;
// inc call  event
event inc before(Counter c): call(* Counter.inc()) && target(c) {count ++;}
// error event
event err after(Counter c): call(* Counter.inc()) && target(c) &&
                    condition(c.getCounter() != count) {}
// the FSM
fsm: safe [inc -> safe    err -> error]        error []

@error { System.out.println("Counter not incremented"); }}
```

Since JavaMOP specifications are compiled into AspectJ, JavaMOP can include and use all the power of AspectJ. However, we believe that mixing implementation and specification notations may encourage the user to insert implementation details in the specification at the expense of abstractness. An important feature of our methodology is the clear separation between the monitored implementation and the high level specification also in terms of notation, as in [17,19].

Comparison with property-based approaches An objective comparison with approaches based on the use of properties is more questionable. In this paper, we assume that the specification is given in operational style instead of the more classical declarative style. There has been an endless debate about which style fits better the designer needs: some argue that with an operational style the designers tend to insert implementation details in the abstract specifications, others observe that practitioners feel uncomfortable with declarative notations like temporal logics. The scope of this paper is to provide evidence that also abstract operational notation can be effectively used for runtime monitoring. Sometimes, operational specifications are easier to write and understand; other times, declarative specifications are preferable. For instance, LTL and PLTL can describe correct sequences of method calls with ease. The correct order of calls for an `Iterator`, is specified by the following PLTL formula: $\Box(next \implies \odot hasNext)$, where the operator $\odot$ means in the previous time step. However, properties about states are more difficult (and sometimes impossible) to write. For instance, the fact that an unbounded counter is correctly incremented is not expressible by LTL. Indeed, LTL does not allow variable quantifiers and,

therefore, formulas like $\forall x\ \Box(counter = x \implies \bigcirc(counter = x + 1))$ are incorrect.

8 Conclusions and Future work

We have presented and briefly evaluated CoMA, a framework for runtime conformance monitoring of Java code with respect to its specification given in terms of Abstract State Machines. The source code must be annotated to link Java elements to ASM elements. The CoMA monitor, based on AspectJ, checks runtime conformance between Java executions and ASM specifications. While the software executes, the monitor simulates step by step the ASM specification and checks the state conformance.

Our approach has some limits. The use of an operational specification can lead the designer into inserting implementation details in the specification. Since each class is linked to its specification, monitoring safety properties involving collections of two or more objects [8] is not possible, but we plan to extend CoMA to support also these scenarios. We deal only with restricted forms of nondeterminism, but we are working on supporting more generic forms of it [2]. Monitoring real time requirements seems problematic: we believe that a monitored function *time* may model the real time and would allow its measurement, but further experiences are needed and the runtime overhead may be an issue. Since CoMA currently checks conformance by interpreting the ASM, it performs much slower than other approaches. We plan to optimize the monitoring process to reduce the temporal overhead.

Despite these limits, we believe that our approach presents a viable technique for checking conformance of an implementation (as Java program) with respect to its formal and abstract operational specification (as ASM). Although it is difficult to give a definitive evaluation, we believe that the operational style should be appealing for those preferring executable models instead of properties and that an operational abstract style of describing system behavior may be more easy to write and understand. In our approach, specifications are developed independently from the implementations and they are linked by Java annotations which however contain minimal behavioral information.

There are some advantages not related to runtime verification in using executable specifications (as also discussed in [4]), including that a specification can be executed in isolation, even before its implementation exists. CoMA fosters the reuse of specifications for further purposes thanks to its integration in the ASMETA framework [3], which supports editing, type checking, simulation, review, formal verification, and test case generation for ASMs.

References

1. C. Allan, P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In R. E. Johnson and R. P. Gabriel, editors, *OOPSLA*, pages 345–364, 2005.
2. P. Arcaini, A. Gargantini, and E. Riccobene. Runtime monitoring of Java programs by Abstract State Machines. TR 131, DTI Dept., Univ. of Milan, 2010.

3. P. Arcaini, A. Gargantini, E. Riccobene, and P. Scandurra. A model-driven process for engineering a toolset for a formal method. *Softw., Pract. Exper.*, 41(2):155–166, 2011.
4. M. Barnett and W. Schulte. Runtime verification of .NET contracts. *The Journal of Systems and Software*, 65(3):199–208, Mar. 2003.
5. A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software and Methodology (TOSEM)*, 20, 2011.
6. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.
7. F. Chen, M. D'Amorim, and G. Roşu. A formal monitoring-based framework for software development and analysis. In J. Davies, W. Schulte, and M. Barnett, editors, *Formal Methods and Software Engineering*, volume 3308 of *Lecture Notes in Computer Science*, pages 357–372. Springer Berlin / Heidelberg, 2004.
8. F. Chen and G. Roşu. Mop: an efficient and generic runtime verification framework. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications - OOPSLA '07*, page 569, Montreal, Quebec, Canada, 2007.
9. S. Colin and L. Mariani. Run-time verification. In M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 525–555. Springer Berlin / Heidelberg, 2005.
10. A. Darvas and K. R. M. Leino. Practical reasoning about invocations and implementations of pure methods. In *Proceedings of the 10th international conference on Fundamental approaches to software engineering*, FASE'07, pages 336–351, Berlin, Heidelberg, 2007. Springer-Verlag.
11. N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, 30(12):859–872, 2004.
12. A. Gargantini, E. Riccobene, and P. Scandurra. A metamodel-based language and a simulation engine for Abstract State Machines. *Journal of Universal Computer Science (JUCS)*, 14(12):1949–1983, 2008.
13. K. Havelund and G. Roşu. Efficient monitoring of safety properties. *Int. J. Softw. Tools Technol. Transf.*, 6:158–173, August 2004.
14. C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall International, Englewood Cliffs, NJ, 1998.
15. K. Kähkönen, J. Lampinen, K. Heljanko, and I. Niemelä. The LIME interface specification language and runtime monitoring tool. In S. Bensalem and D. A. Peled, editors, *Runtime Verification*, volume 5779, chapter 7, pages 93–100. Springer, Berlin, Heidelberg, 2009.
16. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31:1–38, May 2006.
17. I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications. In *Parallel and Distributed Processing Techniques and Applications*, pages 279–287, 1999.
18. M. Leucker and C. Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293 – 303, 2009.
19. H. Liang, J. Dong, J. Sun, and W. Wong. Software monitoring through formal specification animation. *Innovations in Systems and Soft. Eng.*, 5:231–241, 2009.
20. E. W. Weisstein. Knight's tour. from MathWorld–A Wolfram Web Resource.