Runtime monitoring of Java programs by Abstract State Machines

Paolo Arcaini
Università degli Studi di Milano
paolo.arcaini@unimi.it

Angelo Gargantini
Università degli Studi di Bergamo
angelo.gargantini@unibg.it

Elvinia Riccobene
Università degli Studi di Milano
elvinia.riccobene@unimi.it

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 that program behavior conforms or does not conform 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.

Runtime monitoring should be considered when you cannot execute an exhaustive verification of your system (in most of the cases); for example, proving a security property that your system never reaches a dangerous state could require too much time depending on the size of your system. On the contrary, testing could be considered not enough trustworthy, in particular in critical systems. Moreover there could be some information that are available only at run-time, or the behaviour of the system could depend on the (not reproducible) environment where the system runs. Finally it could also be possible that, nevertheless the system has been tested and maybe also proved correct, the developer wants to be sure that the system does not violate some given properties during its execution. Extending the thought of Ed Brinksma, expressed during the 2009 keynote at the Dutch Testing Day and Testcom/FATES, on the relationship between verification and testing[1], we could ask:

Who would want to fly in an airplane with software proved correct, hardly tested, but not monitored at run-time?

In most approaches dealing with run time monitoring of software, the required behavior of the system is formalized by means of correctness properties [8] (often given as temporal logic formulae) which are then translated into *monitors*. The monitor is then used to check the execution of a system if the properties are violated. The properties specify all admissible individual executions of a system and may be expressed using a great variety of different formalisms. These range from, for example, language oriented formalisms like extended regular expressions or tracematches by the AspectJ team. 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 [10, 3]. For an overview about runtime verification techniques and tools, please refer to [14, 8, 7].

[1] *Who would want to fly in an airplane with software proved correct, but not tested?*

In this paper, we assume that the desired behavior of the system is given by means of Abstract State Machines (ASMs) which specify the behavior of the system in an *operational* way: they describe the desired changes of the system state when some particular input conditions occur. A similar approach is taken also in [15], where the specification is given in the Z language, which describes the system states and the ways in which the states can be changed. Note that our approach requires a shift from a *declarative* style of monitoring (based on properties) to an *operational* style, based on ASMs. An *operational* specification describes the desired behavior by providing a model implementation or model program of the system, generally executable. Examples of operational specifications are abstract automata and state machines. Another different specification style is through *descriptive* specifications, which are used to state the desired properties of a software component by using a declarative language. Examples of such notations are logic formulae, JML [13] or the LTL temporal logic. Different 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 support tools, etc. Up to now, descriptive languages have been preferred for run time software monitoring, while the use of operational languages has not been investigated with the same strength.

In this paper we assume that the implementation is a Java program, while the specification is an Abstract State Machine (ASM), whose notation is presented in section 2. In section 3 we present out theoretical framework, in which we explain the relationship between the Java implementation and the ASM specification. This relationship defines syntactical links or mappings between Java and ASM elements and a semantical relation which represents the conformance. The important issue of non-determinism is tackled in section 4. In section 5 we introduce the actual implementation of our monitoring approach which is based on Java annotations and AspectJ. In section 6 we show how our monitoring system can be used in practice through the illustration of some examples.

2 Abstract State Machines

Abstract State Machines (ASMs), whose complete presentation can be found in [5], are an extension of FSMs [4]. Machine *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 $f(t_1, \ldots, t_n) := t$ which are simultaneously executed when *Condition* is true. f is an arbitrary n-ary function and $t_1, \ldots, t_n, t$ are first-order terms.

To fire this rule in a state s_i, $i \geq 0$, all terms $t_1, \ldots, t_n, t$ are evaluated at s_i to their values, say $v_1, \ldots, v_n, v$, then the value of $f(v_1, \ldots, v_n)$ is updated to v, which represents the value of $f(v_1, \ldots, v_n)$ in the next state s_{i+1}. Such pairs of a function name f, which is fixed by the signature, and an optional argument $(v_1, \ldots, v_n)$, 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 $s_0, s_1, \ldots, s_n, \ldots$ of states of the machine, where s_0 is an initial state and each s_{n+1} is obtained from s_n by 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.

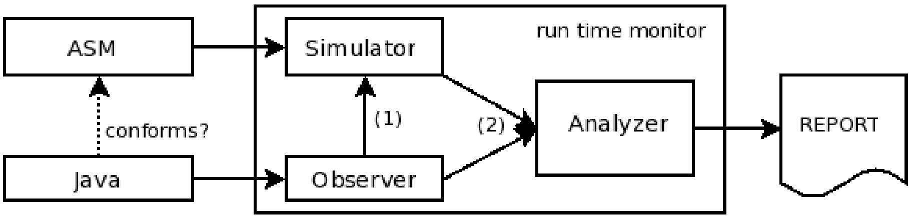

Figure 1: A runtime monitor for Java

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 monitoring process are: the textual notation *AsmetaL*, used to encode fragments of ASM models, and the simulator *AsmetaS*, used to execute ASM models.

3 Runtime monitoring based on ASM specifications

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 correctness of the system behavior by comparing an *observed* state of the system with an *expected* state. The expected behavior is generally provided in term of a formal specification. In this paper, we intend runtime monitoring as conformance analysis at runtime.

Depending if the monitor is designed to consider executions in an incremental fashion or to work on a (finite set of) recorded execution(s), a monitor allows *online monitoring* or *offline monitoring*, respectively [14].

The monitor we propose, which allows online monitoring, takes in input an executing Java software system and an ASM formal model written in AsmetaL. 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. 1, the monitor is, therefore, composed of: an *observer* that evaluates when the Java (observed) state is changed, and leads the abstract ASM to perform a machine step, and an *analyzer* that evaluates the step conformance between the Java execution and the ASM behavior. When a violation of conformance is detected, it quits the monitoring upon the user request.

We here focus only on monitors that are used to detect faults, which occur during the execution of software and results in an incorrect state. Other monitoring systems extend this capability by diagnosing faults, i.e., providing information to the user that will aid the user in understanding the cause of the fault and assist the system in recovering from faults by directing the system to a correct state (forward recovery) or by reverting to a state known to be correct (backward recovery).

In the following sections, we pose the theoretical bases 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. Initially, we assume that either the Java program and the ASM model are deterministic. Non-determinism is dealt in section 4.

3.1 Observable Java elements and their link with ASM entities

In order to mathematically represent a Java 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 (as in JML [13]):

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

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 *observed fields* $OF(C)$ and the subset of *observed methods* $OM(C)$ of $OS(C)$. Note that $OF(C) \subseteq f^{pub}$ and $OM(C) \subseteq m^{pub}_{pure}$. Elements of $OS(C)$ can change by effect of the class method computation. Only methods in $m_{\neg pure}$ may change the program state.

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 of changing $OS(C)$ and that the user wants to observe.*

3.1.1 Linking observable Java elements to ASM entities

In order to be run-time 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. Moreover, the function is not injective, because more than one Java field or method can be linked to the same ASM function.

3.2 Common representation of Java and ASM values

In order to be compared, Java values and ASM values must be translated into a common format. Let us consider a Java class C and the corresponding model $ASM(C)$. The following functions, *cfJ* and *cfA*, provide string representations of, respectively, Java and ASM values, in a given state.

$$\begin{aligned} cfJ :& OS(C) \times S_{Java} \rightarrow String \\ cfA :& ASM(C) \times S_{ASM} \rightarrow String \end{aligned}$$

The first function *cfJ* exploits the built-in Java function *toString* of the class Object. A similar function is provided by the ASMETA simulator to convert type values into strings for ASM models.

cfJ function Let e be a Java field or non-void method whose type is t. Let v_{Java^j} be the value of e in state s^j_{Java}.

The behaviour of the *cfJ* function depends on the type t; if t is

- a primitive type, an array, a String, a List, or a Set, its string representation in state s^j_{Java} is
$$cfJ(e, s^j_{Java}) = v^j_{Java}$$
- a $Map < T, E >$ type[2], we can identify with $\{k^1_{Java^j}, \ldots, k^m_{Java^j}\}$ the keys (of type T) of the map in the state s^j_{Java}, and $\{v^1_{Java^j}, \ldots, v^m_{Java^j}\}$ their corresponding values. The representation of the map in state s^j_{Java} is
$$cfJ(e, s^j_{Java}) = \{k^1_{Java^j} \to v^1_{Java^j}, \ldots, k^m_{Java^j} \to v^m_{Java^j}\}$$
where parentheses, commas and arrows must be interpreted as strings.

cfA function Let f be a function name and $\{(v^1_1, \ldots, v^1_n)_{s^k_{ASM}}, \ldots, (v^m_1, \ldots, v^m_n)_{s^k_{ASM}}\}$ the list of arguments values which identify the defined locations (that is the locations whose value is different from *undef*) in state s^k_{ASM} ; moreover let $\{v^1_{s^k_{ASM}}, \ldots, v^m_{s^k_{ASM}}\}$ the values in the current state s^k_{ASM} of the locations $\{f(v^1_1, \ldots, v^1_n)_{s^k_{ASM}}, \ldots, f(v^m_1, \ldots, v^m_n)_{s^k_{ASM}}\}$. The string representation of the n-ary function f in state s^k_{ASM} is

$$cfA(f, s^k_{ASM}) = \{(v^1_1, \ldots, v^1_n)_{s^k_{ASM}} \to v^1_{s^k_{ASM}}, \ldots, (v^m_1, \ldots, v^m_n)_{s^k_{ASM}} \to v^m_{s^k_{ASM}}\}$$

where parentheses, commas and arrows must be interpreted as strings.

If f is a 0-ary function, it has just one location (if defined). The value of the location in the state s_k is v_{s_k}. The string representation of the 0-ary function f in state s_k is

$$cfA(f, s_k) = v_{s_k}$$

3.3 Execution step in Java and ASM

In order to define a step of a Java class execution, we heavily rely on the concept of *machine step* and *last state* of execution sequence defined in the Unifying Theories of Programming (UTP) [11].

The *step* is defined as a relation between the virtual state before the step and the virtual state after. In the case of an execution of a Java method, the state can be analyzed as a pair $(s; m)$, where s is the data part (actual values of the variables), and m is a representation of the rest of the method code that remains to be executed. When this is Π, there is no more method code to be executed; the state $(t; \Pi)$ is the *last state* of any execution sequence that contains it, and t defines the final values of the variables.

Definition 6. ***Java Step*** *Let m be a method of a Java class. A Java step is defined as the relation* $(s, m) \overset{Jstep}{\to} (s', \Pi)$*, where s is the starting state of the execution of m and s′ the last state of this execution.*

In the sequel, we abbreviate $(s, m) \overset{Jstep}{\to} (s', \Pi)$ with (s, m, s').

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

ASM *state* and ASM computation *step* have been defined in section 2.

[2] T and E must be one of the following type: a primitive type, an array, a String, a List, or a Set.

$$\begin{tikzcd}
ASM_C & {} \arrow[r, "init"] & S_0 \arrow[r, dotted] & S_j \arrow[rr, "step"] & & S_{j+1} \arrow[r, "step"] & {} \\
C & {} \arrow[r, "inst"] & s_0 \arrow[r, dotted] \arrow[u, dotted] & s_k \arrow[r, squiggly, "notCM"] \arrow[u, dotted] & s'_k \arrow[r, "CM"] \arrow[ul, dotted] & s_{k+1} \arrow[u, dotted] &
\end{tikzcd}$$

Figure 3: Runtime conformance

3.4 State Conformance, Step Conformance and Run Conformance

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

Definition 8. ***State Conformance*** *We say that* a state *s of C* conforms to a state *S of ASM_C if all observed elements of C have value string representation equal to the string representation of the values of the locations in ASM_C linked to them; i.e.*

$$conf(s, S) \equiv \forall e \in OS(C) : cfJ(e, s) = cfA(link(e), S) \tag{2}$$

Definition 9. ***Step Conformance*** *We say that* a change step *(s, m, s') of 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. ***Run time conformance*** *Given an* observed computation *of a Java class C, we say that C is* run time conforming *to its specification ASM_C if the following conditions hold:*

- *the initial state s_0 of the computation of 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 C,* conforms *with the step (S, S') of ASM_C with S the current state of ASM_C.*

This definition presumes there exists a computation of the class C one can observe. Furthermore, it assumes that the next state of C and of its specification ASM_C are unique, thus it assumes determinism of the system under monitoring. Non-deterministic computations are considered in the next section.

Due to the run-time conformance definition between a Java class and its ASM specification, 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 (see Fig.3).

$$\begin{tikzcd}
\text{ASM: } ASM_C & S \arrow[r, "step"] & S' \\
\text{Java class: } C & s \arrow[r, "m"] \arrow[u, dotted] & s' \arrow[u, dotted]
\end{tikzcd}$$

Figure 2: Step conformance

4 Dealing with Non-determinism

Definition 10 assumes that, in any computation, the next state of a Java class C and of its specification ASM_C are unique. Thus, the definition is adequate for deterministic systems: non-determinism is limited to monitored quantities, which, once non deterministically fixed by the environment, make the evolution of the system deterministic. In this section, we extend our conceptual framework to deal with non-determinism also *inside* the system (in the class C and/or in the specification ASM_C). We have identified the following non-deterministic situations:

- Non-deterministic Java class and non-deterministic ASM specification. This situation can be due to one of the following scenarios:

  - a class changing method has non-deterministic behavior, and so, therefore, the abstract specification. For instance, it contains a call to a method in the java.util.Random class;
  - the Java class has more then one changing method, each of which may be deterministic; however, it is non-deterministic the choice of the changing method that causes a change step. The abstract ASM model should capture this non-determinism and assure that the behavior of the methods is correct and that calling sequences are those permitted.

- Deterministic Java class and non-deterministic ASM specification. This situation can be due to an underspecification of the ASM model which may result more abstract (with less implementation details) than the corresponding Java code and possibly non-deterministic.

In case C or ASM_C are non-deterministic, the next computational state of C or ASM_C is not always uniquely determined, and, therefore, their conformance, according to definition 10, may fail not because of a wrong behavior of the implementation, but because C and ASM_C may choose a different non-conformant next state. We here refine definition 10 of step conformance and run-time conformance in case of non-determinism, distinguishing between weak and strong conformance.

4.1 Weak and strong conformance

For the *weak* conformance, we require that the next step of C is state-conforming with *at least one* of the next states of the specification ASM_C. For the strong conformance, we require that the next step of C is state-conforming with *one and only one* of the next states of the specification. Formally:

Weak run time conformance We say that C is *weakly* run time conforming to its specification ASM_C if the following conditions hold:

- the initial state s_0 of the computation of 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 $conf(s_0, S_0)$;
- for every change step (s, m, s') with s the current state of C, $\exists\,(S, S')$ step of ASM_C with S the current state of ASM_C, such that (s, m, s') is *step conforming* (S, S').

Strong run time conformance We say that C is *strongly* run time conforming to its specification ASM_C if the following conditions hold:

- the initial state s_0 of the computation of 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 $conf(s_0, S_0)$;
- for every change step (s, m, s') with s the current state of C, $\exists!\,(S, S')$ step of ASM_C with S the current state of ASM_C, such that (s, m, s') is *step conforming* (S, S').

Currently, our monitoring system can only deal with strong conformance. In case of nondeterministic ASM, during the runtime monitoring our system chooses, among the next states of the ASM, the state that is compliant with the Java state. If there is more than one state (weak conformance), the system does not know which one to choose. The feature of weak conformance is not supported for the moment, and a violation is risen since we require that C must be strong conformant with ASM_C. Weak conformance will be considered for future work.

5 Monitor Implementation

We here describe how our system works for the ASM-based runtime monitoring of Java programs. We provide technical details on how the observer and the analyzer have been implemented by exploiting the mechanism of the Java annotations to link observable Java elements to corresponding ASM entities, and the support of external tools as AspectJ to establish the conformance relation.

5.1 Using Java Annotations

Annotations are meta-data tags that can be used to add some information to code elements as class declarations, field declarations, etc. Each annotation have a *RetentionPolicy* that signals how and when the annotation can be accessed; *Runtime* policy, for example, signals that the annotation can be read by the compiler and can also be read reflectively at run-time.

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 of all of our annotations is *runtime* since we need to read them reflectively while the program is running.

5.1.1 Our annotations

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 specification to obtain dynamic information about the target program [6, 12]. 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 separately. This allows the reuse of a highly abstract formal requirement specification when changes happen to the implementation of the target system. Furthermore, annotations are statically type checked and since the annotations are read reflectively at run time, the monitoring setup can be carried out very easily. We found this approach much more convenient than inserting special comments (like JML) and writing our own parser for them.

Let's see in details each annotation.

@Asm In order to link a Java class C with its corresponding ASM model ASM_C, the Java class must be annotated with the `@Asm` annotation having the path of the ASM model as string attribute. The Java class *EuclidGCD* (see code 1) specifies, as its ASM specification, the model shown in code 2.

```
package euclid;

import org.asmeta.monitoring.Asm;
import org.asmeta.monitoring.FieldToFunction;
import org.asmeta.monitoring.Init;
import org.asmeta.monitoring.RunStep;
import org.asmeta.monitoring.StartMonitoring;

@Asm(asmFile=”models/euclidGCD.asm”)
public class EuclidGCD {
    @FieldToFunction(func=”numA”)
    public int numA;
    @FieldToFunction(func=”numB”)
    public int numB;

    @StartMonitoring
    public EuclidGCD(@Init(func=”initNumA”) int a,
                     @Init(func=”initNumB”) int b) {
        numA = a;
        numB = b;
    }

    public int getGCD() {
        while(numA != numB) {
            euclideGCDstep();
        }
        return numA;
    }

    @RunStep
    private void euclidGCDstep() {
        if(numA > numB) {
            numA = numA - numB;
        }
        else {
            numB = numB - numA;
        }
    }
}
```

Code 1: *GCD* Java code

```
asm euclidGCD

import ../../../../asm_examples/STDL/StandardLibrary

signature:
  dynamic controlled numA: Integer
  dynamic controlled numB: Integer
  dynamic monitored initNumA: Integer
  dynamic monitored initNumB: Integer

definitions:

  main rule r_Main =
    if(numA != numB) then
      if(numA > numB) then
        numA := numA - numB
      else
        numB := numB - numA
      endif
    endif

default init s0:
  function numA = initNumA
  function numB = initNumB
```

Code 2: *GCD* ASM model

@FieldToFunction and @MethodToFunction To establish the mapping defined by the function *link* we must annotate each observed field $f \in OF(C)$ and each observed method $m \in OM(C)$. The fields that are linked to controlled functions are annotated by `@FieldToFunction`, while the observed methods by `@MethodToFunction`; both these annotations have a string attribute yielding the name of the corresponding ASM controlled function.

In code 1 a Java code that computes the greatest common divisor (GCD) through the Euclidean algorithm is shown; an equivalent ASM model is shown in code 2. We can see that the Java fields *numA* and *numB* are linked with two homonymous ASM controlled functions.

Both `@FieldToFunction` and `@MethodToFunction` annotations are used to indicate some controlled functions of the ASM specification. The `@FieldToFunction` creates a direct connection

between a field and a function. The @MethodToFunction annotation, instead, permits to create more complicated links: the comparison is made between the value returned by the annotated method and the value of the referenced function. In this way we can link an ASM function with any computation of the Java code (operations between fields, method calls, ...).
The linking between the Java state and the ASM state can be made:

- using only @FieldToFunction annotations,
- using only @MethodToFunction annotations,
- or using both together[3].

Moreover there are no restrictions on the number of variables and methods by which an ASM function is referenced.

It's important to notice that, if it's not possible to establish a link between a field and a function (with the @FieldToFunction annotation)[4], we can establish the link through a getter method (annotated with the @MethodToFunction annotation) whose return value is compatible with the intermediate format of the ASM location values. Let's see, as an example, the Java code shown in code 3 and the ASM model shown in code 4. They both model a door which is, alternatively, *open* or *closed*.

```
package org.asmeta.monitoring;

@Asm(asmFile=”examples/door.asm”)
public class Door {
        boolean doorIsOpen;

        Door() {
                doorIsOpen = false;
        }

        @RunStep
        public void step() {
                doorIsOpen = !doorIsOpen;
        }


        @MethodToFunction(func=”doorStatus”, args = {})
        String getDoorIsOpen() {
                if(doorIsOpen) {
                        return ”OPEN”;
                }
                else {
                        return ”CLOSED”;
                }
        }
}
```

Code 3: *Door* Java code

[3] We can notice that all the links made with the @FieldToFunction annotation can also be made with the @MethodToFunction annotation: we just have to create a getter method for the variable annotated with the @FieldToFunction annotation and annotate it with the @MethodToFunction annotation (using the same values of the @FieldToFunction annotation, that is referencing the same function).

[4] It's not possible to establish a link between a field and a function when the intermediate representations of their values are not compatible.

```
asm door

import ../../../../asm_examples/STDL/StandardLibrary

signature:
  enum domain DoorStatusDomain = {OPEN | CLOSED}
  dynamic controlled doorStatus: DoorStatusDomain

definitions:

  main rule r_Main =
    if(doorStatus = CLOSED) then
      doorStatus := OPEN
    else
      doorStatus := CLOSED
    endif

default init s0:
  function doorStatus = CLOSED
```

Code 4: *Door* ASM model

The Java code represents the door status with the boolean variable *isOpen*. The ASM model, instead, uses the function *doorStatus*, which takes values in the enum domain $\{OPEN, CLOSED\}$, to indicate if the door is open or not. We can observe that the Java code and the ASM model do the same thing, but the intermediate representations of the Java variable *isOpen* and of the function *doorStatus* values are not compatible. So we have written the method *getIsOpen* (annotated with the `@MethodToFunction` annotation) which returns the string "OPEN" when *isOpen* is *true*, "CLOSED" otherwise.

@Monitored The fields whose values are determined at run time by the environment (e.g. values received by any kind of input stream) are linked to monitored ASM functions and they are annotated with `@Monitored`. These fields are used to give values to the corresponding ASM monitored functions before executing a *changing method*, as explained later in section 5.2.
Let's see, as an example, the Java code shown in code 5 and the ASM model shown in code 6. They both model an air conditioner that can be used with three speeds: 0 (turned off), 1 and 2. The speed depends on the temperature of the room. In the Java code the temperature is represented by the integer variable *roomTemperature*. This variable is *monitored* because its value is determined by the environment, i.e. a sensor controlled by an object of the class *TemperatureSensor*.
In the ASM model the temperature is modeled through the monitored function *temperature* that can assume the values 0, 1 and 2.
The Java variable *roomTemperature* is linked with the monitored function *temperature* of the ASM model.

```
package conditioner;

import org.asmeta.monitoring.Asm;
import org.asmeta.monitoring.FieldToFunction;
import org.asmeta.monitoring.Monitored;
import org.asmeta.monitoring.RunStep;
import org.asmeta.monitoring.StartMonitoring;

@Asm(asmFile="models/airConditioner.asm")
public class AirConditionerWithSensor {
        @Monitored(func=”temperature”, args={})
        public int roomTemperature;
        @FieldToFunction(func=”airSpeed”)
        public int airIntensity;
        private TemperatureSensor ts;

        @StartMonitoring
        public AirConditionerWithSensor() {
                airIntensity = 0;
                ts = new TemperatureSensor();
        }

        public void check() {
                readRoomTemperature();
                setAirIntensity();
        }

        @RunStep
        private void setAirIntensity() {
                if(roomTemperature < 20) {
                        airIntensity = 0;
                }
                else if(roomTemperature < 25) {
                        airIntensity = 1;
                }
                else {
                        airIntensity = 2;
                }
        }

        private void readRoomTemperature() {
                this.roomTemperature = ts.readRoomTemperature();
        }
}
```

Code 5: *Air conditioner* Java code

```
asm airConditioner

import ../../../../asm_examples/STDL/StandardLibrary

signature:
  domain AirSpeedDomain subsetof Integer
  dynamic controlled airSpeed: AirSpeedDomain
  dynamic monitored temperature: Integer

definitions:
  domain AirSpeedDomain = {0..2}

  main rule r_Main =
    if(temperature >= 25) then
      airSpeed := 2
    else
      if(temperature < 20) then
```

```
          airSpeed := 0
        else
          airSpeed := 1
        endif
      endif

default init s0:
  function airSpeed = 0
```

Code 6: *Air conditioner* ASM model

@RunStep All methods of $changingMethods(C)$ are annotated with the `@RunStep` annotation. In the Euclidean algorithm example (code 1), the changing method is *euclidGCDstep()* that executes a single step of the algorithm. In the air conditioner example (code 5), the changing method is *setAirIntensity()*.

@StartMonitoring and @Init Finally, the user have to decide the starting point of the monitoring. The annotation `@StartMonitoring` is used to select a proper (not empty) subset of constructors[5].

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 Java code of the Euclidean algorithm example (see code 1), in the constructor, the formal parameter *a* is annotated with the *@Init* annotation that contains a *reference* to the monitored function *initNumA* of the ASM model; in the same way the formal parameter *b* is linked to the monitored function *initNumB*. We can notice, indeed, that in the ASM model 2 the controlled functions *numA* and *numB* are initialized through the monitored functions *initNumA* and *initNumB*: in this way the ASM model can be executed several times to compute the GCD of different couples of numbers.

5.2 Observer implementation through AspectJ

The *observer* is implemented through the facilities of AspectJ that permits to observe easily the execution of Java objects. AspectJ allows to specify different *pointcuts*, that are points of the program execution we want 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.

In the definitions of AspectJ pointcuts, it is possible to add method annotations: this feature has permitted us to define easily the points of a program execution where our monitoring system must perform some given jobs.

For our purposes, we have defined the following two pointcuts:

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

The objCreated pointcut captures the creation of an instance of a class that must be monitored; runStepCalled captures the execution of a *changing method* (we do not consider changing methods that are executed in the scope of other changing methods).

In particular, after a joint point that belongs to objCreated, the monitor executes an advice that initializes a simulator for the corresponding ASM machine. Before the execution of a join

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

point that satisfies runStepCalled, an advice is executed that records the values of the *monitored* fields and executes a state conformance check. After this joint point, another advice is executed that sets the ASM monitored functions, simulates a step of the ASM and forces the analyzer to check again the state conformance.

5.3 Analyzer

The analyzer must execute the comparison between the Java and the ASM state. The values of the ASM functions are obtained through the facilities of the AsmetaS simulator (a simulator for ASMs [9]). The values of the Java fields and methods are obtained through reflection. This is the reason why we require that the methods in $OM(C)$ must be side-effect free: these methods are called through reflection by our monitoring system and we do not want that their execution influence (change) the Java state.

The Java and the ASM values are both transformed in a String representation through the functions cfJ and cfA (see section 3.2); so the conformance check is simply a string comparison.

6 Monitoring settings by examples

In the previous section we have described how it is possible to bind a Java class together with an ASM specification, how their runs are related and how (and when) the conformance analysis is executed.

In this section we want to show, by means of some examples, how the monitoring system can be used in practice. Indeed, based on the kind of Java class we want to monitor and on the kind of the ASM model we use as formal specification, we can identify different kinds of monitoring.

First of all, starting from the definitions given in section 4, we classify the Java classes and the ASM specifications according to their determinism/non-determinism.

Deterministic ASM specification A deterministic ASM specification is a specification that does not contain any choose rule. At each step there is just one possible update set and so just one possible next state.

Non-deterministic ASM specification A non-deterministic ASM specification is a specification that contains at least a choose rule. At each step there could be more than one possible update set and so more than one possible next state.

Internally non-deterministic/deterministic Java class Given a class *C*, let m be the set of its methods. We say that the class is *internally non-deterministic* if $\exists m_i \in m$ that contains non-deterministic statements (e.g. a method call on an object of the java.util.Random class). Otherwise, if $!\exists m_i \in m$ that contains non-deterministic statements, we say that the class is *internally deterministic.*

Externally non-deterministic/deterministic Java class Given a class *C*, let $m^{pub}_{\neg pure}$ be the public methods that can change the object state. We say that the class is *externally non-deterministic* if $|m^{pub}_{\neg pure}| > 1$. Indeed, if there is just one method in $m^{pub}_{\neg pure}$, at each step just one method of C can change the object state[6]. Otherwise, if there is more than a method in $m^{pub}_{\neg pure}$, at each step more than a method that change the object state can be executed. If $|m^{pub}_{\neg pure}| \leq 1$, we say that the class is *externally deterministic.*

Fully deterministic Java class A class C is *fully deterministic* if it is either *internally* and *externally* deterministic.

[6] Also some methods in m^{pub}_{pure} could be called, but these methods does not change the object state.

6.1 Monitoring a fully deterministic Java code with a deterministic ASM model

An example of deterministic Java class monitored by a deterministic ASM model is the *EuclidGCD* class shown in code 1. The class contains just one public method, *getGCD()*, that calls the *change method* method *euclidGCDstep()* until the condition *numA != numB* is satisfied. The Java code is fully deterministic:

- *externally deterministic*: there is just one change method that can be called; the method can be called several times, but just the first execution modifies the Java state;
- *internally deterministic*: the methods do not contain any non-deterministic statements.

The corresponding ASM model is deterministic as well.

6.2 Monitoring a fully deterministic Java code with a non-deterministic ASM model

In this section we show how it is possible to use a non-deterministic ASM model to monitor a deterministic Java class and when this approach is suggested. We will use, as example, the selection sort algorithm.

6.2.1 Selection sort

In code 7 the ASM model of a very trivial sorting algorithm is shown: at each step two non sorted elements are chosen and swapped. It is important to notice that is also possible that an element is swapped with itself (in this case the machine does nothing.)

```
asm randomSort

import ../../../../../asm_examples/STDL/StandardLibrary

signature:
    domain IndexDomain subsetof Natural
    dynamic controlled list: Seq(Integer)
    dynamic monitored initList: Seq(Integer)

definitions:
    domain IndexDomain = {0n..4n}

    main rule r_Main =
        choose $x in IndexDomain, $y in IndexDomain with $x <= $y and
                                   at(list, $x) >= at(list, $y) do
            if($x != $y and at(list, $x) > at(list, $y)) then
                let ($valAtX = at(list, $x), $valAtY = at(list, $y),
                        $list1 = subSequence(list, 0n, $x),
                          $list2 = subSequence(list, $x + 1n, $y),
                          $list3 = subSequence(list, $y + 1n, 5n)) in
                    list := union(
                                union(
                                    append($list1, $valAtY),
                                    append($list2, $valAtX)
                                ),
                                $list3
                            )
                endlet
```

```
          endif

default init s0:
    function list = initList
```

Code 7: *Random sort* ASM model

We can notice that the ASM model is nondeterministic; indeed, usually, at each step more than a step can be executed (more than a couple of elements can be swapped).
It is easy to understand that this ASM machine can model a wide range of sorting algorithms. Let's see, as an example, the selection sort algorithm shown in code 8.

```
package sort;

import org.asmeta.monitoring.Asm;
import org.asmeta.monitoring.FieldToFunction;
import org.asmeta.monitoring.Init;
import org.asmeta.monitoring.RunStep;
import org.asmeta.monitoring.StartMonitoring;

@Asm(asmFile="models/sort/randomSort.asm")
public class SelectionSort {
    @FieldToFunction(func="list")
    public int[] arr;

    @StartMonitoring
    SelectionSort(@Init(func="initList", args={}) int[] initArr) {
        arr = initArr;
    }

    public void sort() {
        for(int i = 0; i < arr.length - 1; i++) {
            swapMin(i);
        }
    }

    @RunStep
    private void swapMin(int i) {
        int minIndex = i;
        //minimum search
        for(int j = i + 1; j < arr.length; j++) {
            if(arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        //swap
        if(minIndex != i) {
            int temp = arr[i];
            arr[i] = arr[minIndex];
            arr[minIndex] = temp;
        }
    }
}
```

Code 8: *Selection sort* Java code

The sorting algorithm is implemented by the *sort()* method. This method iterates over the elements of the array to be sorted: for each element it calls the *swapMin(int i)* method which swaps the i^{th} element of the array with the minimum element of the sub-array identified by the indexes $[i, n-1]$.
The comparison between the Java execution and the ASM execution is made after each execution of the *swapMin(int i)* method. It is clear that the step executed by the Java machine (a particular swapping which is deterministically identified) can also be executed by the ASM machine (it is one of the possible steps of the ASM machine).

Let's see, as an example, how the array $\{2, 3, 1, 5, 4\}$ is sorted in the Java code and how the execution of the ASM model is influenced. Table 1 shows, for each iteration i of the selection sort algorithm, the obtained Java state and the ASM states which can be obtained starting from the $(i-1)^{th}$ ASM state. The state which is conformant with the Java state is shown in red. During the monitoring, the state shown in red is the state that is taken.

Iteration	**Java state**	**Possible next ASM states (in red the compliant state)**
1	$\{1, 3, 2, 5, 4\}$	$\{2, 3, 1, 5, 4\}$, $\{1, 3, 2, 5, 4\}$, $\{2, 1, 3, 5, 4\}$, $\{2, 3, 1, 5, 4\}$, $\{2, 3, 1, 4, 5\}$
2	$\{1, 2, 3, 5, 4\}$	$\{1, 3, 2, 5, 4\}$, $\{1, 2, 3, 5, 4\}$, $\{1, 3, 2, 4, 5\}$
3	$\{1, 2, 3, 5, 4\}$	$\{1, 2, 3, 5, 4\}$, $\{1, 2, 3, 4, 5\}$
4	$\{1, 2, 3, 4, 5\}$	$\{1, 2, 3, 5, 4\}$, $\{1, 2, 3, 4, 5\}$

Table 1: Java and ASM execution of the selection sort algorithm

6.3 Monitoring an externally non-deterministic Java code with a non-deterministic ASM model

In this section we show how our monitoring approach can be used to check that, not only that the implementation of the methods is correct, but also that the order in which the methods are called is correct. Sometimes, indeed, it's possible that, on a given object, methods can be called only following some particular orders.

Let's see, as an example, the *railroad gate* problem [7].

6.3.1 Railroad gate

A railroad gate is composed of a *gate* and a *light*. The light can be turned *off* or can *flash*. The gate can be in four states: *opened*, *closing*, *closed*, *opening*. Some states are forbidden; for example it's not possible that the gate is *closed* when the light is *off*.

In code 9 a code that implements an interface of the railroad gate is shown. At the beginning the gate is *opened* with the light *off*. The class exposes several methods; each method permits to handle a particular signal: for example the method *opening()* is used to change the status of the gate in *opening*.

In a real system, an instance of this class should be accessed by other different objects (or also threads) of the program, that should call the different methods it exposes. For example, in the real system, there should be a sensor on the gate which indicates that the gate is closed. A thread reads the value of this sensor and, when needed, calls some methods on the *RailroadGate* instance: when it intercepts a change in the sensor signal, from *not closed* to *closed*, it calls the method *closed()*. In the same way, the thread which knows that the gate must start closing (maybe because a train is coming), should call the method *closing()*.

```
package railroad;

import org.asmeta.monitoring.Asm;
import org.asmeta.monitoring.FieldToFunction;
import org.asmeta.monitoring.RunStep;
import org.asmeta.monitoring.StartMonitoring;

@Asm(asmFile = ”models/railroadGate.asm”)
public class RailroadGate {
    @FieldToFunction(func = ”light”)
    public LightState light;
    @FieldToFunction(func = ”gate”)
    public GateState gate;

    @StartMonitoring
    public RailroadGate() {
        light = LightState.OFF;
        gate = GateState.OPENED;
    }

    /**
     * Executed by the object which knows that the gate must be closed.
     */
    @RunStep
    public void closing() {
        gate = GateState.CLOSING;
    }

    /**
     * Executed by the object (sensor) which knows that the gate has
     * reached the closed position.
     */
    @RunStep
    public void closed() {
        gate = GateState.CLOSED;
    }

    /**
     * Executed by the object which knows that the gate must be opened.
     */
    @RunStep
    public void opening() {
        gate = GateState.OPENING;
    }

    /**
     * Executed by the object (sensor) which knows that the gate has
     * reached the opened position.
     */
    @RunStep
    public void opened() {
        gate = GateState.OPENED;
    }

    /**
     * Executed by the object which knows that the light must be
     * turned off.
     */
    @RunStep
    public void off() {
        light = LightState.OFF;
    }

    /**
     * Executed by the object which knows that the light must be
     * turned on.
     */
    @RunStep
    public void flashing() {
        light = LightState.FLASH;
    }
}
```

Code 9: *Railroad gate* Java code

It is easy to see that the Java code does not execute any check in order to verify that its methods are used correctly. For example, the method *off()* that turns off the light could be called also if the gate is *closed*; in this situation there will be a dangerous state with the gate *closed* and the light *off*.

In order to monitor that the methods of the class are called in a correct way, we have written the ASM model of the railroad. This model is a non-deterministic model that, at each step, randomly moves to a valid next state. It is important to notice that any step of the ASM machine satisfies the requirements of the problem. For example, if the gate is *CLOSING* there are two possible next states: in the first one the gate remains *CLOSING*, and in the second one the gate becomes *CLOSED*. So, all the runs of this ASM model satisfy the requirements.

```
asm railroadGate

import ../../../../asm_examples/STDL/StandardLibrary

signature:
  enum domain LightState = {FLASH | OFF}
  enum domain GateState = {CLOSED | OPENED | CLOSING | OPENING}
  dynamic controlled light: LightState
  dynamic controlled gate: GateState

definitions:

  rule r_lightOff =
    choose $l in LightState with true do
      light := $l

  rule r_gateClosed =
    if(gate = CLOSED) then
      choose $g in GateState with $g = CLOSED or $g = OPENING do
        gate := $g
    endif

  rule r_gateClosing =
    if(gate = CLOSING) then
      choose $g1 in GateState with $g1 = CLOSING or $g1 = CLOSED do
        gate := $g1
    endif
  rule r_gateOpening =
    if(gate = OPENING) then
      choose $g2 in GateState with $g2 = OPENING or $g2 = OPENED do
        gate := $g2
    endif

  rule r_gateOpened =
    if(gate = OPENED) then
      choose $i in {1..3} with true do
        switch $i
          case 1: light := OFF
          case 2: gate := CLOSING
          case 3: gate := OPENED
        endswitch
    endif

  main rule r_Main =
    if(light = OFF) then
      r_lightOff[]
    else
      par
        r_gateClosed[]
        r_gateClosing[]
        r_gateOpening[]
        r_gateOpened[]
      endpar
    endif

default init s0:
  function gate = OPENED
  function light = OFF
```

Code 10: *Railroad gate* ASM model

We have linked the Java class shown in code 9 with the ASM model shown in code 10. The Java fields *light* and *gate* correspond to the homonymous 0-ary ASM functions. All the changing methods of the class are annotated with the *@RunStep* annotation: this means that each execution of a method of the Java class corresponds to a step of simulation of the ASM machine.

Let's see in code 11 a main method which creates an instance of RailroadGate and calls some of its methods in a correct order.

```
package org.asmeta.programMonitoring;

import railroad.RailroadGate;

public class RailroadGateTest {
        public static void main(String[] args) {
                RailroadGate r = new RailroadGate();
                r.flashing();
                r.closing();
                r.closed();
                r.opening();
                r.opened();
                r.off();
        }
}
```

Code 11: *Railroad gate* correct execution

The program monitor, at each step (after the execution of a change method annotated with *@RunStep*), can find, between the possible next states of the ASM model, one (and only one) state which is compliant with the Java state.

In code 12, instead, the calling sequence is not correct because the light is switched off while the gate is closing.

```
package org.asmeta.programMonitoring;

import railroad.RailroadGate;

public class RailroadGateTest {
        public static void main(String[] args) {
                RailroadGate r = new RailroadGate();
                r.flashing();
                r.closing();
                r.off();//Error
        }
}
```

Code 12: *Railroad gate* wrong execution

We can see that, after executing the method *closing()*, the *gate* is *CLOSING* with the *light FLASH*; there is just one next ASM state compliant with the Java state and so the program can continue. After executing the *off()* method, instead, the program monitor is not able to find any next ASM state compliant with the Java state. Indeed the Java state obtained after the execution of the method is $\{gate = CLOSING, light = OFF\}$, whereas the possible next ASM states are $\{gate = CLOSING, light = FLASH\}$ and $\{gate = CLOSED, light = FLASH\}$. See table 2 for the complete description.

Method call	**Java state**	**Possible next ASM states** **(in red the conformant state)**

r.flashing();	gate = GateState.OPENED,
	light = LightState.FLASH
		{gate = OPENED, light = OFF},
		{gate = OPENED, light = FLASH}

r.closing();	gate = GateState.CLOSING,
	light = LightState.FLASH
		{gate = OPENED, light = FLASH},
		{gate = CLOSING, light = FLASH}

r.off();	gate = GateState.CLOSING,
	light = LightState.OFF
		{gate = CLOSING, light = FLASH},
		{gate = CLOSED, light = FLASH}

Table 2: Railroad gate - Java and ASM execution

In code 13 the class *RailroadGateSmart* is shown, a more secure solution for the railroad gate problem. The class contains just one change method, the method *exec* which accepts as input a variable of type *Command*, an enumerative that identifies the signals that the class can receive in order to modify its state. Depending on the command received and the current state, the method *exec* executes the command if it is permitted in the state, otherwise it ignores it.

```
package railroad;

import org.asmeta.monitoring.Asm;
import org.asmeta.monitoring.FieldToFunction;
import org.asmeta.monitoring.RunStep;
import org.asmeta.monitoring.StartMonitoring;

@Asm(asmFile = ”models/railroadGate.asm”)
public class RailroadGateSmart {
    @FieldToFunction(func = ”light”)
    public LightState light;
    @FieldToFunction(func = ”gate”)
    public GateState gate;

    @StartMonitoring
    public RailroadGateSmart() {
        light = LightState.OFF;
        gate = GateState.OPENED;
    }

    @RunStep
    public void exec(Command command) {
        switch(command) {
            case FLASH:
                light = LightState.FLASH;
                break;
            case OFF:
                if(gate == GateState.OPENED) {
                    light = LightState.OFF;
                }
                break;
            case CLOSED:
                if(gate == GateState.CLOSING || gate == GateState.CLOSED) {
                    gate = GateState.CLOSED;
                }
                break;
            case OPENED:
                if(gate == GateState.OPENING || gate == GateState.OPENED) {
                    gate = GateState.OPENED;
                }
                break;
            case CLOSING:
                if(light == LightState.FLASH &&
                  (gate == GateState.OPENED ||gate == GateState.CLOSING)) {
                    gate = GateState.CLOSING;
                }
                break;
            case OPENING:
                if(gate == GateState.CLOSED || gate == GateState.OPENING) {
                    gate = GateState.OPENING;
                }
                break;
        }
    }
}
```

Code 13: *Railroad gate smart* Java code

The Java code is still bound with the ASM model shown in code 10. Unlike the Java code shown in code 9, in this case any calling sequence is permitted[7]. We can run the Java code with any sequence of commands; the Java execution will be always compliant with the ASM simulation because if there is a valid command the Java code reacts in a correct way, and if there is a non correct command the Java code does not change its state. As we have previously seen, if the Java program moves to a valid state also the ASM machine can move to a valid state; if the Java program does not change its state (because it has received a wrong command), also the ASM machine can keep the state unchanged[8].

6.4 Monitoring an internally non-deterministic Java code with a non-deterministic ASM model

Let's see how it is possible to monitor an internally non-deterministic Java code with a non-deterministic ASM machine. We will see, as an example, the *Knight's Tour* problem [16].

6.4.1 Knight's tour

A knight is placed on the empty board; moving according to the rules of chess, it must reach each square of the board just once. The tour is *closed* if the last square visited by the knight is the square from which it began, otherwise is *open* (our model looks for open tours).
Table 3 shows in green the squares of the board that can be reached by the knight placed in `h6`.

	a	b	c	d	e	f	g	h
8						2	5	
7						×		3
6					1	4		♞
5						×		
4							×	
3								
2								
1								

Table 3: Possible moves of knight placed in `h6`

We can see that the knight has started his tour in `e6` and, after 5 moves, he has arrived in `h6` (`e6 - f8 - h7 - f6 - g8 - h6`). From there he can go in `f7`, `f5` or `g4`, but not in `g8`, because he has already visited it.

It is possible that a tour, nevertheless there are squares not yet visited, can not be completed because the knight is blocked in a square from which it can not execute any valid move. Table 4 shows a tour that can not be completed; the knight has continued the tour shown in table 3 with 4 more moves (`h6 - f7 - e5 - g6 - h8`) arriving in `h8`: from there he can not execute any valid move because all the squares that are reachable with the *knight move* (`f7` and `g6`) have already been visited.

Let's see the formal specification of the problem in code 14.

```
asm KnightTour
```

[7] A sequence of method calls is also identified by the current values of the method parameters. For this reason we can talk of different sequences of call methods also in this example in which there is just one method.

[8] It's more correct to say that the ASM machine executes an empty update set that leaves the state unchanged.

	a	b	c	d	e	f	g	h
8						2	5	♞
7						7		3
6					1	4	9	6
5					8			
4								
3								
2								
1								

Table 4: Knight blocked in `h8`

```
import ../../../../../asm_examples/STDL/StandardLibrary

signature:
  enum domain Status = {VISITED | EMPTY}
  domain Rows subsetof Integer
  domain Columns subsetof Integer
  dynamic controlled posX: Rows
  dynamic controlled posY: Columns
  dynamic monitored initX: Rows
  dynamic monitored initY: Columns
  dynamic controlled board: Prod(Rows, Columns) -> Status

definitions:
  domain Rows = {0..7}
  domain Columns = {0..7}

  main rule r_Main =
    choose $x in Rows, $y in Columns with
        board($x, $y) = EMPTY and
        ((abs(posX - $x) = 1 and abs(posY - $y) = 2) or
         (abs(posX - $x) = 2 and abs(posY - $y) = 1)) do
      par
        posX := $x
        posY := $y
        board($x, $y) := VISITED
      endpar

default init s0:
  function posX = initX
  function posY = initY
  function board($x in Rows, $y in Columns) =
                          if ($x = initX and $y = initY) then
                            VISITED
                          else
                            EMPTY
                          endif
```

Code 14: *Knight's tour* ASM model

The ASM model, at each step, chooses non-deterministically a square of the board between those that are associated with a legal move and marks it as *VISITED*. The ASM model stops its execution when the update set is empty, that is when it can not find any legal move; a legal move can not be find because the knight is blocked or because he has finished the tour.

In code 15 we can see a non-deterministic Java code. The non-determinism is *internal* because the behavior of the method *execMove()* is non-deterministic.

```
package knightTour;

import java.util.ArrayList;
import java.util.Random;

import org.asmeta.monitoring.Asm;
import org.asmeta.monitoring.FieldToFunction;
import org.asmeta.monitoring.Init;
import org.asmeta.monitoring.RunStep;
import org.asmeta.monitoring.StartMonitoring;

@Asm(asmFile = ”models/nonDetModels/KnightTour.asm”)
public class KnightTour {
    @FieldToFunction(func = ”posX”)
    public int x;
    @FieldToFunction(func = ”posY”)
    public int y;
    private Status[][] board;

    @StartMonitoring
    public KnightTour(@Init(func = ”initX”, args = {}) int x,
            @Init(func = ”initY”, args = {}) int y) {
        this.x = x;
        this.y = y;
        board = new Status[8][8];
        for(int i = 0; i < board.length; i++) {
            for(int j = 0; j < board[i].length; j++) {
                board[i][j] = Status.EMPTY;
            }
        }
        board[x][y] = Status.VISITED;
    }

    public void findTour() {
        while(execMove());
    }

    @RunStep
    private boolean execMove() {
        ArrayList<int[]> possibleChoices = getPossibleMoves();
        int numOfMoves = possibleChoices.size();
        if(numOfMoves > 0) {
            int[] choice = possibleChoices.get(
                                        new Random().nextInt(numOfMoves));
            x = choice[0];
            y = choice[1];
            board[x][y] = Status.VISITED;
            return true;
        }
        return false;
    }

    private boolean availableMove(int newX, int newY) {
        return ((newX >= 0 && newX < 8) && (newY >= 0 && newY < 8)) &&
                    board[newX][newY]==Status.EMPTY;
    }

    private ArrayList<int[]> getPossibleMoves() {
        ArrayList<int[]> possibleChoices = new ArrayList<int[]>();
        int[] choice = new int[2];
        choice[0] = x + 1;
        choice[1] = y + 2;
        if (availableMove(choice[0], choice[1])) {
            possibleChoices.add(choice.clone());
        }
        choice[1] = y - 2;
        if (availableMove(choice[0], choice[1])) {
            possibleChoices.add(choice.clone());
        }
        choice[0] = x - 1;
        if (availableMove(choice[0], choice[1])) {
            possibleChoices.add(choice.clone());
        }
        choice[1] = y + 2;
        if (availableMove(choice[0], choice[1])) {
            possibleChoices.add(choice.clone());
        }
        choice[0] = x - 2;
        choice[1] = y - 1;
        if (availableMove(choice[0], choice[1])) {
            possibleChoices.add(choice.clone());
        }
        choice[0] = x + 2;
        if (availableMove(choice[0], choice[1])) {
            possibleChoices.add(choice.clone());
        }
        choice[1] = y + 1;
        if (availableMove(choice[0], choice[1])) {
            possibleChoices.add(choice.clone());
        }
        choice[0] = x - 2;
        if (availableMove(choice[0], choice[1])) {
            possibleChoices.add(choice.clone());
        }
        return possibleChoices;
    }
    private enum Status {
        VISITED, EMPTY;
    }
}
```

Code 15: *Knight's tour* Java code

As the ASM model, also the Java code chooses non-deterministically one move between those valid (let's see the change method *execMove()*).

Deterministic version of the knight's tour problem As we have seen in section 6.2 it is possible that the Java code is deterministic, nevertheless the corresponding ASM model is non-deterministic. Code 16 shows a modified version of code 15, in which a resolution strategy is implemented[9]. The code chooses, between the legal moves, the move which gets the knight closer to the middle of the board. We can notice that the Java code is *fully deterministic*:

- *internally deterministic*: the bodies of the methods do not contain any non-deterministic statements;
- *externally deterministic*: there is only one change method *findTour()*, that is the user of the class can interact with the object in just one way. So, we do not have to monitor the way the object is used.

The corresponding ASM model is the same as before. So, if the Java code is compliant, the Java run (given an initial state there is just one run) corresponds to one of the several ASM runs.

```
package knightTour;

import java.util.ArrayList;
import java.util.Collections;

import org.asmeta.monitoring.Asm;
import org.asmeta.monitoring.FieldToFunction;
import org.asmeta.monitoring.Init;
import org.asmeta.monitoring.RunStep;
import org.asmeta.monitoring.StartMonitoring;

@Asm(asmFile = "models/nonDetModels/KnightTour.asm")
public class KnightTourMiddleStrategy {
    @FieldToFunction(func = "posX")
    public int x;
    @FieldToFunction(func = "posY")
    public int y;
    private Status[][] board;

    @StartMonitoring
    public KnightTourMiddleStrategy(
                                 @Init(func = "initX", args = {}) int x,
                                 @Init(func = "initY", args = {}) int y) {
        this.x = x;
        this.y = y;
        board = new Status[8][8];
        for(int i = 0; i < board.length; i++) {
            for(int j = 0; j < board[i].length; j++) {
```

[9] This code never reaches a complete tour from any initial state.

```
                board[i][j] = Status.EMPTY;
            }
        }
        board[x][y] = Status.VISITED;
    }

    public void findTour() {
        while(execMove());
    }

    @RunStep
    private boolean execMove() {
        ArrayList<Square> possibleChoices = getPossibleMoves();
        int numOfMoves = possibleChoices.size();
        if(numOfMoves > 0) {
            Collections.sort(possibleChoices);
            Square choice = possibleChoices.get(0);
            x = choice.x;
            y = choice.y;
            board[x][y] = Status.VISITED;
            return true;
        }
        return false;
    }

    private boolean availableMove(int newX, int newY) {
        return ((newX >= 0 && newX < 8) && (newY >= 0 && newY < 8)) &&
                    board[newX][newY]==Status.EMPTY;
    }

    private ArrayList<Square> getPossibleMoves() {
        ArrayList<Square> possibleChoices = new ArrayList<Square>();
        int tempX = x + 1;
        int tempY = y + 2;
        if (availableMove(tempX, tempY)) {
            possibleChoices.add(new Square(tempX, tempY));
        }
        tempY = y - 2;
        if (availableMove(tempX, tempY)) {
            possibleChoices.add(new Square(tempX, tempY));
        }
        tempX = x - 1;
        if (availableMove(tempX, tempY)) {
            possibleChoices.add(new Square(tempX, tempY));
        }
        tempY = y + 2;
        if (availableMove(tempX, tempY)) {
            possibleChoices.add(new Square(tempX, tempY));
        }
        tempX = x - 2;
        tempY = y - 1;
        if (availableMove(tempX, tempY)) {
            possibleChoices.add(new Square(tempX, tempY));
```

```java
        }
        tempX = x + 2;
        if (availableMove(tempX, tempY)) {
            possibleChoices.add(new Square(tempX, tempY));
        }
        tempY = y + 1;
        if (availableMove(tempX, tempY)) {
            possibleChoices.add(new Square(tempX, tempY));
        }
        tempX = x - 2;
        if (availableMove(tempX, tempY)) {
            possibleChoices.add(new Square(tempX, tempY));
        }
        return possibleChoices;
    }

    private enum Status {
        VISITED, EMPTY;
    }
}

class Square implements Comparable<Square> {
    int x;
    int y;

    public Square(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public int compareTo(Square o) {
        double thisDistanceFromMiddle = Math.pow(x - 3.5, 2) +
                                        Math.pow(y - 3.5, 2);
        double oDistanceFromMiddle = Math.pow(o.x - 3.5, 2) +
                                     Math.pow(o.y - 3.5, 2);
        if(thisDistanceFromMiddle > oDistanceFromMiddle) {
            return 1;
        }
        else if(thisDistanceFromMiddle == oDistanceFromMiddle) {
            return 0;
        }
        else {
            return -1;
        }
    }

}
```

Code 16: *Knight's tour* Java code - Middle strategy

7 Related work

Complete surveys about runtime verification can be found in [7, 14, 8].

Our work has been inspired by the work presented in [15], 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 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 where to specify the sequence of methods one wants 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) [6] 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. As we do, they use AspectJ to insert the monitoring code into the monitored code; in particular AspectJ gives them the ability to place the monitoring code before or after some methods invocations. Another tool that, like MOP and our system, uses AspectJ to weave the monitoring code into the monitored code is *Lime* [12]. The 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 non-deterministic finite automata. For each call/return specification they build an AspectJ advice. This approach is different from ours, since we do not need to build an advice for each specification: our ASM specifications are simulated by an ASM simulator. We have a fixed number of advices that coordinate the monitoring jobs, as for example, building the ASM simulators and executing the conformance checking.

Another approach that uses the ASMs as formal specification for system monitoring purpose is presented in [2]. That approach shares with ours many common features: the use of operational specifications (called model programs), and dealing with non-determinism and 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 [3] an approach is presented in which 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.

8 Discussion and Conclusions

We wanted to assess the viability of our approach by taking several examples in literature and (1) check whether we were able to apply our approach to existing runtime case studies and (2) compare with them in terms of usability. We have gathered several examples, including the Railroad Gate [7], the Initialization Fiasco problem [3], a robotic assembly system [15], the Knight's Tour problem [16], and we have written the Java code if not available and their ASM specification. Several cases were non-deterministic. To avoid the abstract implementation to be biased by its implementation, one

```
@Asm(asmFile = "models/mod2.asm")
public class IncrDecr {
  @FieldToFunction(func="foo")
  public int foo;

  @StartMonitoring
  public IncrDecr() {
    foo = 1;
  }

  @RunStep
  public void incr() {
    foo++;
  }

  @RunStep
  public void decr() {
    foo--;
  }
}
```

Code 17: *Increment/decrement* Java code

```
asm mod2
import StandardLibrary

signature:
    controlled foo: Integer

definitions:
    main rule r_Main =
        foo := (foo + 1) mod 2

default init s0:
    function foo = 1
```

Code 18: *Increment/decrement* ASM model

of us wrote the ASM specifications while the others wrote the Java implementation. Of course, the ASM specifications and their implementations were similar, but we found also that we gave different interpretations to the requirements which could be easily detected by runtime monitoring. Overall we found our approach applicable to all the case studies we found.

Comparing our approach with others based on the use of properties in terms of usability is more difficult. 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 observed that practitioners feel uncomfortable with declarative notations like temporal logics. The scope of this paper is to provide evidence that also operational notation can be efficiently used for run time monitoring. Sometimes, operational specifications are easy to write and understand. Consider for example the following requirement about the gate [7]: "always in the future whenever the light is off, then the gate will not be closing until the light will be flashing", which can be formalized by the LTL formula: $\Box(\mathit{is\_off} \rightarrow \neg \mathit{is\_closing}\ \mathcal{U}\ \mathit{is\_flashing})$. In ASM, this requirement is easily translated in the following two rules:

```
//only if the light is OFF, it can start flashing
if light = OFF then choose $l in {FLASH | OFF} do light := $l endif
// the gate can starts closing only if it is open and the light is flashing
if light = FLASH and gate = OPENED then
   choose $g in {OPENED | CLOSING} do gate := $g endif
```

It is also possible to specify correct sequences of method calls by using LTL too. For instance, the program reported in code 17 requires that foo is alternatively updated to 0 and 1. An LTL formula capturing this behavior would be $(foo = 0 \wedge \Box(foo = 0 \rightarrow \bigcirc(foo = 1) \wedge foo = 1 \rightarrow \bigcirc(foo = 0))$. Although LTL notation is more compact, we believe that it may be more difficult to understand than a simple assignment instruction shown in code 18.

Although it is difficult to give a definitive evaluation, we believe that an operational style may be more easy to write and understand. Moreover, there are some advantages non related to run time verification in using executable specifications (as also discussed in [2]), including that specifications can be executed in isolation, even before the implementation exists.

Our approach has some limits. The use of an operational specification can prone the designer to insert implementation details in the specification. Other limits derive from the use of Asmeta. In Asmeta, non-determinism is restricted: the choose rule works only over bounded set (as in [2]). Dealing with metric time seems problematic: we believe that a *time* monitored function may model the real time and would allow its measurement, but further experiences are needed.

Despite these limits, we believe that our approach presents a viable technique for checking conformance of an implementation (as Java program) with respect its formal and abstract operational specification (as ASM). The operational style should be appealing for those preferring executable models instead of properties. In our approach, specifications are developed independent from the implementations and they are linked by Java annotations which however contain minimal behavioral information. We are able to deal with non-deterministic systems and this allows furthermore to check sequences of method calls and to write more abstract specifications.

References

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

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

[3] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software and Methodology (TOSEM)*, accepted, 2010. accepted for publication.

[4] 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.

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

[6] 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.

[7] S. Colin and L. Mariani. 18 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.

[8] 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.

[9] 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.

[10] K. Havelund and G. Roşu. Efficient monitoring of safety properties. *Int. J. Softw. Tools Technol. Transf.*, 6:158–173, August 2004.

[11] C. A. R. Hoare and J. He. *Unifying Theories of Programming.* Prentice-Hall International, Englewood Cliffs, NJ, 1998.

[12] 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, Berlin, Heidelberg, 2009.

[13] 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.

[14] M. Leucker and C. Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293 – 303, 2009. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07).

[15] H. Liang, J. Dong, J. Sun, and W. Wong. Software monitoring through formal specification animation. *Innovations in Systems and Software Engineering*, 5:231–241, 2009. 10.1007/s11334-009-0096-1.

[16] E. W. Weisstein. Knight's tour. MathWorld–A Wolfram Web Resource.