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

Paolo Arcaini\textsuperscript{1}  Angelo Gargantini\textsuperscript{1}  Elvinia Riccobene\textsuperscript{2}

\textsuperscript{1} Dipartimento di Ingegneria, Università degli Studi di Bergamo, Italy
\textsuperscript{2} Dipartimento di Informatica, Università degli Studi di Milano, Italy

September 24, 2014
AVoCS 2014
Outline

- Runtime verification
- Abstract State Machines (ASMs)
- Runtime verification through ASMs
- Nondeterminism in runtime verification
- Proposed approach
  - SMT encoding for ASMs
  - SMT-based runtime verification approach
- Experimental results
Most of the approaches use *declarative* specifications for the monitor (e.g., LTL)

We proposed the use of *operational* specifications (ASMs) [AGR11]
Abstract State Machines (ASMs)

- Abstract State Machines (ASMs) are an extension of FSMs
- States are multi-sorted first-order structures, i.e., domains of objects with functions and predicates defined on them
- The transition relation is specified by rules describing how functions interpretations change:
  - block rule, update rule, forall rule, choose rule, ...
- Functions are classified in:
  - controlled, monitored, static, derived, ...
- A run is a finite or infinite sequence of states
- We use the following definition:
  - $ASM = \langle \text{signature}, \text{funcDefs}, \text{funcInit}, r\_main \rangle$
ASM – Running case study (Tank)

```asm
tank

signature:
  controlled level: Integer
  derived full: Boolean

definitions:
  function full = (level = 1000)

main rule r_main =
  choose $x$ in $\{-50..50\}$ with level + $x$ $\geq$ 0 and level + $x$ $\leq$ 1000 do
  level := level + $x$

default init s0:
  function level = 0
```

Arcaini, Gargantini, Riccobene  Using SMT for dealing with nondet. in ASM-based RV
Conformance Monitoring by ASMs (CoMA) [AGR11]

- It executes runtime verification of Java programs by ASMs.
- A link between a Java class and an ASM model must be established:
  - some fields and pure methods are connected to controlled functions; they describe the state the user wants to monitor.
  - an ASM step corresponds to the execution of a method $m \in$ changing methods.
  - method parameters can be linked to monitored functions.
State and step conformance

Three notions of conformance have been defined:

- **State conformance**: the values of the fields under monitoring and the values of their corresponding functions are *conformant*.

- **Step conformance**: the Java and ASM states before and after the execution of a changing method are conforming.

\[ \text{ASM}_C \quad \xrightarrow{\text{simulation step}} \quad \text{ASM}' \]

\[ \text{OC} \quad \xrightarrow{\text{invocation of method } m} \quad \text{OC}' \]
Runtime conformance (deterministic case)

- **Runtime conformance**: given an observed Java execution, the object $O_C$ of class $C$ is runtime conforming to its specification $ASM_C$ iff:
  - the initial states are conforming
  - every *changing step* is step conforming
  - no specification invariant is ever violated

\[
\begin{align*}
ASM_C & \quad \xrightarrow{\text{init}} \quad S_0 \quad \xrightarrow{\text{step}} \quad S_j \quad \xrightarrow{\text{step}} \quad S_{j+1} \\
O_C & \quad \xrightarrow{\text{inst}} \quad s_0 \quad \xrightarrow{\text{CM}} \quad s_k \quad \xrightarrow{\text{CM}} \quad s_{k+1} \quad \xrightarrow{\text{CM}} \quad s_{k+2}
\end{align*}
\]
Different settings:
- The implementation and the formal specification are nondeterministic
- The implementation is deterministic, while the specification is nondeterministic:
  - Abstraction

Runtime verification with nondeterministic specifications
- *Univocal* runtime conformance [AGR13]
- *Multiple* runtime conformance
  - supported in the current work using an SMT-based approach
Univocal runtime conformance [AGR13]

Given an observed Java execution, the object $O_C$ of class $C$ is *univocally* runtime conforming to its specification $ASM_C$ iff:

- the initial state of $O_C$ conforms to *one and only one* initial state of the computation of $ASM_C$
- every change step is *step conforming* with *one and only one* of the possible steps of $ASM_C$

![Diagram showing the relationship between $O_C$ and $ASM_C$]

- Supported by CoMA by explicitly representing all the conformant states
Univocal runtime conformance – Tank case study

Java code

```java
@Asm(asmFile = "models/Tank.asm")
public class Tank {
    private int level;

    @StartMonitoring
    public Tank() { level = 0; }

    @RunStep
    public void increaseLevel(int valueToAdd) {
        if (valueToAdd >= -5 && valueToAdd <= 5) {
            int newValue = level + valueToAdd;
            if (newValue >= 0 && newValue <= 50) {
                level = level + valueToAdd;
            }
        }
    }

    @MethodToLocation(func = "full")
    public boolean isFull() {
        return level == MAX_LEVEL;
    }

    @MethodToLocation(func = "level")
    public int getLevel() {
        return level;
    }
}
```

ASM model

```asm
asm tank

signature:
controlled level: Integer
derived full: Boolean

definitions:
function full = (level = 1000)

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

default init s0:
    function level = 0
```
Multiple runtime conformance

Given an observed Java execution, the object $O_C$ of class $C$ is *multiply* runtime conforming to its specification $ASM_C$ iff:

- the initial state of $O_C$ conforms to *at least one* initial state of the computation of $ASM_C$
- every change step is *step conforming* with *at least one* of the possible steps of $ASM_C$

\[ O_C \xrightarrow{\text{conf}} \{ S_1, \ldots, S_k \} \]

\[ s \xrightarrow{\text{step}} s' \]

\[ m \xrightarrow{\text{conf}} \text{conf}(s', S_i) \]

\[ \text{conf}(s', S_i') \]

\[ S_1' \xrightarrow{\text{step}} \ldots \]

\[ S_i' \xrightarrow{\text{step}} S_j' \]

\[ \ldots \]

\[ \text{Not supported by CoMA} \]

- Proposed approach: symbolically representing ASM computations

Arcaini, Gargantini, Riccobene

Using SMT for dealing with nondet. in ASM-based RV
Multiple runtime conformance – Tank case study

Java code

@Asm(asmFile = "models/Tank.asm")
public class Tank {
    private int level;

    @StartMonitoring
    public Tank() { level = 0; }

    @RunStep
    public void increaseLevel(int valueToAdd) {
        if (valueToAdd >= -5 && valueToAdd <= 5) {
            int newValue = level + valueToAdd;
            if (newValue >= 0 && newValue < 50) {
                level = level + valueToAdd;
            }
        }
    }

    @MethodToLocation(func = "full")
    public boolean isFull() {
        return level == MAX_LEVEL;
    }

    // The level is not linked
    public int getLevel() {
        return level;
    }
}

ASM model

asm tank

signature:
    controlled level: Integer
derived full: Boolean

definitions:
    function full = (level = 1000)

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

    default init s0:
        function level = 0
We want to represent the set $RS_i$ of ASM states reachable in $i$ steps.

We define four mapping operators (with parameter $i$):

- $T_f$ for function declarations

<table>
<thead>
<tr>
<th>ASM function declaration</th>
<th>Yices</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>funcType</strong> $f$: Dom</td>
<td>$(\text{define } f^i :: \text{Dom})$</td>
</tr>
<tr>
<td><strong>funcType</strong> $f$: $D_1 \rightarrow D_2$</td>
<td>$(\text{define } f^i :: (\rightarrow D_1 D_2))$</td>
</tr>
<tr>
<td><strong>funcType</strong> $f$: Prod($D_1, \ldots, D_n$) $\rightarrow D$ with $n \geq 2$</td>
<td>$(\text{define } f^i :: (\rightarrow D_1 \ldots D_n D))$</td>
</tr>
</tbody>
</table>
ASM symbolic representation

- We want to represent the set $RS_i$ of ASM states reachable in $i$ steps
- We define four mapping operators (with parameter $i$):
  - $T_f$ for function declarations
  - $T_d$ for function definitions

<table>
<thead>
<tr>
<th>ASM function definition</th>
<th>Yices</th>
</tr>
</thead>
<tbody>
<tr>
<td>function $f = \text{fd}$</td>
<td>$(\text{assert } (= f^i \ T_t(f, i)))$</td>
</tr>
<tr>
<td>function $f(x_1 \ \text{in} \ D_1, \ldots, x_n \ \text{in} \ D_n) = \text{fd}[x_1, \ldots, x_n]$</td>
<td>$(\text{assert } (\text{and } (= T_t(f(d^1_1, \ldots, d^n_n), i) \ T_t(f(d^1_1, \ldots, d^n_n), i)) \ldots (= T_t(f(d^1_1, \ldots, d^n_n), i) \ T_t(f(d^1_1, \ldots, d^n_n), i))))$</td>
</tr>
</tbody>
</table>

with $n \geq 1$ and $D_1 = \{d^1_1, \ldots, d^1_{m_1}\} \ldots D_n = \{d^n_1, \ldots, d^n_{m_n}\}$
We want to represent the set $RS_i$ of ASM states reachable in $i$ steps.

We define four mapping operators (with parameter $i$):

- $T_f$ for function declarations
- $T_d$ for function definitions
- $T_t$ for terms

<table>
<thead>
<tr>
<th>ASM term</th>
<th>Yices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location term: $f$</td>
<td>$f^i$</td>
</tr>
<tr>
<td>Location term: $f(a_1, \ldots, a_n)$ with $n \geq 1$</td>
<td>($f^i \ T_t(a_1, i) \ldots T_t(a_n, i)$)</td>
</tr>
<tr>
<td>Boolean term: $b$ with $b \in {\text{true, false}}$</td>
<td>$b$</td>
</tr>
<tr>
<td>Integer term: $h$ with $h \in \mathbb{Z}$</td>
<td>$h$</td>
</tr>
<tr>
<td>Natural term: $hn$ with $h \in \mathbb{N}$</td>
<td>$h$</td>
</tr>
<tr>
<td>Enumeration term: $E$</td>
<td>$E$</td>
</tr>
<tr>
<td>if guard then $T$then else $T$else endif</td>
<td>(if $T_t$ (guard, $i$) $T_t$ (then, $i$) $T_t$ (else, $i$))</td>
</tr>
<tr>
<td>(forall $x_1$ in $D_1, \ldots, x_n$ in $D_n$ with $\text{cond} [x_1, \ldots, x_n]$)</td>
<td>(and $c_1 \ldots c_m$ with $m = \prod_{j=1}^{n}</td>
</tr>
<tr>
<td>(exists $x_1$ in $D_1, \ldots, x_n$ in $D_n$ with $\text{cond} [x_1, \ldots, x_n]$)</td>
<td>(or $c_1 \ldots c_m$ with $m = \prod_{j=1}^{n}</td>
</tr>
</tbody>
</table>
ASM symbolic representation

- We want to represent the set $RS_i$ of ASM states reachable in $i$ steps

- We define four mapping operators (with parameter $i$):
  - $T_f$ for function declarations
  - $T_d$ for function definitions
  - $T_t$ for terms
  - $T_r$ for transition rules

---

<table>
<thead>
<tr>
<th>ASM transition rule</th>
<th>Yices</th>
</tr>
</thead>
<tbody>
<tr>
<td>updLoc := updTer</td>
<td>$T_t$(updLoc, $i + 1) = T_t$(updTer, $i$)</td>
</tr>
<tr>
<td>par $R_1 \ldots R_n$ endpar</td>
<td>$(\text{and } T_r(R_1, i) \ldots T_r(R_n, i))$</td>
</tr>
<tr>
<td>if guard then Rthen else Relse endif</td>
<td>$(\text{if } T_t(\text{guard}, i) \text{ T}_r(\text{Rthen}, i) \text{ T}_r(\text{Relse}, i))$</td>
</tr>
<tr>
<td>if guard then Rthen endif</td>
<td>$(\Rightarrow T_t(\text{guard}, i) \text{ T}_r(\text{Rthen}, i))$</td>
</tr>
<tr>
<td>forall $x_1 \in D_1, \ldots, x_n \in D_n$ with guard[$x_1, \ldots, x_n$] do $R[x_1, \ldots, x_n]$</td>
<td>$(\text{and } r_1 \ldots r_m) \text{ with } m = \prod_{j=1}^n</td>
</tr>
<tr>
<td>choose $x_1 \in D_1, \ldots, x_n \in D_n$ with guard[$x_1, \ldots, x_n$] do $R[x_1, \ldots, x_n]$</td>
<td>for each $x_j$: (define $cv^j_i$ :: $D_j$) $(\Rightarrow (\exists d_1:: D_1 \ldots d_n:: D_n) \text{ T}_t(\text{guard}[x_1 \leftrightarrow d_1, \ldots, x_n \leftrightarrow d_n], i))$ $(\text{and } T_t(\text{guard}[x_1 \leftrightarrow cv^j_i, \ldots, x_n \leftrightarrow cv^i_n], i)$ $T_r(R[x_1 \leftrightarrow cv^j_i, \ldots, x_n \leftrightarrow cv^i_n], i))$</td>
</tr>
<tr>
<td>main rule $\text{r_main = mainBody}$</td>
<td>$(\text{assert } T_r(\text{mainBody}, i))$</td>
</tr>
</tbody>
</table>
Logical context creation

- The logical context is initialized and then extended at every level $i$ along the ASM computation.
- *Context initialization* describes the initial state

$$\text{contInit} = T_f(signature, 0), T_d(funcDefs, 0), T_d(funcInit, 0)$$

- *Context extension* represents the transition relation between states in $RS_i$ and their successor states in $RS_{i+1}$:

$$\text{contExt}_i = T_f(signature, i + 1), T_d(funcDefs, i + 1), T_r(r\_main, i), unchLocs_i$$

- $unchLocs_i$ keeps unchanged the functions that are not updated, e.g.,

$$\text{ASM model} \quad unchLocs_i$$

| if cond then | $f := f + 1$ | $(=> (not \ \text{cond}) (= f_{i+1} f_i)$ |
Representing ASM runs

An ASM run is given by a context initialization and a sequence of context extensions

\[ contInit, contExt_0, contExt_1, \ldots, contExt_n \]

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

;;; \( T_f(\text{signature, 0}) \): Functions declarations — state 0
(define level0::int)
(define full0::bool)
;;; \( T_d(\text{funcDefs, 0}) \): Derived functions definitions — state 0
(assert (= full0 (= level0 1000)))
;;; \( T_d(\text{funcInit, 0}) \): Initial state definition
(assert (= level0 0))

;;; \( T_f(\text{signature, 1}) \): Functions declarations — state 1
(define level1::int)
(define full1::int)
;;; \( T_d(\text{funcDefs, 1}) \): Derived functions definitions — state 1
(assert (= full1 (= level1 1000)))
;;; \( T_r(\text{r.main, 0}) \): Transition rules — from state 0 to state 1
(define cv0::(subrange −50 50)) ;; Declaration of a variable for the choose rule
(assert (= > (exists (x::(subrange −50 50)) (and (> = (+ level0 x) 0) (<= (= (+ level0 x) 1000))
(and (and (> = (+ level0 cv0) 0) (<= (= (+ level0 cv0) 1000)) (= level1 (+ level0 cv0)) )) ) ) )
;;; \( \text{unchLocs0} \): Unchanged controlled locations — from state 0 to state 1
(assert (= > (not (and (> = (+ level0 cv0) 0) (<= (= (+ level0 cv0) 1000)) (= level0 level1)) ) )

Arcaini, Gargantini, Riccobene
CoMA-SMT – Monitoring procedure

\[ o_C \leftarrow \textbf{new} \ C() \]  

▷ Monitored program: object instantiation
CoMA-SMT – Monitoring procedure

\[ o_C \leftarrow \text{new } C() \]
\[ ctx \leftarrow \text{mk\_context()} \]
\[ \text{add\_to\_context}(\text{contInit}, ctx) \]

- Monitored program: object instantiation
- Logical context creation
- Context initialization
- Monitored program: method execution
- Context extension at level
- Java values after step
- Observed values
- Assertion of observed values
- Is SAT?
- Return NotConformantException

Arcaini, Gargantini, Riccobene
CoMA-SMT – Monitoring procedure

\[ o_C \leftarrow \texttt{new } C() \]
\[ ctx \leftarrow \texttt{mk\_context}() \]
\[ \text{add\_to\_context}(contInit, ctx) \]
\[ i \leftarrow 0 \]
\[ \textbf{while true do} \]
\[ \text{add\_to\_context}(contExt_i, ctx) \]
\[ o_C.m() \]
\[ \textbf{end while} \]

- Monitored program: object instantiation
- Logical context creation
- Context initialization
- Monitored program: method execution
CoMA-SMT – Monitoring procedure

\[ o_C \leftarrow \textbf{new} \ C() \]

\[ ctx \leftarrow \text{mk\_context}() \]

\[ \text{add\_to\_context}(\text{contInit}, ctx) \]

\[ i \leftarrow 0 \]

\[ \textbf{while} \ \text{true} \ \textbf{do} \]

\[ o_C.m() \]

\[ \text{add\_to\_context}(\text{contExt}_i, ctx) \]

\[ \textbf{end while} \]

▷ Monitored program: object instantiation
▷ Logical context creation
▷ Context initialization

▷ Monitored program: method execution
▷ Context extension at level \( i \)
CoMA-SMT – Monitoring procedure

\[
\text{o}_C \leftarrow \text{new } C() \\
\text{ctx} \leftarrow \text{mk\_context()} \\
\text{add\_to\_context(} \text{contInit, ctx} \text{)} \\
i \leftarrow 0 \\
\text{while } \text{true} \text{ do} \\
\quad \text{\text{o}_C.m()} \\
\quad \text{add\_to\_context(} \text{contExt}_i, \text{ctx} \text{)} \\
\quad \text{sJava} \leftarrow |\text{o}_C| \\
\quad \text{javaValuesConstr} \leftarrow \text{getValues(} \text{sJava} \text{)}
\]

- Monitored program: object instantiation
  - Logical context creation
- Context initialization
  - Monitored program: method execution
  - Context extension at level \(i\)
- Observed Java state after step \(i\)
  - Observed values
- Is SAT?
  - Java state not conformant
    - return NotConformantException
CoMA-SMT – Monitoring procedure

\[
\begin{align*}
& o_C \leftarrow \textbf{new} \ C() \\
& ctx \leftarrow \text{mk\_context}() \\
& \text{add\_to\_context}(\text{contInit}, ctx) \\
& i \leftarrow 0 \\
& \textbf{while} \ \text{true} \ \textbf{do} \\
& \quad o_C.m() \\
& \quad \text{add\_to\_context}(\text{contExt}_i, ctx) \\
& & s_{Java} \leftarrow |o_C| \\
& & \text{javaValuesConstr} \leftarrow \text{getValues}(s_{Java}) \\
& & \text{add\_to\_context}(\text{javaValuesConstr}, ctx)
\end{align*}
\]

- Monitored program: object instantiation
- Logical context creation
- Context initialization
- Monitored program: method execution
- Context extension at level \(i\)
- Observed Java state after step \(i\)
- Observed values
- Assertion of observed values
CoMA-SMT – Monitoring procedure

\[ o_C \leftarrow \textbf{new } C() \]
\[ ctx \leftarrow \text{mk\_context()} \]
\[ \text{add\_to\_context}(\text{contInit}, ctx) \]
\[ i \leftarrow 0 \]
\[ \textbf{while } true \textbf{ do} \]
\[ \quad o_C.m() \]
\[ \quad \text{add\_to\_context}(\text{contExt}_i, ctx) \]
\[ \quad \text{sJava} \leftarrow |o_C| \]
\[ \quad \text{javaValuesConstr} \leftarrow \text{getValues(sJava)} \]
\[ \quad \text{add\_to\_context(javaValuesConstr, ctx)} \]
\[ \quad \textbf{if } \text{check}(ctx) = \text{UNSAT} \textbf{ then} \]
\[ \quad \quad \text{return NotConformantException} \]
\[ \quad \textbf{end if} \]
\[ \quad i \leftarrow i + 1 \]
\[ \textbf{end while} \]

- Monitored program: object instantiation
- Logical context creation
- Context initialization
- Monitored program: method execution
- Context extension at level \( i \)
- Observed Java state after step \( i \)
- Observed values
- Assertion of observed values
- Is SAT?
- Java state not conformant
Exp. – Univocal conformance – CoMA-SMT and CoMA

**Tank**

**Univocal**: the linking is based on the tank level

**Tic-Tac-Toe**

**Univocal**: the linking is based on the board configuration
Exp. – Univocal and multiple conformance – CoMA-SMT

**Tank**

**Univocal**: the linking is based on the tank level

**Multiple**: the linking is based on a boolean flag specifying whether the tank is full

**Tic-Tac-Toe**

**Univocal**: the linking is based on the board configuration

**Multiple**: the linking is based on a boolean flag specifying whether the game is terminated
Conclusions

- We propose an SMT-based approach (CoMA-SMT) for ASM-based runtime verification
  - It extends an exiting framework based on explicit state representation
- CoMA-SMT can deal with a more general notion of conformance in the presence of nondeterminism
- Optimizations are needed for keeping the size of the logical context small:
  - context simplification
  - stopping monitoring policies