Kalpa Publications in Computing

Volume 5, 2018, Pages 1–18

Automated Formal Methods

SMT for state-based formal methods: the ASM case study

Paolo Arcaini[1]*, Angelo Gargantini[2], and Elvinia Riccobene[3]

[1] Charles University, Faculty of Mathematics and Physics, Czech Republic
arcaini@d3s.mff.cuni.cz
[2] DIGIP, University of Bergamo, Italy
angelo.gargantini@unibg.it
[3] Department of Computer Science, University of Milan, Italy
elvinia.riccobene@unimi.it

Abstract

State-based transition systems can take advantage of a symbolic representation of the concepts of state and transition in order to automatically solve verification questions that could not be otherwise tackled in terms of explicit representation of the transition system. We report here our experience in developing solutions, approaches and supporting tools of verification problems regarding the Abstract State Machines (ASMs), a transition system which can be considered as an extension of Finite State Machines. We present the symbolic representation of an ASM and of its computational model in terms of the Yices SMT solver. We also discuss two scenarios of verification questions regarding the ASMs for which the symbolic representation helped us to formalize and solve the problem by satisfiability checking, namely automatic proof of correct ASM refinement and runtime verification.

1 Introduction

The architectural and behavioural complexity of modern systems and the need to guarantee critical properties yet at the early stages of the system life-cycle, require a rigorous development process based on the use of formal methods for system specification, validation and verification. Formal models can be used to understand if the system under development satisfies the given requirements (validation) and guarantees system properties (verification).

Specification can be given in operational style, when the system behaviour is expressed in terms of states and computation steps, or in declarative style, when the system is specified in terms of holding properties. There has been an endless debate about which style fits better the designer needs: some argue that with an operational style designers tend to insert implementation details in the abstract specifications, others observe that practitioners feel uncomfortable with declarative notations like temporal logics. Operational specifications are easier to write and understand; however, declarative specifications could be more suitable for verification purposes.

*The research reported in this paper has been partially supported by the Czech Science Foundation project number 17-12465S.

Abstract State Machines (ASMs) [15] are a state-based operational formal method that has been widely used as a system engineering method in different contexts: definition of industrial standards for programming and modeling languages, design of industrial control systems, modeling service and cloud systems, design and analysis of protocols, architectural design, language design, verification of compilers, etc.

Originally defined by Y. Gurevich in 1993 with the goal to sharpen the Church-Turing thesis [19], along the years, ASMs have been used in the field of software engineering to model systems and investigate their properties. To this aim, the usage of such formalism provides benefits under several viewpoints. If we consider *expressive power*, ASMs represent a general model of computation where the static view of a system is represented by means of a mathematical algebra and its dynamics is given in terms of transition rules. Concerning *understandability*, ASMs provide a way to describe behavioral issues by means of pseudo-code working over abstract data structures; therefore, ASM models are easily understandable without strong mathematical skills. If we consider *methodological issues*, the ASM formalism is the basis of a rigorous development method based on the concept of a "ground model" representing a precise but concise high-level formalization of the system, and on the "refinement principle" that allows to capture all details of the system design by a sequence of refined models till the desired level of detail. For *analysis purposes*, the rigorous mathematical foundation of ASM models, allows the application of formal techniques for model validation and verification. From the *implementation point of view*, the simple pseudo-code can be translated into a high level programming language source code in a quite simple manner (even automatically [13]).

To facilitate the practical usage of ASMs as software engineering formal method and to overcome the lack of automated tool support and poor tools integration around ASMs, in the past we worked on the development of the ASMETA framework [8], allowing ASMs to be used in an efficient and tool supported manner during the entire software development life cycle. In [7, 9], we have also indicated the process usually followed to develop systems from the definition of informal requirements to code implementation and execution.

It was by developing specific model analysis approaches around ASMs that we came across the necessity of having a symbolic representation of an ASM and of its computational model. Although a representation of ASMs in the theorem prover PVS already exists [18], it requires some expertise in theorem proving in order to be used. Therefore, we preferred to explore the use of Satisfiability Modulo Theories (SMT) solvers that guarantee a higher degree of automation. Thanks this representation in SMT, we were able to reduce verification questions, that would have been unfeasible to solve automatically by means of explicit representation of ASM states and of ASM computations as explicit sequences of states, to SMT problems. In particular, proving correctness of ASM model refinement [6] and runtime verification through nondeterministic ASMs [5] were solved as SMT satisfiability checking. In both cases, the symbolic representation helps *to formalize the problem* in a very concise way by means of a *logical context* representing ASM states and computation steps, and *to solve the problem* as satisfability checking of the context.

In this position paper, we present the symbolic representation of an ASM and of its computational model in terms of a context of the Yices SMT solver. We then introduce the formalization of the two main ASM verification questions we tackled by the use of Yices: proof of the correct ASM refinement and runtime verification. Along the presentation, we discuss the problems we faced to move from an explicit representation of the machine state and its computation step to a symbolic one. We also motivate formalization choices made for reducing a computational problem to a satisfiability problem.

The paper is organized as follows. Sect. 2 briefly introduces the Abstract State Machines.

Sect. 3 presents the symbolic representation in Yices of an ASM and how to build an SMT context representing the initial state and the computation step from a given ASM model. Sect. 4 discusses the problem of reducing to an SMT problem satisfaction the question of verifying correctness of an ASM model refinement step. Sect. 5 introduces the problem of runtime verification using ASM nondeterministic models and shows how to formalize its solution in terms of a symbolic SMT problem. We conclude the paper with Sect. 6, where we discuss strengths and weaknesses of our symbolic representation of the ASMs and we outline possible further verification contexts where this representation could be exploited.

2 Abstract State Machines

Abstract State Machines (ASMs) [15] are an extension of FSMs, where unstructured control states are replaced by states with arbitrarily complex data.

ASM *states* are algebraic structures, i.e., domains of objects with functions and predicates defined on them. An ASM *location*, defined as the pair (*function-name*, *list-of-parameter-values*), represents the abstract ASM concept of basic object containers. The couple (*location*, *value*) represents a machine memory unit. Therefore, ASM states can be viewed as abstract memories.

Location values are changed by firing *transition rules*. They express the modification of functions interpretation from one state to the next one. Note that the algebra signature is fixed and that functions are total (by interpreting undefined locations $f(x)$ with value *undef*). Location *updates* are given as assignments of the form $loc := v$, where loc is a location and v its new value. They are the basic units of rules construction. There is a limited but powerful set of *rule constructors* to express: guarded actions (`if-then`), simultaneous parallel actions (`par`), sequential actions (`seq`), nondeterminism (existential quantification `choose`), and unrestricted synchronous parallelism (universal quantification `forall`).

An ASM *computation* is, therefore, defined as 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 the unique *main rule* which in turn could fire other transitions rules. An ASM can have more than one *initial state*.

During a machine computation, not all the locations can be updated. Functions are classified as *static* (never change during any run of the machine) or *dynamic* (may change as a consequence of agent actions or *updates*). Dynamic functions are distinguished between *monitored* (only read by the machine and modified by the environment) and *controlled* (read and written by the machine). A further classification is between *basic* and *derived* functions, i.e., those coming with a specification or computation mechanism given in terms of other functions. It is possible to specify state *invariants*.

An ASM can be *nondeterministic* due to the presence of monitored functions (*external* nondeterminism) and of choose rules (*internal* nondeterminism).

A set of tools exists to support the ASM modeling process. Tools are part of the ASMETA (ASM mETAmodeling) framework[1] [8], and are strongly integrated in order to permit reusing information about models during different development phases. ASMETA provides basic functionalities for ASM model editing, and supports advanced model analysis techniques (as validation, verification, testing, model review, runtime verification, refinement proof, etc.).

[1]`http://asmeta.sourceforge.net/`

ASM domains declarations	**Yices**
enum domain D = $\{E_1, \ldots, E_n\}$	(**define−type** D (scalar $E_1 \ldots E_n$ D_undef))
Boolean	bool
Integer	int
Natural	nat

Table 1: $\mathtt{T_{dom}}$: Mapping schema of ASM domains to Yices

3 ASM symbolic representation

An SMT problem is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality. An SMT instance is a generalization of a boolean SAT instance in which various sets of variables are replaced by predicates from a variety of underlying theories. SMT solvers can be used, as in our case, as automatic theorem provers by checking unsatisfiability. We use Yices [16] as SMT solver.

The following sections describe the mapping (in terms of *mapping functions*) from ASM models to Yices elements. The theories needed for the mapping are: uninterpreted functions, and linear and non-linear integer and real arithmetic.

The way to use the Yices elements obtained by such mapping depends on the technique that uses them; we will give examples of these usages in Sects. 4 and 5 in which we describe how we exploited the ASM symbolic representation for proving ASM refinement correctness and for runtime verification.

We first describe the mapping of the signature, of terms, of function definitions, and of transition rules. Then, we describe how to exactly capture the semantics of an ASM computation step, and we provide the complete description of the mapping of the initial state and of a generic step. We finally provide an example of translation of a simple ASM model. The tool implementing the translator is available online[2].

Signature

The signature is given by the domains and the function declarations.

The mapping function $\mathtt{T_{dom}}$ from ASM domains to Yices types is reported in Table 1. For each ASM enumerative domain, we define a Yices scalar type. Basic type-domains Boolean, Integer, and Natural have a straightforward mapping to a corresponding type in Yices. For each domain, we also provide a constant representing the *undef* value; for each enumerative domain D we add the constant D_undef, while for the Integer an Natural domains we select as undef a value of the domain that cannot be assumed by any function of the model[3]. We are not able to provide an undef value for the Boolean domain and, therefore, we require boolean functions to not assume the undef value. An alternative solution could be to model the Boolean domain as a three-valued enumerative domain, but this would greatly complicate the translation. Moreover, since in ASMs rule guards are *formulas* [15] (note 51 at page 64) that need to be always defined, boolean functions used in rule guards cannot be undef: therefore, the limitation of the mapping is limited to boolean functions not used in guards.

Table 2 reports the mapping $\mathtt{T_f}$ from ASM function declarations to the corresponding Yices definitions in terms of uninterpreted functions. Note that ASM function declarations must

[2]The tool can be downloaded from `http://asmeta.sourceforge.net/download/asm2SMT.html`.
[3]Note that, in general, we cannot statically determine such value.

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

Table 2: `T_f`: Mapping schema of the ASM function declarations to Yices at state i

be translated for a generic state i, since they could be added to the logical context multiple times for different states: for this reason, `T_f` is parametrized with the state i that is reported in the mapped function name. Both 0-ary functions and n-ary functions (with $n > 0$) have a straightforward representation in terms of Yices definitions. Note that there is no difference in the mapping of the declaration of different types of ASM functions (i.e., controlled, monitored, derived, static); only the way to determine their values is mapped in different ways. Static and derived functions have their own definition mapped as Yices definitions (see next but one section), while updates of controlled functions are mapped in the translation of transition rules. In ASMs, monitored functions are not updated by transition rules, but their value is determined by the environment; therefore, their mapping does not require anything but the declaration: in this way, they can assume any value of their domain. This allows us to easily model the *external* nondeterminism due to the environment.

Terms

Both function definitions and transition rules (whose mapping is described in the following two sections) contain terms. Table 3 shows the mapping `T_t` that maps ASM terms in Yices. Since terms can contain function names, `T_t` is parametrized with state i.

The mapping of Boolean, Natural, Integer, and enumeration terms is straightforward. An ASM location term is mapped as a Yices function application, where function arguments are the translation of the location parameters values. The conditional term (7th row in Table 3) is mapped in a conditional expression. Forall and exists terms are respectively mapped as conjunction and disjunction of the condition *cond* instantiated over all the possible tuples d_1, ..., d_n. Although in ASMs the two terms can quantify over infinite domains, the AsmetaL language requires domains D_1, ..., D_n to be finite: therefore, the described mapping in SMT is feasible, although the obtained formula could be particularly big in case the domains are big[4]. Note that we could have mapped these terms using the Yices `forall` and `exists` quantified expressions; however, since Yices is not complete when quantifiers are used[5], a Yices context containing quantifiers often cannot be evaluated by the solver (it returns the `unknown` result). Therefore, we preferred to adopt this more verbose mapping that, however, can be always evaluated.

Derived and static functions

In ASMs, besides dynamic functions, the user can introduce derived functions defined by a law over the current state and static functions defined by a law over a number of parameters. These

[4]However, since usually the translation in SMT is done for verification purposes, in order to keep the execution time reasonable the domains are usually not too big.

[5]`http://yices.csl.sri.com/old/language.html#language_quantifiers`

ASM term	**Yices**
Boolean term: b with $b \in \{\mathsf{true}, \mathsf{false}\}$	b
Integer term: h with $h \in \mathbb{Z}$	h
Natural term: $h\mathsf{n}$ with $h \in \mathbb{N}$	h
Enumeration term: E	E
Location term: f	f^i
Location term: $\mathsf{f}(\mathsf{a}_1, \ldots, \mathsf{a}_n)$ with $n \geq 1$	$(\mathsf{f}^i\ \ \mathtt{T_t}(\mathsf{a}_1, i) \ldots \mathtt{T_t}(\mathsf{a}_n, i))$
if guard **then** Tthen **else** Telse **endif**	(**if** $\mathtt{T_t}$(guard, i) $\mathtt{T_t}$(Tthen, i) $\mathtt{T_t}$(Telse, i))
(**forall** x_1 **in** D_1, ..., x_n **in** D_n **with** cond[x_1, ..., x_n])	(**and** $\mathsf{c}_1 \ldots \mathsf{c}_m$) with $m = \prod_{j=1}^{n} \lvert D_j \rvert$ where for each $\overline{d}_k = (d_1, \ldots, d_n) \in D_1 \times \ldots \times D_n$: $\mathsf{c}_k = \mathtt{T_t}(\mathsf{cond}[x_1 \mapsto d_1, \ldots, x_n \mapsto d_n], i)$
(**exists** x_1 **in** D_1, ..., x_n **in** D_n **with** cond[x_1, ..., x_n])	(**or** $\mathsf{c}_1 \ldots \mathsf{c}_m$) with $m = \prod_{j=1}^{n} \lvert D_j \rvert$ where for each $\overline{d}_k = (d_1, \ldots, d_n) \in D_1 \times \ldots \times D_n$: $\mathsf{c}_k = \mathtt{T_t}(\mathsf{cond}[\mathsf{x}_1 \mapsto \mathsf{d}_1, \ldots, \mathsf{x}_n \mapsto \mathsf{d}_n], i)$

Table 3: $\mathtt{T_t}$: Mapping schema of ASM terms to Yices at state i

ASM function definition	**Yices**
function f = fd	$(= \mathsf{f}^i\ \mathtt{T_t}(\mathsf{fd}, i))$
function f(x_1 **in** D_1, ..., x_n **in** D_n) = fd[x_1, ..., x_n] with $n \geq 1$ and $\mathrm{D}_1 = \{\mathrm{d}_1^1, \ldots, \mathrm{d}_{m_1}^1\}$ ... $\mathrm{D}_n = \{\mathrm{d}_1^n, \ldots, \mathrm{d}_{m_n}^n\}$	(**and** $(= \mathtt{T_t}(\mathsf{f}(\mathsf{d}_1^1, \ldots, \mathsf{d}_1^n), i)$ $\mathtt{T_t}(\mathsf{fd}[\mathsf{x}_1 \mapsto \mathsf{d}_1^1, \ldots, \mathsf{x}_n \mapsto \mathsf{d}_1^n], i)\ )$ ... $(= \mathtt{T_t}(\mathsf{f}(\mathsf{d}_{m_1}^1, \ldots, \mathsf{d}_{m_n}^n), i)$ $\mathtt{T_t}(\mathsf{fd}[\mathsf{x}_1 \mapsto \mathsf{d}_{m_1}^1, \ldots, \mathsf{x}_n \mapsto \mathsf{d}_{m_n}^n], i)\ )\ )$

Table 4: $\mathtt{T_d}$: Mapping schema of ASM function definitions to Yices at state i

functions cannot be updated by transition rules, but they come with their own definition (i.e., they are functions in the mathematical sense). Their definition is mapped by $\mathtt{T_d}$ as shown in Table 4. For a 0-ary function, the mapping simply consists in creating equality of the function name with the translation of the function body (using the mapping function of terms $\mathtt{T_t}$ shown in Table 3). For n-ary functions (with $n > 0$), instead, the mapping consists in the conjunction of the mapping of the single locations. Note that, also in this case, we could have mapped these function definitions using the Yices `forall` quantified expression; however, as seen before, we could risk to obtain a formula that cannot be always evaluated.

Transition rules

Table 5 reports the mapping function $\mathtt{T_r}$ for transitions rules. It produces a formula that sym-

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

Table 5: $\mathtt{T_r}$: Mapping schema of ASM transition rules to Yices at state i

bolically represents the ASM transition relation. The formula is built by recursively applying the mapping starting from the main rule (last row in the table); the formula is usually **assert**ed in the Yices context in order to represent a generic ASM step (more details on this will be given in Sect. 3.2).

In an *update rule* (first row of Table 5), the location term on the left-hand side of the rule refers to the *next* state, while the term on the right-hand side of the rule refers to *current* state. For this reason, the updated term is mapped with parameter $i+1$, whereas the term on the right side is mapped with parameter i.

An ASM *parallel rule* for the parallel execution of a set of rules, is mapped as conjunction of the mappings of the single rules.

A complete ASM *conditional rule* (with the else branch) is mapped using the Yices conditional expression, while a partial conditional rule (without the else branch) is mapped as an implication.

An ASM *forall rule* (5th row in Table 5) requires that rule R is executed in parallel with all the values of $x_1 \ldots x_n$ that make *guard* true. The rule is mapped as a conjunction of implications stating that if *guard* evaluated with values $d_1 \ldots d_n$ holds, then also the mapping of rule R (instantiated with values $d_1 \ldots d_n$) must hold. As said before for the forall term and for function definitions, we could use the Yices `forall` quantifier for the translation: however, the obtained Yices context would often produce the `unknown` result when evaluated.

An ASM *choose rule* requires that rule R is executed once with some values of $x_1 \ldots x_n$ (nondeterministically chosen) that make *guard* true (if any). The mapping (6th row in Table 5) first creates a definition cv_j^i for each logical variable x_j of the choose rule[6]; then, it expresses the choose rule through an implication stating that if a tuple of values $d_1 \ldots d_n$ exists that make the *guard* true, then the translation of the *guard* and of R must hold. Note that the nondeterminism of the choose rule semantics (*internal* nondeterminism) is compactly embedded in the Yices symbolic representation: any model of the Yices formula (in terms of $cv_1^i \ldots cv_n^i$) is a tuple $d_1 \ldots d_n$ that allows to execute R[7]: we exploit this feature in the runtime verification of nondeterministic systems described in Sect. 5.

An alternative version of the choose rule allows to specify a rule Ro that must be executed if R cannot be executed because *guard* cannot be true. The mapping in Yices (7th row in Table 5) is similar to the previous one, but a conditional expression is used instead of the implication: the mapping of rule Ro is specified as else expression.

Similarly to what happens in the translation of the forall and exists terms, the size of the formula obtained by the translation of the forall and choose rules grows with the size of domain $D_1, \ldots, D_n$; however, the translation is always feasible since the domains are required to be finite by the starting notation AsmetaL.

3.1 Computation step semantics

In ASMs, a computation step is performed by evaluating the transition rules (starting from the main rule), collecting (in the *update set*) all the updates of controlled locations, and applying the updates. Controlled locations that are not updated keep their value unchanged. The formula obtained by applying the mapping function $\mathtt{T_r}$ to the transition rules (Table 5) correctly determines the update set, but does not guarantee this latter condition. Therefore, for each controlled location f^i, we add the following formula

$unchLoc_f^i =$ (=> (**not** (**or** $\mathsf{guard}_1 \ldots \mathsf{guard}_n$)) (= f^{i+1} f^i))

being $\mathsf{guard}_1, \ldots, \mathsf{guard}_n$ the conditions upon which f^i is updated, and f^{i+1} the location in the next state. Conditions $\mathsf{guard}_1, \ldots, \mathsf{guard}_n$ are statically derived from the transition rules that lead to the updates of the location. Let CF_i be all the controlled locations of the ASM model at level i; we define the following formula that asserts the condition for all the locations in CF_i:

$$unchLocs_i = \bigwedge_{f \in CF_i} unchLoc_f^i$$

3.2 Initial state and generic step

We here show how the initial state and a generic step can be represented by using the mapping described in the previous sections.

Let us consider an ASM $M = \langle sig, funcDefs, funcInit, r\_main \rangle$, where sig is the signature, $funcInit = \{fi_1, \ldots, fi_p\}$ the function initializations and $funcDefs = \{fd_1, \ldots, fd_q\}$ the function definitions. We define predicates $init$ and $step_i$ to formalize the initial state and the generic step i of the machine, as follows:

[6] Note that also the definitions for choose rules variables must be parametrized by the state i, since they must be created for each step asserted in the logical context.

[7] Actually, in order to fully describe the semantics of the ASM choose rule, we need to avoid that the mapping of R holds when the *guard* cannot be true. This is guaranteed by additional formulas that state that locations that are not updated do not have to change their value, as explained in Sect. 3.1.

```
asm Tank
import StandardLibrary

signature:
  domain Level subsetof Integer
  domain IncrDom subsetof Integer
  dynamic controlled level: Level
  derived full: Boolean

definitions:
  domain Level = {0..50}
  domain IncrDom = {−3..3}
  function full = (level = 50)

  main rule r_Main =
    choose $x in IncrDom with level + $x >= 0 and level + $x <= 50 do
      level := level + $x

default init s0:
  function level = 0
```

Code 1: ASM model of the Tank case study

init = (**and** $\mathtt{T_d}(\mathsf{fi}_1)$ ... $\mathtt{T_d}(\mathsf{fi}_p)$)
step_i = (**and** $\mathtt{T_r}(\mathsf{r\_main}, i)$ $\mathit{unchLocs}_i$ $\mathtt{T_d}(\mathsf{fd}^i_1)$ ... $\mathtt{T_d}(\mathsf{fd}^i_q)$)

The initial state is determined by the mapping of the controlled function initializations. A generic step is determined by the mapping of the main rule, by the condition on the controlled locations that do not have to change their value, and by the definition of the derived functions at state i[8].

3.3 Example of SMT translation

As an example, we consider a tank that can be either filled or emptied. At every step, the tank level can be increased/decreased of up to 3 units of product. The tank is full when it contains 50 units of product. Such tank can be modeled by a simple ASM, as shown in Code 1. The function level records the number of units in the tank; in the initial state the tank is empty. Boolean function full signals whether the tank is full; the function is derived because its value depends on the value of function level. In the main rule, a choose rule nondeterministically increments/decrements the level of the tank of at most three units at a time, not exceeding the maximum capacity. Code 2 shows the translation of the ASM code in Yices. The Yices context contains the definition of the initial state and of a step starting from the initial state. Note that such context represents all the possible computations that can be done in one step.

[8]Note that transition rules could read the value of derived functions at state i.

```
;; signature state 0
(define level0::(subrange 0 50))
(define full0::bool)
;; initial state
(assert (= level0 0))
;; logic variables for choose rule − step 0
(define x0State0::(subrange −3 3))
;; signature state 1
(define level1::(subrange 0 50))
;; step 0
(assert (and
          ;; transition rules
          (=> (or
                (and (>= (+ level0 −3) 0) (<= (+ level0 −3) 50))
                (and (>= (+ level0 −2) 0) (<= (+ level0 −2) 50))
                (and (>= (+ level0 −1) 0) (<= (+ level0 −1) 50))
                (and (>= level0 0) (<= level0 50))
                (and (>= (+ level0 1) 0) (<= (+ level0 1) 50))
                (and (>= (+ level0 2) 0) (<= (+ level0 2) 50))
                (and (>= (+ level0 3) 0) (<= (+ level0 3) 50))
            (and
              (and (>= (+ level0 x0State0) 0) (<= (+ level0 x0State0) 50))
              (= level1 (+ level0 x0State0))))
          ;; unchanged locations
          (=>
            (not (and (>= (+ level0 x0State0) 0) (<= (+ level0 x0State0) 50)))
            (= level0 level1)
          )
          ;; derived functions definitions
          (= full0 (= level0 50))
        )
)
```

Code 2: Yices context of the Tank model (one simulation step)

4 Refinement proof

One of the key concepts of the ASM method is *model refinement* that prescribes that a complete system model should be obtained through a chain of refined models: starting from a high-level abstract description of the system, more precise models should be obtained by iteratively adding more details (possibly reaching a model that is very close to the implementation). A notion of correct refinement has been originally presented in [14]; given an abstract model A and a refined model R, a proof of correct refinement must be able to associate each R-run with some A-run, according to some desired schema (1-1, 1-m, or n-m) and a conformance relation between abstract and refined states. Note that refinement proof checks a property related to the *construction* of the model, i.e., that the model has been correctly refined. Such property is independent of the particular model; model-dependent properties regarding the *behaviour*

$$\begin{tikzcd}
A & S_0 \arrow[r] & S_1 \arrow[rrr] & & & S_2 \arrow[r] & S_3 \\
R & \widetilde{S}_0 \arrow[r] \arrow[u, Rightarrow, "\equiv"'] & \widetilde{S}_1 \arrow[r] \arrow[u, Rightarrow, "\equiv"'] & \widetilde{S}_2 \arrow[r] \arrow[ul, dotted, "\equiv"] & \widetilde{S}_3 \arrow[r] \arrow[ull, dotted, "\equiv"'] & \widetilde{S}_4 \arrow[r] \arrow[u, Rightarrow, "\equiv"'] & \widetilde{S}_5 \arrow[u, Rightarrow, "\equiv"'] \\
 & \underbrace{\quad}_{\rho_0} & & \underbrace{\qquad\qquad\qquad}_{\rho_1} & & \underbrace{\quad}_{\rho_2} & \underbrace{\quad}_{\rho_3}
\end{tikzcd}$$

Figure 1: Stuttering refinement: relation between runs

of the model should be specified using temporal logics and verified using the ASMETA model checker [2].

Automatically proving the original notion of refinement would require to explicitly represent ASM runs of the two machines and compare them using, for example, techniques as model checking. However, developing a technique able to prove any refinement schema for any model is almost impossible. Actually, in our experience in modeling with ASMs [7, 9, 1], we observed that a particular type of 1-m refinement occurs. We called it *stuttering refinement*:

Definition 1 (Stuttering Refinement)**.** An ASM R is a correct *stuttering refinement* of an ASM A if and only if each R-run can be split in a sequence of subruns $\widetilde{\rho}_0, \widetilde{\rho}_1, \ldots$ and there is an A-run $S_0, S_1, \ldots$ such that for each $\widetilde{\rho}_i$ it holds $\forall \widetilde{S} \in \widetilde{\rho}_i \colon \mathit{conf}(\widetilde{S}, S_i)$.

Predicate *conf* formalizes the conformance relation between states of the abstract machine and states of the refined machine:

Definition 2 (Conformance)**.** Let S be a state of the abstract machine A (called *abstract state*), $\widetilde{S}$ a state of the refined machine R (called *refined state*). The two states are *conformant* iff corresponding *locations of interest* have equivalent values, i.e.,

$$\mathit{conf}(\widetilde{S}, S) \quad \text{iff} \quad \forall l_R \forall l_A\, \mathit{corrLoc}(l_R, l_A) \rightarrow [\![l_R]\!]_{\widetilde{S}} = [\![l_A]\!]_S$$

where *corrLoc* is a one-to-one correspondence between the locations of interest (i.e., locations on which compare the states) of A and R.

Fig. 1 shows the correspondence between a refined R-run and an abstract A-run.

Proving refinement with SMT

Proving stuttering refinement, differently from the more general notion of refinement [14], does not require to explicitly represent the abstract and refined runs, but can be reduced to the proof of two properties, similarly to what is done in [21] for compiler verification.

Theorem 1. *If the following properties hold*

$$\forall \widetilde{S} \colon (\mathit{init}(\widetilde{S}) \rightarrow \exists S \colon (\mathit{init}(S) \wedge \mathit{conf}(\widetilde{S}, S))) \tag{1}$$

$$\forall \widetilde{S} \forall \widetilde{S}' \forall S \colon \left[\begin{pmatrix} \mathit{step}(\widetilde{S}, \widetilde{S}') \\ \wedge \\ \mathit{conf}(\widetilde{S}, S) \end{pmatrix} \rightarrow \begin{pmatrix} \exists S' \colon (\mathit{step}(S, S') \wedge \mathit{conf}(\widetilde{S}', S')) \\ \vee \\ \mathit{conf}(\widetilde{S}', S) \end{pmatrix} \right] \tag{2}$$

then R is a stuttering refinement of A. $\mathit{init}(S)$ holds iff S is an initial state, and $\mathit{step}(S, S')$ holds iff the state S' can be reached by S in one simulation step.

```
(define fr_1 :: Dr_1) ... (define fr_m :: Dr_m)
(define init_R :: bool init_R(f̄_R))
(define existsInit_A :: bool (exists ((fa_{L+1} Da_{L+1}) ... (fa_n Da_n)) init_A(f̄^c_R + f̄^{nc}_A) ) )
(assert (not (=> init_R existsInit_A) ) )
(check)
```

Code 3: Yices context for proving initial refinement

We name property (1) as *initial refinement*, and property (2) as *step refinement.* Such properties are expressed in terms of states of the abstract and refined machines; we must reduce them to a symbolic representation in order to be able to express them using our encoding. In order to do this, we give the following definitions.

Let $\bar{f}_A = [fa_1, \dots, fa_n]$ be the functions of the abstract machine A and $\bar{f}'_A = [fa'_1, \dots, fa'_n]$ their renamed copy in the next state. In the same way, $\bar{f}_R = [fr_1, \dots, fr_m]$ and $\bar{f}'_R = [fr'_1, \dots, fr'_m]$ are functions of the refined machine R. In these lists, the first L functions are the locations of interest. We split a list of functions $\bar{f}$ between the functions corresponding to locations of interest and those which are not related in the conformance relation: $\bar{f} = \bar{f}^c + \bar{f}^{nc}$.

Now, we can express the predicates *init*, *step*, and *conf* used in Thm. 1, in terms of the function lists of the abstract and refined machines, and rewrite Formulas 1 and 2 as follows[9]:

$$init_R(\bar{f}_R) \to (\exists \bar{f}^{nc}_A \colon init_A(\bar{f}^c_R + \bar{f}^{nc}_A)) \tag{3}$$

$$step_R(\bar{f}_R, \bar{f}'_R) \to \begin{pmatrix} \exists \bar{f}^{nc\prime}_A \colon step_A(\bar{f}^c_R + \bar{f}^{nc}_A, \bar{f}^{c\prime}_R + \bar{f}^{nc\prime}_A) \\ \vee \\ \bar{f}^{c\prime}_R = \bar{f}^c_R \end{pmatrix} \tag{4}$$

The two properties can be mapped in Yices using the mapping described in Sect. 3. Therefore, the automatic proof of stuttering refinement can be reduced to a satisfability checking problem i.e., to two Yices contexts[10]. The two symbolic properties have two useful characteristics w.r.t. the formulation in terms of states:

- the universal quantifications over states are substituted by the introduction of new function symbols (by Herbrandization); such formulation is suitable for checking validity with a logical solver;
- since conformance between abstract and refined states consists in the equality between functions having the same name in the abstract and in the refined machine, we can express the conformance relation by simply using only one copy of the variables. Such formulation is convenient, since it reduces the number of functions in the logical context and also the complexity of the formula, so reducing the complexity of the problem.

For Formula 3, we build the Yices context shown in Code 3. The antecedent of the implication is represented through the SMT definition init_R, and the consequent by definition $\mathsf{existsInit}_A$.

For Formula 4, we build the Yices context shown in Code 4. The antecedent of the implication is represented by definition step_R. The consequent is represented by the disjunction of

[9]In [6], we show how Formulas 3 and 4 are derived from Formulas 1 and 2.
[10]The tool implementation of the approach is available at http://asmeta.sourceforge.net/download/asmrefprover.html.

```
(define $\mathsf{fr}_1$ :: $\mathsf{Dr}_1$) ... (define $\mathsf{fr}_m$ :: $\mathsf{Dr}_m$)
(define $\mathsf{fr}'_1$ :: $\mathsf{Dr}_1$) ... (define $\mathsf{fr}'_m$ :: $\mathsf{Dr}_m$)
(define $\mathsf{fa}_{L+1}$ :: $\mathsf{Da}_{L+1}$) ... (define $\mathsf{fa}_n$ :: $\mathsf{Da}_n$)
(define $\mathsf{step}_R$ :: bool $\mathsf{step}_R(\bar{f}_R, \bar{f}'_R)$)
(define $\mathsf{existsStep}_A$::bool (exists (($\mathsf{fa}'_{L+1}$ $\mathsf{Da}_{L+1}$) ... ($\mathsf{fa}'_n$ $\mathsf{Da}_n$)) $\mathsf{step}_A(\bar{f}_R^c+\bar{f}_A^{nc}, \bar{f}_R^{c\prime}+\bar{f}_A^{nc\prime})$))
(define stutteringState :: bool (and (= $\mathsf{fr}'_1$ $\mathsf{fr}_1$) ... (= $\mathsf{fr}'_L$ $\mathsf{fr}_L$) ))
(assert (not (=> $\mathsf{step}_R$ (or $\mathsf{existsStep}_A$ stutteringState) ) ) )
(check)
```

Code 4: Yices context for proving step refinement

definitions $\mathsf{existsStep}_A$ and stutteringState; the latter one models the equality of vectors in the stuttering state (i.e., $\bar{f}_R^{c\prime} = \bar{f}_R^c$ in Formula 4) as a conjunction of equalities.

As usual, since we need to prove validity of formulas, we check that their negation is unsatisfiable. Therefore, if the two Yices instances are proved unsatisfiable, the refinement is proved correct. However, step refinement condition is sufficient but not necessary: when Formula 4 is proved not valid (i.e., the corresponding SMT instance is satisfiable), we cannot state that the refinement is not correct. Nevertheless, when refinement is not proved correct, the SMT solver returns a model that we can inspect to understand whether it is really the case that the refinement is not correct, or it is a false negative result (i.e., the model represents a state that is not reachable); in the latter case, we can strengthen the properties using an invariant that restricts the set of states that must be inspected in the proof [6].

5 Runtime verification

Another use of the encoding proposed in Sect. 3, is for runtime verification of Java code w.r.t. its ASM specification. This idea, implemented in a tool called CoMA [4], consists in observing the behaviour of a Java object O_C during its execution and checking that its observed elements conform to the expected state of its specification ASM_C. A suitable *link* must be established between O_C fields and pure methods (i.e., methods that cannot change O_C state) and the ASM_C state; this can be done by code annotation. This link identifies the conformance relation (in brief *conf*) among O_C and ASM_C states. Moreover, the designer must designate a set of methods, called *changing methods*, whose invocation corresponds to an ASM step. The runtime verification can be performed by checking that the following schema holds:

$$\begin{tikzcd}
(O_C) & (ASM_C) \\
s \arrow[r, dotted, "conf"] \arrow[d, "execution\ of\ changing\ method\ m"] & S \arrow[d, "simulation\ step"] \\
s' \arrow[r, dotted, "conf"] & S'
\end{tikzcd}$$

In case of deterministic behaviour (of both the specification and the implementation), runtime verification requires that the next state s' of O_C conforms to the next state S' of the specification. Conformance can be verified by simulation: if O_C conforms to ASM_C in the current state, observed the next state s' of O_C, the monitor checks that it conforms to the next state S' of ASM_C, which can be obtained by applying its transition rules. During simulation, the monitor stores and updates only the current state S of the specification.

$$\begin{tikzcd}
(O_C) & & & & (ASM_C) & & & \\
s \arrow[rrr, dotted, "conf"] \arrow[d, "m"] & & & \{S_1, \arrow[dll, "step"'] \arrow[d, "step"] \arrow[drr, "step"] & \dots & S_k\} \arrow[dll, "step"] \arrow[drr, "step"] & & \\
s' \arrow[rrr, bend right=10, "conf(s'{,}S_i')"'] \arrow[rrrrr, bend right=30, "conf(s'{,}S_j')"'] & S_1' & \dots & \mathbf{S_i'} & \dots & \mathbf{S_j'} & \dots & S_n'
\end{tikzcd}$$

Figure 2: Conformance checking with nondeterminism

In case of nondeterministic behaviours, runtime verification can become complex since the specification takes into account all the possible correct system evolutions. The nondeterminism due to monitored quantities (e.g., the system inputs or external actions), called *external*, is still easy to monitor: once these quantities are fixed by the environment, the system behaves deterministically. However, in most cases, the specification is *internally* nondeterministic, sometimes even when the system is deterministic. For instance, when the system makes random choices, the specification must be nondeterministic. Moreover, nondeterministic specifications can be useful when the modeler wants to be more abstract (with less implementation details) than the corresponding system. Nondeterminism in the behavioural specifications can simplify the representation of complex functionalities and achieve a better separation of concerns [11].

When the behaviour is nondeterministic, the monitor must ensure that the relation shown in Fig. 2 holds. Formally, it can be defined as follows.

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

Definition 4. Runtime conformance A class C is runtime conforming to its specification ASM_C if the following conditions hold:

1) the initial state s_0 of the computation of O_C conforms to *at least one* initial state S_0 of the computation of ASM_C, i.e., $\exists S_0$ initial state of ASM_C such that $\mathit{conf}(s_0, S_0)$;

2) for every change step (s, m, s') with s the current state of O_C, for each $S \in \mathit{confSet}(s)$ $\exists (S, S')$ step of ASM_C, such that $\mathit{conf}(s', S')$.

Runtime conformance can be checked as shown in Fig. 2: assuming the current Java state s is conformant with the ASM states $\mathit{confSet}(s) = \{S_1, \dots, S_k\}$; the Java state s' is produced by the execution of the method m at the state s; ASM states $S_1', \dots, S_n'$ are reachable in one step from $\mathit{confSet}(s)$; the Java state s' is checked conformant with at least one state S_j'.

Example 1. Let us consider the Tank case study. If the conformance link between the ASM specification and the Java program is only based on the value of function full (since the level value is not observable, for example), then the ASM is nondeterministic. At each step, the ASM has between 4 and 7 possible next states; at most one next state can have value *true* for full. Therefore, if the implementation is correct and the value of full is *false*, more than one of the possible next ASM states can be conformant with the implementation.

Algorithm 1 CoMA-SMT: monitoring procedure

```
1: (assert $init$)                                   ▷ Context initialization
2: $i \leftarrow 0$
3: while $o_C.m()$ is invoked do
4:     (assert $step_i$)                              ▷ Extend context at level $i$
5:     $s_{Java} \leftarrow [|o_C|]$                   ▷ Observed Java state after step $i$
6:     $javaValConstr \leftarrow$ getValues($s_{Java}$)  ▷ Get observed values
7:     (assert $javaValConstr$)                        ▷ Add observed values
8:     if (check) = $UNSAT$ then                       ▷ Is SAT?
9:         return $NotConformantException$
10:    end if
11:    $i \leftarrow i + 1$
12: end while
```

Dealing with this kind of monitoring in an explicit state approach would require to keep track of all the possible states to which the monitored system can be conformant. At the ith step of monitoring, the framework should record in the set $confSet(s_i)$ (see Def. 3) the ASM states reachable in i steps of simulation that are conformant with the current Java state (as proposed in [17]). If $confSet(s_i)$ becomes empty, then an error is found.

Exploiting the Yices representation of the ASMs, we reduce the runtime conformance checking in the presence of nondeterminism to the satisfiability checking of an STM problem. We symbolically represent the set of states RS_i reachable in i steps of the ASM execution, and the transition relation induced by the ASM transition rules between states in RS_i and their successor states in RS_{i+1}. These formulas establish the logical *context*.

Example 2. Let us consider the Tank case study. The ASM states reachable in one step can be symbolically described as $\mathsf{level} = 0 \wedge (\mathsf{level} - 3 \leq \mathsf{level}' \leq \mathsf{level} + 3)$, where level' represents the updated version of level.

Informally, in order to perform the conformance checking, we build a logical context with the initial state and then we extend the context by asserting a set of formulas stating the values of the observed elements in the implementation current state. Yices is used to check the satisfiability of the obtained context. If the context becomes unsatisfiable, then the implementation is not conformant. Alg. 1 depicts the monitoring procedure of the proposed approach (called CoMA-SMT).

At the beginning, the framework initializes the context (line 1) with the initial state (see Sect. 3.2). The monitoring consists of a never ending loop in which, when a Java changing method m is executed (line 3), the following actions are executed:

- the context is extended for describing the transition relation between ASM states at the current level i and the possible next states at level $i + 1$ (line 4) by $step_i$ as defined in Sect. 3.2;
- from the Java state s_{Java}, obtained after the changing method execution (line 5), and from the linking between the specification and the code, the framework builds the formula *javaValConstr* in which the linked ASM locations are forced to assume the actual values of the corresponding Java elements (line 6). Let $f_1^{i+1}, \ldots, f_g^{i+1}$ be the locations linked to Java fields or methods (i.e., the observed elements) and $v_1, \ldots, v_g$ the values of the linked fields and methods at state $i + 1$. Formula *javaValConstr* is built as follows:

$$(\textsf{and}\ (= \mathsf{f}_1^{i+1}\ v_1)\ \ldots\ (= \mathsf{f}_g^{i+1}\ v_g))$$

- formula *javaValConstr* is asserted in the logical context (line 7);
- the logical context is checked for satisfiability (line 8):
  - if the context is unsatisfiable, it means that the implementation is not conformant with the specification. In this case, the monitoring is interrupted by throwing an error message (line 9);
  - otherwise, if the context is still satisfiable, it means that the implementation is conformant and the monitoring can continue.

Experiments have shown that CoMA-SMT (i.e., the SMT-based runtime verification approach) has better performances than CoMA (i.e., the explicit state approach) when the degree of nondeterminism is high [5]. However, the introduced overhead is still not negligible (around 5 ms per step for the Tank case study); moreover, experiments have also shown that CoMA-SMT does not scale well when the logical context grows after each simulation step. Therefore, techniques should be devised in order to keep the size of the logical context (and, therefore, the computation time) reasonable.

6 Conclusion and Future work

We have presented a translation of ASMs to SMT that we used for two purposes, namely proving refinement and runtime conformance checking in the presence of nondeterminism. Our experience with SMT is very positive: in the refinement correctness prover, we only needed the formalization of a single step since the proof is inductive, while in the runtime checker we were able to represent (step by step) the reachable conformant states thanks to the support for incremental solving of the SMT solver. The use of the declarative style of SMT theories allowed us to easily deal with nondeterministic behaviours by not having to explicitly keep track of all the possible states. One problem we faced is how to express what does not change (see Sect. 3.1) and, in this, operational notations are still better than declarative approaches. Note that we could use BDDs to symbolically represent ASM states; however, for bounded model checking it has been shown that a SAT/SMT approach scales better [12, 20]. So, since both our runtime verification and refinement approaches have several commonalities with BMC, we believe that the SMT approach has still several advantages.

As future work, we plan to employ our encoding in other V&V activities. In [3], we presented an approach for doing *model review* of ASMs that checks some *meta-properties* that any model should guarantee. The approach is sound and complete but, since based on model checking, does not scale well. As future work, we plan to detect possible inconsistencies of the ASM model using the SMT representation (e.g., an inconsistent context is a signal of the presence of inconsistent updates); that approach could be not always complete: it could only find a meta-property violation, but fail in proving their validity. However, we believe that SMT would scale much better than the pure model checking approach and we plan to apply induction for proving meta-properties.

In [10], we presented a formalization of self-adaptive systems in terms of ASMs. One problem with self-adaptive systems is how to resolve conflicts of different adaptation scenarios at runtime; in general, among n adaptation scenarios that are applicable in a given state, $m \leq n$ scenarios can simultaneously fire without conflicts. We plan to exploit the SMT representation to find the maximum subset of adaptation scenarios that can simultaneously fire.

Finally, we plan to exploit the symbolic representation for checking model-dependent behavioural properties, in a classical formal verification approach like that presented in [22].

References

[1] Paolo Arcaini, Silvia Bonfanti, Angelo Gargantini, Atif Mashkoor, and Elvinia Riccobene. Integrating formal methods into medical software development: The ASM approach. *Science of Computer Programming*, pages –, 2017.

[2] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. AsmetaSMV: A way to link high-level ASM models to low-level NuSMV specifications. In Marc Frappier, Uwe Glässer, Sarfraz Khurshid, Régine Laleau, and Steve Reeves, editors, *Abstract State Machines, Alloy, B and Z*, volume 5977 of *Lecture Notes in Computer Science*, pages 61–74. Springer Berlin Heidelberg, 2010.

[3] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. Automatic review of abstract state machines by meta property verification. In César Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215*, pages 4–13, Langley Research Center, Hampton VA 23681-2199, USA, April 2010. NASA.

[4] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. CoMA: Conformance monitoring of Java programs by Abstract State Machines. In Sarfraz Khurshid and Koushik Sen, editors, *Runtime Verification*, volume 7186 of *Lecture Notes in Computer Science*, pages 223–238. Springer Berlin Heidelberg, 2012.

[5] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. Using SMT for dealing with nondeterminism in ASM-based runtime verification. *ECEASST*, 70, 2014.

[6] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. SMT-based automatic proof of ASM model refinement. In Rocco De Nicola and Eva Kühn, editors, *Software Engineering and Formal Methods: 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings*, Lecture Notes in Computer Science, pages 253–269. Springer International Publishing, Cham, 2016.

[7] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. Rigorous development process of a safety-critical system: from ASM models to Java code. *International Journal on Software Tools for Technology Transfer*, 19(2):247–269, 2017.

[8] Paolo Arcaini, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A model-driven process for engineering a toolset for a formal method. *Software: Practice and Experience*, 41:155–166, 2011.

[9] Paolo Arcaini, Roxana-Maria Holom, and Elvinia Riccobene. ASM-based formal design of an adaptivity component for a cloud system. *Formal Aspects of Computing*, 28(4):567–595, 2016.

[10] Paolo Arcaini, Elvinia Riccobene, and Patrizia Scandurra. Formal design and verification of self-adaptive systems with decentralized control. *ACM Trans. Auton. Adapt. Syst.*, 11(4):25:1–25:35, January 2017.

[11] Pieter Bekaert and Eric Steegmans. Non-determinism in conceptual models. In *Proceedings of the Tenth OOPSLA Workshop on Behavioral Semantics*, pages 24 – 34, 2001.

[12] Armin Biere. Bounded model checking. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 457–481. IOS Press, 2009.

[13] Silvia Bonfanti, Marco Carissoni, Angelo Gargantini, and Atif Mashkoor. Asm2C++: A Tool for Code Generation from Abstract State Machines to Arduino. In Clark Barrett, Misty Davies, and Temesghen Kahsai, editors, *NASA Formal Methods: 9th International Symposium, NFM 2017, Proceedings*, pages 295–301. Springer, 2017.

[14] Egon Börger. The ASM refinement method. *Formal Aspects of Computing*, 15(2):237–257, 2003.

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

[16] Bruno Dutertre and Leonardo de Moura. The Yices SMT solver. Technical report, SRI Available at `http://yices.csl.sri.com/tool-paper.pdf`, 2006.

[17] Yliès Falcone, Klaus Havelund, and Giles Reger. A tutorial on runtime verification. In Manfred Broy, Doron A. Peled, and Georg Kalus, editors, *Engineering Dependable Software Systems*, volume 34 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 141–175. IOS Press, 2013.

[18] Angelo Gargantini and Elvinia Riccobene. Encoding abstract state machines in PVS. In Yuri Gurevich, Philipp W. Kutter, Martin Odersky, and Lothar Thiele, editors, *Abstract State Machines - Theory and Applications: International Workshop, ASM 2000 Monte Verità, Switzerland, March 19–24, 2000 Proceedings*, volume 1912 of *Lecture Notes in Computer Science*, pages 303–322, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[19] Yuri Gurevich. Specification and validation methods. chapter Evolving Algebras 1993: Lipari Guide, pages 9–36. Oxford University Press, Inc., New York, NY, USA, 1995.

[20] Robert P. Kurshan. Verification technology transfer. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 46–64. Springer Berlin Heidelberg, 2008.

[21] Kedar S. Namjoshi, Giacomo Tagliabue, and Lenore D. Zuck. A witnessing compiler: A proof of concept. In Axel Legay and Saddek Bensalem, editors, *Runtime Verification: 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*, pages 340–345, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[22] Margus Veanes, Nikolaj Bjørner, Yuri Gurevich, and Wolfram Schulte. Symbolic bounded model checking of abstract state machines. *Int. J. Software and Informatics*, 3(2-3):149–170, 2009.