SMT-based automatic proof of ASM model refinement*

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

[1] Charles University in Prague, Faculty of Mathematics and Physics, Czech Republic
arcaini@d3s.mff.cuni.cz
[2] Dipartimento di Ingegneria, Università degli Studi di Bergamo, Italy
angelo.gargantini@unibg.it
[3] Dipartimento di Informatica, Università degli Studi di Milano, Italy
elvinia.riccobene@unimi.it

Abstract. Model refinement is a technique indispensable for modeling large and complex systems. Many formal specification methods share this concept which usually comes together with the definition of refinement correctness, i.e., the mathematical proof of a logical relation between an abstract model and its refined models.
Model refinement is one of the main concepts which the Abstract State Machine (ASM) formal method is built on. Proofs of correct model refinement are usually performed manually, which reduces the usability of the ASM model refinement approach. An automatic support to assist the developer in proving refinement correctness along the chain of refinement steps could be of extreme importance to improve, in practice, the adoption of ASMs.
In this paper, we present how the integration between the ASMs and Satisfiability Modulo Theories (SMT) can be used to automatically prove correctness of model refinement for the ASM method.

1 Introduction

Modeling is a fundamental activity of system life-cycle: models allow developers to reason about the systems under construction and represent central artifacts of their development. Building models of large and complex systems is, however, not an easy task since lots of requirements have to be taken into consideration.

To manage such a complexity, many specification methods share a modeling process based on *model refinement* [1]. It consists in developing models starting from a high-level description of the system and proceeding through a sequence of more detailed models each introducing, step-by-step, design decisions and implementation details. The concept of model refinement usually comes together with the definition of *refinement correctness*, i.e., the mathematical proof of a logical relation between an abstract model and its refined models.

* This work was partially supported by the Grant Agency of the Czech Republic project 14-11384S.

Model refinement is a key concept for the *Abstract State Machine* (ASM) formal method. The ASM modeling process is based on the concept of a *ground model* representing a precise but concise high-level system specification, and on the *refinement principle* that allows to capture all details of the system design by a sequence of refined machines to the desired level of detail, possibly to the code level. In [12,14], Börger presents the ASM refinement, discusses its characteristics compared to other refinement approaches, and provides the definition of correctness proofs, namely the guaranty that a machine is a correct refinement of an abstract machine.

In developing ASM specifications of different case studies [3,4,6,8], we have modeled through refinement and we have observed that (a) the usual refinement schema a modeler uses is a (1:n) refinement in which one step of the abstract machine corresponds to n steps of the refined machine; (b) each refinement step introduces very small changes, either in terms of data and of control structure; (c) along the chain of models, the proofs of refinement correctness are similar and often tedious to repeat. Such observations reinforced in us the idea, felt for a long time, of having a tool assisting the modeler along the refinement steps and being able to provide automatic proof of the refinement correctness.

A mechanized approach to prove correctness of the ASM refinement already exists [22]. It requires the encoding of an ASM model into dynamic logic, a deep knowledge of the KIV theorem prover and an active role of the modeler in conducting the proofs. The tool is not integrated in any existing framework for ASM model development and manipulation [17,7], thus this verification activity appears separated with respect to other activities on models and does not permit reusing information. Our goal is, instead, to have a prover of correct model refinement fully integrated into a framework for editing, simulating, validating and verifying ASM models, so to improve the practical usability of the ASM method. We cannot expect practitioners to have deep skills in theorem provers or verification strategies, and we are aware of the necessity to compensate these lacks with suitable mechanized support which hides the mathematical complexity of the proof obligations that model refinement requires.

By exploiting the symbolic representation of ASMs into Satisfiability Modulo Theories (SMT), already presented in [5] as part of an SMT-based technique for runtime verification, we here present an automatic approach where the proof of ASM refinement is performed by means of satisfability checking.

We introduce the definition of ASM stuttering refinement between two ASMs. It is a restricted form of the ASM model refinement defined in [12], but we have found it recurring in our modeling experience and shared with other formal approaches [2,20]. It has also the advantage of allowing the reduction of the ASM correct refinement problem to an SMT problem, since the proof strategy to guarantee stuttering refinement does not reason on possible corresponding subruns of the two machines, but on the concept of a state to be initial and to be in (transition) relation with another state. This SMT problem consists of two conditions (initial refinement and step refinement) that guarantee a machine to

be a stuttering refinement of an abstract machine. An SMT solver is used to prove the validity of such properties.

The paper is organized as follows. Sect. 2 briefly introduces the ASMs and their use when modeling through refinement. A running case study is used for exemplification purposes. In Sect. 3 we give our notion of refinement and in Sect. 4 we provide a technique for proving it. Sect. 5 presents the SMT encoding of the model refinement correctness problem. Sect. 6 gives a preliminary evaluation of the approach. Sect. 7 presents work related to the verification of correct refinement for ASMs, and Sect. 8 concludes the paper.

2 Abstract State Machines

Abstract State Machines (ASMs) [14] are an extension of FSMs, where unstructured control states are replaced by states with arbitrary complex data. The method has a rigorous mathematical foundation; however, a practitioner can understand ASMs as pseudo-code or virtual machines working over abstract data structures. We here give the necessary background to understand our approach.

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*. It is possible to specify state *invariants*.

During a machine computation, not all the locations can be updated. Indeed, 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.

ASMs allow modeling any kind of computational paradigm, from a *single* agent executing parallel actions, to distributed *multiple* agents interacting in a

```
asm LGS_GM

signature:
  enum domain HandleStatus = {UP | DOWN}
  enum domain DoorStatus =
      {CLOSED | OPENING | OPEN | CLOSING}
  enum domain GearStatus =
      {RETRACTED|EXTENDING|EXTENDED|RETRACTING}
  dynamic monitored handle: HandleStatus
  dynamic controlled doors: DoorStatus
  dynamic controlled gears: GearStatus

definitions:
  rule r_closeDoor =
    switch doors
      case OPEN: doors := CLOSING
      case CLOSING: doors := CLOSED
      case OPENING: doors := CLOSING
    endswitch
```

```
rule r_retractionSequence =
  if gears != RETRACTED then
    switch doors
      case CLOSED: doors := OPENING
      case CLOSING: doors := OPENING
      case OPENING: doors := OPEN
      case OPEN:
        switch gears
          case EXTENDED:
            gears := RETRACTING
          case RETRACTING:
            gears := RETRACTED
          case EXTENDING:
            gears := RETRACTING
        endswitch
    endswitch
  else
    r_closeDoor[]
  endif
```

```
main rule r_Main =
  if handle = UP then
    r_retractionSequence[]
  else
    r_outgoingSequence[]
  endif

default init s0:
  function doors = CLOSED
  function gears = EXTENDED
```

Code 1. Landing Gear System – Abstract model

synchronous or asynchronous way. Moreover, 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[4] [7], and are strongly integrated in order to permit reusing information about models during different development phases. ASMETA provides basic functionalities for ASM models creation and manipulation (as editing, storage, interchange, access, etc.), and supports advanced model analysis techniques (as validation, verification, testing, model review, requirements analysis, runtime verification, etc.).

Example 1 (Landing Gear System case study). We here consider, as supporting case study, the Landing Gear System [11] (LGS), which is the airplane component responsible for the maneuvering of the landing gears and associated doors. The system can be in *nominal* mode or in *emergency* mode. In nominal mode, a landing sequence is: opening of the doors of the landing gear boxes, extension of the landing gears, and closing of the doors. The system also elaborates health parameters for all the equipments and, if necessary, switches to emergency mode.

Model LGS_GM (shown in Code 1) specifies the system behavior at a very abstract level: we only represent the statuses of the gears and of their doors and how they change in the retraction and outgoing sequences. Although there are three landing sets, we abstract and we model all of them as one. Functions `doors` and `gears` represent the status of the doors and of the gears, respectively. The state transitions are driven by the value of the monitored function `handle`. As long as `handle` is `UP`, the *retraction sequence* is executed; when `handle` is `DOWN`, the *outgoing sequence* is executed. Let us see how the retraction sequence works. We assume `handle` to be `UP` in each state. In the initial state, the `doors` are `CLOSED` and the `gears` are `EXTENDED`; then the `doors` start `OPENING`. When the `doors` become `OPEN`, the `gears` start `RETRACTING`. When the `gears` become `RETRACTED`, the `doors` start `CLOSING`. The retraction sequence terminates with

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

the `doors CLOSED` and the `gears RETRACTED`. The outgoing sequence behaves similarly. Note that a retraction (resp. an outgoing) sequence can be always interrupted by switching the value of the `handle`; in this case, an outgoing (resp. a retraction) sequence begins, starting from the status of the `doors` and the `gears` reached in the previous sequence.

2.1 ASM modeling through refinement

Modeling by ASMs starts by developing an initial abstract model, called *ground model*, which is a precise and concise high-level system description and can be considered as reference model for the further steps of the design. Model LGS_GM shown in Code 1 is an example of ground model.

Modeling proceeds by *model refinement*, namely by a chain of step-wise refined models, starting from the ground model. At each refined level, further details are added to capture the major design decisions and provide descriptions of the complete software architecture and component design of the system. The end point of the chain is decided by the designer, and it should be a model detailed enough to be mapped into executable code or at least a model against which the code can be automatically tested for conformance checking.

Several examples [9,21,24] show the applicability of this approach which permits to keep the complexity of the system under control, and to bridge, in a seamless manner, the gap between specification and code.

In model refinement, a key point is to prove that a refined model is correct w.r.t. the abstract one. For ASMs, the original description of the refinement method and the definition of correct model refinement are due to Börger in [12]. That definition of refinement is very general and makes it difficult to prove refinement correctness in an automatic way or, at least, to find proof patterns.

3 Stuttering Refinement

We here define stuttering refinement between two ASMs, which is a restricted form of the ASM model refinement as in [12] (a comparison is given in Sect. 7). This notion of refinement allows us to provide an automatic approach to refinement proof, based on a logic representation of ASM signatures and transition rules. We consider deterministic and nondeterministic single-agent ASMs.

As stated in Sect. 2, a state S of an ASM M is a set of locations with value. We here denote by $val(l, S)$ the value of a location l at state S.

A model refinement first requires the definition of corresponding *locations of interest*, i.e., pairs of (possibly sets of) locations one wants to relate in corresponding abstract and refined states.

Definition 1 (Corresponding Locations of Interest). *Given two ASMs A and R, we denote by* corrLoc *the correspondence over the set of locations of interest of the refined machine R and their corresponding locations of the abstract machine A, i.e., $corrLoc(l_R, l_A)$ is true iff l_R is a location of interest in R and l_A is its unique corresponding location in A.*

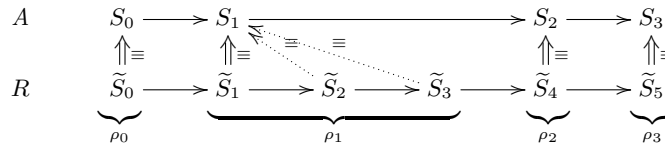

Fig. 1. Stuttering refinement – Relation between a refined run and an abstract run

On the base on the corresponding locations of interest, we define *conformant states* between the abstract and refined machines, namely states having equivalent values for the corresponding locations of interest. Obviously, a notion of equivalence $\equiv$ of the data in the locations of interest is assumed available.

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

$$conf(\widetilde{S}, S) \quad \text{iff} \quad \forall l_R \forall l_A \, corrLoc(l_R, l_A) \rightarrow val(l_R, \widetilde{S}) \equiv val(l_A, S)$$

Typically, *corrLoc* is a one-to-one correspondence between the locations of interest of A and R and the designer uses the same function symbols to denote these corresponding locations. We here assume – as in [16] – linked locations to have the same names and equality as equivalence relation. More complicated conformance relations can be easily reduced to this simplified form by introducing convenient derived functions representing predicates over the abstract or refined states. Suppose to have a function fuelStatus in the abstract machine defined over the domain {NORMAL, RESERVE} and that this function is refined by the function fuelLevel defined over the interval [1,30]. The two specifications can be linked by introducing in the refined machine a derived function fuelStatus that specifies the desired conformance relation (e.g., fuelStatus = **if** fuelLevel > 10 **then** NORMAL **else** RESERVE **endif**).

Once the notions of corresponding locations of interest and of state conformance have been determined, one can define that R is a correct stuttering refinement of A as follows:

Definition 3 (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 $\forall \widetilde{S} \in \widetilde{\rho}_i$: $conf(\widetilde{S}, S_i)$.*

Note that infinite R-runs can be split in an infinite number of finite subruns, or in a finite number of subruns where only the last one is infinite. Fig. 1 depicts a stuttering refined run and a corresponding abstract run.

Example 2 (Refinement of the case study). We modeled the LGS by means of four refinement steps. The model LGS_GM in Code 1 is the ground ASM. In the refined model LGS_SE, we have added the modeling of the sensors that

```
asm LGS_SE

signature:
  ...
  dynamic monitored doorsOpen: Boolean
  dynamic monitored doorsClosed: Boolean
  dynamic monitored gearsExtended: Boolean
  dynamic monitored gearsRetracted: Boolean
definitions:
  rule r_closeDoor =
    switch doors
      case CLOSING:
      if doorsClosed then
        par
          generalEV := false
          closeDoorsEV := false
          doors := CLOSED
        endpar
      endif
  ...
```

```
rule r_retractionSequence =
  if gears != RETRACTED then
    switch doors
      case CLOSED:
        par
          generalEV := true
          openDoorsEV := true
          doors := OPENING
        endpar
      case OPENING:
        if doorsOpen then
          par
            openDoorsEV := false
            doors := OPEN
          endpar
        endif
      ...

invariant over doorsClosed, doorsOpen: not(doorsClosed and doorsOpen)
invariant over gearsExtended, gearsRetracted: not(gearsExtended and gearsRetracted)
```

Code 2. Landing Gear System – Refined model

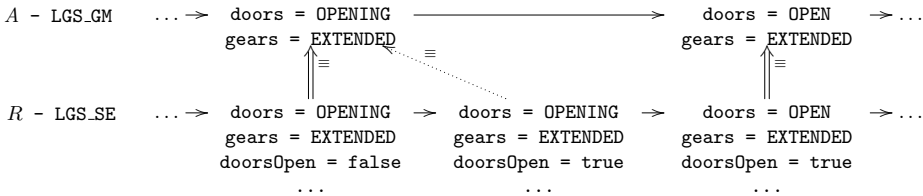

Fig. 2. LGS case study – Relation between a refined run and an abstract run

detect when the doors and the gears change their status. Code 2 shows the new elements introduced in the model. Four boolean monitored functions are used to indicate whether the gears are extended (`gearsExtended`) or retracted (`gearsRetracted`), and whether the doors are closed (`doorsClosed`) or open (`doorsOpen`). In this model, we have refined some rules by adding the reading of sensors. Some update rules have been guarded by conditional rules checking the value of the monitored functions; for example, we can see in Code 2 that if the `doors` are `CLOSING`, they become `CLOSED` only if the sensor `doorsClosed` is turned on (i.e., the guard of conditional rule is true). Note that LGS_SE is a stuttering refinement of LGS_GM because when the sensors do not detect any change, the state is still conformant to the previous abstract state (i.e., the doors and the gears have not changed their statuses). See Fig. 2 as an example of corresponding runs.

4 Proving refinement

We here aim at automating the proof of stuttering refinement between two ASMs by reducing it to a satisfability checking problem (to be solved by, e.g., an SMT solver). Therefore, we need to describe the concepts of ASM state and ASM transition by means of suitable predicates, and Def. 3 as a first order formula. The validity of such a formula should guarantee the stuttering machine refinement.

We introduce the following predicates capturing the concepts of initial state, next state, and conformant states.

– $init(S)$ iff S is an initial state;
– $step(S, S')$ iff the state S' can be obtained by applying the main rule at S;
– $conf(\widetilde{S}, S)$ iff states S and $\widetilde{S}$ are conformant (see Def. 2).

In the following, let us indicate by S a state of the abstract machine A and by S' the next state of S. Similarly, $\widetilde{S}$ and $\widetilde{S}'$ are two subsequent states of the refined machine R.

Theorem 1. *If the following properties hold*

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

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

then R is a stuttering refinement of A.

Proof. Def. 3 follows from properties (1) and (2) by induction on the length of a run of the refined machine R.

Let $\widetilde{\rho} = \widetilde{S}_0, \widetilde{S}_1, \ldots$, be a run of R. $\widetilde{\rho}$ can be splitted in subruns $\widetilde{\rho}_0, \widetilde{\rho}_1, \ldots$ such that all the states in each $\widetilde{\rho}_i$ have the same values of the linking variables, whereas states of two consecutive subruns $\widetilde{\rho}_i$ and $\widetilde{\rho}_{i+1}$ have different values for the linking variables.

We build a run $\rho = S_0, S_1, \ldots$ of A such that ρ and $\widetilde{\rho}$ satisfy Def. 3.

By property (1), it holds $conf(\widetilde{S}_0, S_0)$ by taking $\widetilde{S}_0$ as $\widetilde{S}$ and S_0 as one existing state S satisfying the implication in (1).

Let us suppose that Def. 3 holds till state $\widetilde{S}_j$ of $\widetilde{\rho}_k$ and that $\widetilde{S}_j$ conforms to the abstract state S_k of ρ. We now consider the next state $\widetilde{S}_{j+1}$ (i.e., $step(\widetilde{S}_j, \widetilde{S}_{j+1})$). By inductive hypothesis, it holds $conf(\widetilde{S}_j, S_k)$. By property (2), considering $\widetilde{S} = \widetilde{S}_j$, $\widetilde{S}' = \widetilde{S}_{j+1}$, $S = S_k$, one of the two conditions must hold: $\exists S' \colon (step(S_k, S') \wedge conf(\widetilde{S}_{j+1}, S'))$ or $conf(\widetilde{S}_{j+1}, S_k)$. In the first case, we take $S_{k+1} = S'$ and we start considering a new subrun $\widetilde{\rho}_{k+1}$ of refined states conformant to S_{k+1}, while in the second case $\widetilde{S}_{j+1}$ is still part of the subrun $\widetilde{\rho}_k$ whose states conform to S_k. In both cases the A-run satisfies the property of Def. 3.

In the sequel, we refer to property (1) as *initial refinement*, and to property (2) as *step refinement*.

Example 3 (Proof of LGS stuttering refinement). The model LGS_SE is a correct stuttering refinement of LGS_GM. The two models have the same initial state and then property (1) of Thm. 1 is guaranteed. Moreover, model LGS_SE step refines model LGS_GM (i.e., property (2) of Thm. 1 is guaranteed). Indeed, in each state, the refined machine can move to a state in which the doors status or the gears status are either changed (if the sensors detect the changing) or unchanged (if the sensors do not detect any change). Therefore, if a state $\widetilde{S}$ of the refined model LGS_SE is conformant with a state S of the abstract model LGS_GM, a step in the refined model can lead to a state $\widetilde{S}'$ that is either conformant with S (if the sensors do not detect any change) or with the next state S' of the abstract model (if the sensors detect the changing).

```
asm M0
signature:
    controlled x: Integer
definitions:
    main rule r_Main = x := x + 1

default init s0:
    function x = 0
```

Code 3. Abstract model

```
asm M1
signature:
  controlled x: Integer
definitions:
  main rule r_Main =
    if x > 0 then x := x + 1
    else x := 1
    endif

default init s0: function x = 0
```

Code 4. Refined model of Code 3

4.1 Using invariants in refinement proof

The step refinement property in Thm. 1 is a sufficient but not a necessary condition for a correct (stuttering) refinement. A machine R could be a correct refinement of a machine A but it may not guarantee step refinement. Indeed, step refinement is also checked over states that are not reachable in R: if step refinement is violated only in unreachable states, then we falsely judge the refinement not correct.

Example 4. Let us consider the ASM model *M0* in Code 3 that simply increments function x, and the model *M1* in Code 4 that increments x if it is greater than 0, otherwise updates it to 1. Model *M1* does not step refine model *M0* (property (2) of Thm. 1), because, when x is negative, x is incremented in *M0* and updated to 1 in *M2*: therefore, by Thm. 1 we could not state that model *M1* is a correct stuttering refinement of model *M0*. However, *M1* is a correct stuttering refinement of *M0*; indeed, states in which x is negative are not reachable in both models.

Thm. 1 can be modified by strengthening the inductive hypothesis by introducing a state invariant I over the refined machine as follows:

Theorem 2. *If there exists an invariant I such that the following properties hold*

$$\forall \widetilde{S}\colon (init(\widetilde{S}) \rightarrow (I(\widetilde{S}) \wedge \exists S\colon (init(S) \wedge conf(\widetilde{S}, S)))) \tag{3}$$

$$\forall \widetilde{S} \forall \widetilde{S}' \forall S : \left[\begin{pmatrix} I(\widetilde{S}) \wedge \\ step(\widetilde{S}, \widetilde{S}') \wedge \\ conf(\widetilde{S}, S) \end{pmatrix} \rightarrow I(\widetilde{S}') \wedge \begin{pmatrix} \exists S' : (step(S, S') \wedge conf(\widetilde{S}', S')) \\ \vee \\ conf(\widetilde{S}', S) \end{pmatrix} \right] \tag{4}$$

then R is a stuttering refinement of A.

Proof. The invariant used in the formulas simply restricts the set of states of the refined machine over which we need to verify refinement correctness. For this reason, the proof of Thm. 1 is applicable also in this case.

Example 5. The refinement between model *M0* in Code 3 and *M1* in Code 4 is correctly proved correct using the invariant $I = x \geq 0$ in Formulas 3 and 4.

Although finding a suitable invariant I may be difficult, thanks to Thm. 2 designers can prove arbitrary complex stuttering refinements.

4.2 Towards an SMT encoding

To be useful for our final goal, namely reducing the proof of initial and step refinement to an SMT problem, in Formulas (1) and (2) we need to symbolically represent the states and the transition relation.

Functions of any arity are supported by our technique (and by the tool implementation), provided that all the function domains are finite; note that infinite domains may introduce quantifications that are not evaluated (i.e., `unknown` result) by the SMT solver. In the following, in order not to over complicate the notation of our formulas, we only consider 0-ary functions.

Let $\bar{f}_A = [fa_1, \ldots, fa_n]$ be the ordered list of the functions of the abstract machine A and $\bar{f}'_A = [fa'_1, \ldots, fa'_n]$ a renamed copy of the functions in the next state. Similarly, we define ordered lists $\bar{f}_R = [fr_1, \ldots, fr_m]$ and $\bar{f}'_R = [fr'_1, \ldots, fr'_m]$ for the refined machine R. We order the functions of all the previous lists such that the first L functions are the locations of interest. When necessary, we split a list of functions $\bar{f}$ between the functions corresponding to locations of interest (those for which we are interested in checking the conformance relation) and those which are not related: $\bar{f} = \bar{f}^c + \bar{f}^{nc}$.

We can express the predicates *init*, *step*, *conf*, and I used in Thm. 2, in terms of the function lists of a machine.

- Given a machine M with functions $\bar{f}_M$, we introduce the predicates $init_M(\bar{f}_M)$ and $step_M(\bar{f}_M, \bar{f}'_M)$ formalizing the initial predicate and the step predicate of the machine.
- We can define the conformance relation between states of two related machines by using a relation between the lists of machine functions. Given two ordered lists $\bar{p} = [p_1, \ldots, p_L, \ldots]$ and $\bar{q} = [q_1, \ldots, q_L, \ldots]$, both long at least L, we introduce

$$conf(\bar{p}, \bar{q}) \quad \equiv \quad \bigwedge_{i=1}^{L} p_i = q_i \quad \equiv \quad \bar{p}^c = \bar{q}^c \tag{5}$$

  to represent conformance: if $conf(\bar{p}, \bar{q})$ is true, all the locations of interest have equal values in $\bar{p}$ and $\bar{q}$.
- The invariant I, if necessary, is provided by the user as a predicate over the functions $\bar{f}_R$.

In order to prove initial refinement (property (3) of Thm. 2), we check whether the following formula is valid:

$$\forall \bar{f}_R \colon (init_R(\bar{f}_R) \rightarrow (I(\bar{f}_R) \wedge \exists \bar{f}_A \colon (init_A(\bar{f}_A) \wedge conf(\bar{f}_R, \bar{f}_A)))) \tag{6}$$

In order to prove step refinement (property (4) of Thm. 2), we check whether the following formula is valid:

$$\forall \bar{f}_R \, \forall \bar{f}'_R \, \forall \bar{f}_A \colon \left[\begin{pmatrix} I(\bar{f}_R) \wedge \\ step_R(\bar{f}_R, \bar{f}'_R) \wedge \\ conf(\bar{f}_R, \bar{f}_A) \end{pmatrix} \rightarrow I(\bar{f}'_R) \wedge \begin{pmatrix} \exists \bar{f}'_A : (step_A(\bar{f}_A, \bar{f}'_A) \wedge conf(\bar{f}'_R, \bar{f}'_A)) \\ \vee \\ conf(\bar{f}'_R, \bar{f}_A) \end{pmatrix} \right] \tag{7}$$

We can transform Formulas 6 and 7 in order to eliminate universal quantifiers (by Herbrandization) and reduce the number of variables (by exploiting the equality of variable values induced by the conformance), as follows:

$$init_R(\bar{f}_R) \rightarrow (I(\bar{f}_R) \wedge \exists \bar{f}_A^{nc} \colon init_A(\bar{f}_R^c + \bar{f}_A^{nc})) \tag{8}$$

$$\begin{pmatrix} I(\bar{f}_R) \wedge \\ step_R(\bar{f}_R, \bar{f}'_R) \end{pmatrix} \rightarrow I(\bar{f}'_R) \wedge \begin{pmatrix} \exists \bar{f}_A^{nc\prime} \colon step_A(\bar{f}_R^c + \bar{f}_A^{nc}, \bar{f}_R^{c\prime} + \bar{f}_A^{nc\prime}) \\ \vee \\ \bar{f}_R^{c\prime} = \bar{f}_R^c \end{pmatrix} \tag{9}$$

Formulas 8 and 9 no longer contain the variable lists $\bar{f}_A^c$ and $\bar{f}_A^{c\prime}$; so we can avoid the duplication for A of all the locations of interest in the current and next state.

5 Proving refinement by SMT

In this section, we show how we can prove stuttering refinement in an automatic way by reducing it to a Satisfiability Modulo Theories (SMT) problem.

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.

5.1 SMT-based refinement proof

We need to represent the initial states and a generic step of the ASM machine in an SMT solver and prove initial and step refinement (i.e., Thm. 2 encoded as Formulas 8 and 9). In order to do this, we here extend the mapping from ASM to SMT already presented in [5] for different purposes.

Given a machine $M = \langle sig, funcDefs, funcInit, r\_main \rangle$, being sig the signature containing the functions $\bar{f}$, $funcInit = \{fi_1, \ldots, fi_p\}$ the sequence of function initializations and $funcDefs = \{fd_1, \ldots, fd_q\}$ the sequence of function definitions, we define the predicates $init_M$ and $step_M$, formalizing the initial state and the generic step of the machine (see Sect. 4), as follows:

$\text{init}_M = (\text{and } \mathtt{T_d}(\text{fi}_1) \ldots \mathtt{T_d}(\text{fi}_p))$ $\qquad$ $\text{step}_M = (\text{and } \mathtt{T_r}(\text{r\_main}) \ \mathtt{T_d}(\text{fd}_1) \ldots \mathtt{T_d}(\text{fd}_q))$

where $\mathtt{T_d}$ and $\mathtt{T_r}$ are functions that map, respectively, ASM function definitions and transition rules to SMT formulas. Note that $\mathtt{T_r}(r\_main)$ fully captures the semantics of ASM transition rules: it specifies that a location must be updated under some given conditions, and must be kept unchanged otherwise. ASMs semantics prescribes that non-updated locations are kept unchanged; in SMT this must be specified explicitly. We refer to [5] for details on the mapping.

We now show how we verify the validity of Formulas 8 and 9 using two SMT instances. Let $\{Da_1, \ldots, Da_n\}$ be the codomains of the functions $\bar{f}_A$ of the abstract machine A and $\{Dr_1, \ldots, Dr_m\}$ those of the functions $\bar{f}_R$ of the

refined machine R. We identify with *inv* the mapping of the proof invariant, i.e., $inv = \mathtt{T_t}(I)$, being $\mathtt{T_t}$ the map function from ASM terms to SMT.

For Formula 8, we build the following SMT instance:

(**declare**−**fun** fr_1 () Dr_1) ... (**declare**−**fun** fr_m () Dr_m)
(**define**−**fun** init_R () Bool $\mathrm{init}_R(\bar{f}_R)$)
(**define**−**fun** inv_R () Bool $\mathrm{inv}(\bar{f}_R)$)
(**define**−**fun** $\mathrm{existsInit}_A$ () Bool
(**exists** ((fa_{L+1} Da_{L+1}) ... (fa_n Da_n)) $\mathrm{init}_A(\bar{f}_R^c + \bar{f}_A^{nc})$))
(**assert** (not (=> init_R (and inv_R $\mathrm{existsInit}_A$))))

where the antecedent of the implication is represented through the SMT function init_R, and the consequent by the conjunction of functions inv_R and $\mathsf{existsInit}_A$.

For Formula 9, we build the following instance[5]:

(**declare**−**fun** fr_1 () Dr_1) ... (**declare**−**fun** fr_m () Dr_m)
(**declare**−**fun** fr'_1 () Dr_1) ... (**declare**−**fun** fr'_m () Dr_m)
(**declare**−**fun** fa_{L+1} () Da_{L+1}) ... (**declare**−**fun** fa_n () Da_n)
(**define**−**fun** step_R () Bool $\mathrm{step}_R(\bar{f}_R, \bar{f}'_R)$)
(**define**−**fun** inv_R () Bool $\mathrm{inv}(\bar{f}_R)$)
(**define**−**fun** inv'_R () Bool $\mathrm{inv}(\bar{f}'_R)$)
(**define**−**fun** $\mathrm{existsStep}_A$ () Bool
(**exists** ((fa'_{L+1} Da_{L+1}) ... (fa'_n Da_n)) $\mathrm{step}_A(\bar{f}_R^c + \bar{f}_A^{nc}, \bar{f}_R^{c\prime} + \bar{f}_A^{nc\prime})$))
(**define**−**fun** stutteringState () Bool (and (= fr'_1 fr_1) ... (= fr'_L fr_L)))
(**assert** (not (=> (and inv_R step_R) (and inv'_R (or $\mathrm{existsStep}_A$ stutteringState)))))

where the conjunction of the antecedent of the implication is represented by functions inv_R and step_R. The consequent is represented by functions inv'_R, $\mathsf{existsStep}_A$ and $\mathsf{stutteringState}$; 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 9) as a conjunction of equalities.

As usual in SMT solvers, in order to prove validity of a formula, we check that its negation is unsatisfiable. Therefore, if both previous two instances are proved to be unsatisfiable, the refinement is proved correct. However, since the step refinement condition is sufficient but not necessary, when Formula 9 is proved not valid (i.e., the corresponding SMT instance is satisfiable), we cannot state that the refinement is not correct.

Note that, when the refinement is not proved correct, the SMT solver provides us a model (over functions $\bar{f}_R$, $\bar{f}'_R$, and $\bar{f}_a^{nc}$) that acts as a *witness* of the refinement incorrectness: by examining the witness, we can understand whether it is really the case that the refinement is not correct, or it is a false negative result and so we have to strengthen the invariant. For example, proving refinement between Codes 3 and 4 (without any invariant) returns as witness `(= x0 -1) (= x1 1)`. The witness tells us that step refinement does not hold from the state in which `x` is -1; however, since `x` cannot be negative, the result is a false negative and we can strengthen the proof by adding the invariant $x \geq 0$.

[5] Note that in concrete instances we also do not declare constants for monitored and derived functions belonging to $\bar{f}_R^{nc\prime}$ and $\bar{f}_A^{nc\prime}$, as they do not appear in the asserted formulas.

```
(define−fun doors0 () DoorStatus) (define−fun gears0 () GearStatus)
(define−fun doorsOpen0 () Bool) (define−fun doorsClosed0 () Bool) ...
(define−fun generalElectroValve0 () Bool) ...
(define−fun init_LGS_SE () Bool (and (= doors0 CLOSED) (= gears0 EXTENDED)
                                     (not generalElectroValve0) (not extendGearsElectroValve0) ...))
(define−fun existsInit_LGS_GM () Bool (and (= doors0 CLOSED) (= gears0 EXTENDED)))
(assert (not (=> init_LGS_SE existsInit_LGS_GM)))
(check−sat)
```

Code 5. LGS case study – Initial refinement proof (from Code 1 to Code 2)

```
(define−fun doors0 () DoorStatus) (define−fun gears0 () GearStatus)
(define−fun doorsOpen0 () Bool) (define−fun doorsClosed0 () Bool) ...
(define−fun generalElectroValve0 () Bool) ...
(define−fun doors1 () DoorStatus) (define−fun gears1 () GearStatus)
(define−fun doorsOpen1 () Bool) (define−fun doorsClosed1 () Bool) ...
(define−fun generalElectroValve1 () Bool) ...
(define step_LGS_SE () Bool (and (if (= handle0 UP) ...)))
(define existsStep_LGS_GM () Bool (exists (handle HandleStatus)
                                          (and (if (= handle UP) (if (/= gears0 RETRACTED) ...)))
(define−fun stutteringState () Bool (and (= gears0 gears1) (= doors0 doors1)))
(assert (not (=> step_LGS_SE (or existsStep_LGS_GM stutteringState))))
(check−sat)
```

Code 6. LGS case study – Step refinement proof (from Code 1 to Code 2)

Example 6. Codes 5 and 6 show the SMT instances built for proving initial and step refinement between the ASMs shown in Codes 1 and 2 for the LGS. In this case, there is no need to specify any invariant.

6 Evaluation

Based on the translation presented in previous sections, we have developed a tool[6] that, given two ASMs, builds the SMT instances and calls the SMT solver Yices in order to prove refinement correctness. The refinement prover is *integrated* in the ASMETA toolset.

The effectiveness of our approach has been tested on different case studies. Some are taken from the literature [13] and are examples of ASM model refinement whose correctness was manually proved. Others are specification case studies developed by ourselves in different contexts: Cloud-based applications [8], a Landing Gear System [6], and the validation of medical software [3,4]. In almost all the cases, the refinement has been proved in less than 10 secs on a Linux machine, Intel(R) Core(TM) i7, 4 GB RAM. However, for one refinement step in [4], we were not able to complete the proof in less than 5 min, the fixed timeout after which we stop the proof. The limiting factor for scalability is the number of monitored functions that are existentially quantified in Formula 9; the refined model whose refinement correctness we were not able to prove has

[6] The tool and experimental results can be found at `http://asmeta.sourceforge.net/download/asmrefprover.html`

32 boolean monitored functions. As future work, we plan to assess the approach scalability and apply techniques to reduce the time and memory consumption of the tool (e.g., using cone of influence reduction techniques).

7 Related work

Formal methods whose computational model is a transition system, e.g., B [2], Z [15], I/O automata [19,18], support the concept of model refinement. The ASM refinement can be compared to that of all the other formalisms and this has already been extensively done in [12]. For this reason, we here relate our work only with Börger's original notion of ASM refinement and its definition of correct model refinement.

Börger's refinement definition [12] is based on checking correspondence between run segments of abstract and refined machines, in a way that the starting and ending states (those of interest) of such corresponding subruns are conformant. The definition allows (m, n)-refinements, namely a run segment of length n in the refined machine simulates as a run segment of length m in the abstract machine. Moreover, it permits that some abstract/refined states don't have corresponding refined/abstract states. We keep the concepts of locations of interests and state conformance given in terms of data equivalence relation between locations of interest. Stuttering refinement is a particular case of Börger's definition, i.e., $(1, n)$-refinement with the constraint of total conformance relation on the states of the refined machine. In our opinion, the restriction of Börger's schema of refinement we propose here is not particularly disadvantageous. Firstly, $(1, n)$-refinement is a kind of refinement that is already considered in literature (with the name of *action refinement* [10]). Secondly, this restricted schema applies to all the ASM specifications we have considered to evaluate the effectiveness of our approach (see Sect. 6). Furthermore, when modeling, it is often useful to guarantee that invariants holding in the abstract level, still hold in the refined one (this was the case for the Landing Gear System specification [6]). The classical refinement [12] preserves the invariants only weakly, since intermediate refined states are not required to conform to some abstract state, while stuttering refinement preserves all the invariants (as also the approach in [23]): if a property is true in every abstract state, it will be true also in every refined state (modulo the *conf* relation). The need to guarantee preservation of those state invariants inspired our definition.

Another framework supporting ASM refinement is that proposed by Schellhorn [22], which is based on the use of the KIV theorem prover. With respect to that framework, ours has several differences. Our *definition of conformity* is much simpler than that used by the KIV tool, because we simply assume that a refined state conforms to its abstract one if they have equal values of functions having the same name (which are the functions of interest). If the user wants to define an ad hoc conformance relation, (s)he must add a derived function representing a predicate over the abstract or refined states (see Sect. 3). In order to *prove refinement*, a relation between the runs must be proved in [22],

while we require to prove only a relation between the initial states and between two consecutive states. Using runs permits completeness but requires the use of temporal logics, while our proof is much simpler. Our approach is analogous to induction-based bounded model checking, in which the *next* relation suffices in proving the validity of temporal invariants. KIV supports interactive verification, while we aim to a *completely automatic* technique. Also for this reason, we have chosen an SMT solver. In case the proof fails, we are able to show a counterexample in which the refinement is not preserved. As shown before, there are some cases in which our technique produces spurious counterexamples and it is unable to prove the refinement. These spurious counterexamples can be eliminated by invariant strengthening.

8 Conclusions

We have presented an approach for proving the refinement correctness of Abstract State Machines. The approach considers a particular type of refinement (i.e., stuttering refinement) that frequently occurs in concrete case studies. The proposed approach exploits the symbolic representation of an ASM model in an SMT solver, and reduces the proof of refinement correctness to a satisfiability problem that is automatically solved by the SMT solver. The technique has been implemented in a tool integrated in the ASMETA framework. Although the limits in terms of completeness (some refinements could be very hard to prove) and expressive power (some refinements may be not stuttering), the tool has the advantages of usability, integration in an existing framework, and automation in proving refinement correctness. This relieves the modeler of the necessity to drive a mathematical proof manually or in an interactive way (as requested in [22,23]), which requires certain verification skills. Furthermore, in case a model is not proved a correct stuttering refinement of another model, our framework provides counterexamples useful to reason about incorrect modeling of the refined machine.

In this work, we have considered deterministic and nondeterministic single-agent ASMs. As future work, we plan to prove refinement of multi-agent ASMs. Moreover, we want to study techniques for automatic invariant generation.

References

1. M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, May 1991.
2. J.-R. Abrial and S. Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. *Fundam. Inform.*, 77(1):1–28, 2007.
3. P. Arcaini, S. Bonfanti, A. Gargantini, A. Mashkoor, and E. Riccobene. Formal validation and verification of a medical software critical component. In *Proceedings of MEMOCODE 2015*, pages 80–89. IEEE, Sept 2015.
4. P. Arcaini, S. Bonfanti, A. Gargantini, and E. Riccobene. How to assure correctness and safety of medical software: the hemodialysis machine case study. In *Abstract*

State Machines, Alloy, B, TLA, VDM, and Z. 5th International Conference, ABZ 2016, Linz, Austria, May 23-27, 2016, Proceedings, volume 9675 of *Lecture Notes in Computer Science*. Springer International Publishing, 2016.
5. P. Arcaini, A. Gargantini, and E. Riccobene. Using SMT for dealing with nondeterminism in ASM-based runtime verification. *ECEASST*, 70, 2014.
6. P. Arcaini, A. Gargantini, and E. Riccobene. Rigorous development process of a safety-critical system: from ASM models to Java code. *International Journal on Software Tools for Technology Transfer*, pages 1–23, 2015.
7. P. Arcaini, A. Gargantini, E. Riccobene, and P. Scandurra. A model-driven process for engineering a toolset for a formal method. *Software: Practice and Experience*, 41:155–166, 2011.
8. P. Arcaini, R.-M. Holom, and E. Riccobene. ASM-based formal design of an adaptivity component for a cloud system. *Formal Aspects of Computing*, pages 1–29, 2016.
9. C. Beierle, E. Börger, I. Durdanović, U. Glässer, and E. Riccobene. Refining abstract machine specifications of the steam boiler control to well documented executable code. In *Formal Methods for Industrial Applications*, pages 52–78. Springer, 1996.
10. E. A. Boiten. Introducing extra operations in refinement. *Formal Aspects of Computing*, 26(2):305–317, 2012.
11. F. Boniol and V. Wiels. The Landing Gear System Case Study. In *ABZ 2014: The Landing Gear Case Study*, volume 433 of *Communications in Computer and Information Science*, pages 1–18. Springer International Publishing, 2014.
12. E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 15(2):237–257, 2003.
13. E. Börger. The Abstract State Machines method for high-level system design and analysis. In *Formal Methods: State of the Art and New Directions*, pages 79–116. Springer London, 2010.
14. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.
15. J. Derrick and E. Boiten. *Refinement in Z and object-Z: Foundations and Advanced Applications*. Springer-Verlag, London, UK, UK, 2001.
16. G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif. Modular refinement for submachines of ASMs. In *ABZ 2014*, volume 8477 of *Lecture Notes in Computer Science*, pages 188–203. Springer Berlin Heidelberg, 2014.
17. R. Farahbod and U. Glässer. The CoreASM modeling framework. *Software: Practice and Experience*, 41(2):167–178, 2011.
18. N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.
19. N. A. Lynch and F. W. Vaandrager. Forward and backward simulations: I. untimed systems. *Information and Computation*, 121(2):214–233, 1995.
20. J. Meseguer, M. Palomino, and N. Martí-Oliet. Algebraic simulations. *The Journal of Logic and Algebraic Programming*, 79(2):103–143, 2010.
21. E. Riccobene and J. Schmid. Capturing requirements by abstract state machines: The light control case study. *J. UCS*, 6(7):597–620, 2000.
22. G. Schellhorn. Verification of ASM refinements using generalized forward simulation. *J. UCS*, 7(11):952–979, 2001.
23. G. Schellhorn. ASM refinement preserving invariants. *J. UCS*, 14(12):1929–1948, 2008.
24. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine*, volume 24. Springer-Verlag, 2001.