Addressing Usability in a Formal Development Environment

Paolo Arcaini[1(✉)], Silvia Bonfanti[2] and Angelo Gargantini[2], Elvinia Riccobene[3], and Patrizia Scandurra[2]

[1] National Institute of Informatics, Tokyo, Japan
arcaini@nii.ac.jp
[2] University of Bergamo, Bergamo, Italy
{silvia.bonfanti,angelo.gargantini,patrizia.scandurra}@unibg.it
[3] Università degli Studi di Milano, Milan, Italy
elvinia.riccobene@unimi.it

Abstract. Even though the formal method community tends to overlook the problem, formal methods are sometimes difficult to use and not accessible to average users. On one hand, this is due to the intrinsic complexity of the methods and, therefore, some level of required expertise is unavoidable. On the other hand, however, the methods are sometimes hard to use because of lack of a user-friendly tool support. In this paper, we present our experience in addressing usability when developing a framework for the Abstract State Machines (ASMs) formal method. In particular, we discuss how we enhanced modeling, validation, and verification activities of an ASM-based development process. We also provide a critical review of which of our efforts have been more successful as well as those that have not obtained the results we were expecting. Finally, we outline other directions that we believe could further lower the adoption barrier of the method.

Keywords: Abstract State Machines · ASMETA · Usability · Formal methods

1 Introduction

One of the seven myths that Hall listed in his well-known paper [27] is that "formal methods are unacceptable to users". Bowen and Hinchey discussed seven more myths [19] and, among these, they reported the lack of tool support as another myth. However, as formal method community, we have to admit that there is a part of truth in each myth: formal methods can be sometimes difficult to use and not accessible to average users. On one hand, this is due to the intrinsic complexity of the methods and, therefore, some level of required expertise is unavoidable.

P. Arcaini is supported by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST. Funding Reference number: 10.13039/501100009024 ERATO.

E. Sekerinski et al. (Eds.): FM 2019 Workshops, LNCS 12232, pp. 61–76, 2020.
https://doi.org/10.1007/978-3-030-54994-7_6

On the other hand, however, the methods are hard to use because of lack of a user-friendly support. Hall himself, while dispelling the method, recognized that designers should "make the specification comprehensible to the user" [27], and Bowen and Hinchey recognized that more effort must be spent on tool support [19].

The Abstract State Machines (ASMs) formal method [18] is a state-based formal method that is usually claimed to be usable, since a practitioner can understand ASMs as pseudo-code or virtual machines working over abstract data structures. However, from our long time experience in using the method and in teaching it, we realized that there are some aspects of the method that can prevent from using it in the most fruitful way.

In 2006, we started developing the ASMETA framework, with the aim of building a set of tools around the ASM method. While developing validation and verification techniques for the method, we kept *usability* as one of our leading principles. This was also motivated by the fact that, in addition to us, the primary users of the framework are our students to which we teach ASMs. As most of them are not naturally attracted by formal methods, we wanted to build a framework that could assist them in using the ASM method and would lower the adoption barriers of the method. In particular, we declined usability in three more concrete driving principles:

- *smoothness*: the framework should be usable with as less effort as possible. The user should not care about technical details that can be hidden and automatized;
- *understandability*: the framework should help in understanding the model itself and the results of its validation and verification;
- *interoperability*: the different tools of the framework should be integrated as much as possible, such that the user can inspect the results of one tool with another tool without any effort. As an example, the counterexamples of a model checker should be re-executable by the simulator.

In this paper, we describe how the different tools/techniques of ASMETA try to fulfil these principles.

The paper is structured as follows. Section 2 briefly introduces the ASM method, and Sect. 3 gives a general overview of the ASMETA framework. Section 4 describes how we addressed usability at the modeling, validation, and verification levels. Then, Sect. 5 critically reviews our efforts and outlines other directions that could further increase the usability of the framework. Finally, Sect. 6 reviews some related work, and Sect. 7 concludes the paper.

2 Abstract State Machines

Abstract State Machines (ASMs) [18] are an extension of FSMs, where unstructured control states are replaced by states with arbitrary 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 ASM concept of basic object

```
asm HemodialysisGround

signature:
  enum domain Phases = {PREPARATION | INITIATION | ENDING}
  controlled phase: Phases

definitions:
  macro rule r_run_preparation =
    phase := INITIATION

  macro rule r_run_initiation =
    phase := ENDING

  macro rule r_run_ending =
    skip
```

```
  macro rule r_run_dialysis =
    par
      if phase = PREPARATION then
        r_run_preparation[]
      endif
      if phase = INITIATION then
        r_run_initiation[]
      endif
      if phase = ENDING then
        r_run_ending[]
      endif
    endpar

  main rule r_Main = r_run_dialysis[]

default init s0: function phase = PREPARATION
```

Fig. 1. Example of ASM model

container. The couple (*location*, *value*) represents a 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`, `switch-case`), simultaneous parallel actions (`par`), sequential actions (`seq`), nondeterminism (existential quantification `choose`), and unrestricted synchronous parallelism (universal quantification `forall`).

An ASM *computation* (or *run*) 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 also 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 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). Figure 1 shows a simple example of an ASM model (the ground model of the haemodialysis case study [4]).

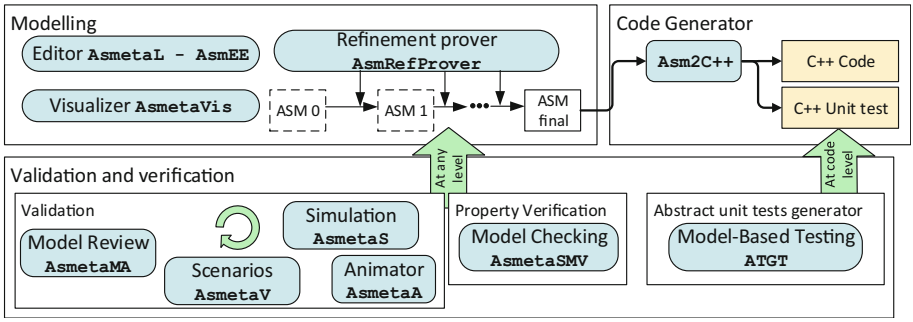


Fig. 2. The ASM development process powered by the ASMETA framework

3 ASMETA

The ASM method is applied along the entire life cycle of software development, i.e., from modeling to code generation. Figure 2 shows the development process based on ASMs.

The process is supported by the ASMETA (ASM mETAmodeling) framework[1] [11] which provides a set of tools to help the developer in various activities:

- **Modeling**: the system is modeled using the language `AsmetaL`. The user is supported by the editor `AsmEE` and by `AsmetaVis`, the ASMs visualizer which transforms the textual model into a graphical representation. The refinement process can be adopted in case the model is complex: the designer can start from the first model (also called the ground model) and can refine it through the refinement steps by adding details to the behavior of the ASM. The `AsmRefProver` tool checks whether the current ASM model is a correct refinement of the previous ASM model.
- **Validation**: the process is supported by the model simulator `AsmetaS`, the animator `AsmetaA`, the scenarios executor `AsmetaV`, and the model reviewer `AsmetaMA`. The simulator `AsmetaS` allows to perform two types of simulation: interactive simulation and random simulation. The difference between the two types of simulation is the way in which the monitored functions are chosen. During interactive simulation the user provides the value of functions, while in random simulation the tool randomly chooses the value of functions among those available. `AsmetaA` allows the same operation of `AsmetaS`, but the states are shown using tables. `AsmetaV` executes scenarios written using the `Avalla` language. Each scenario contains the expected system behavior and the tool checks whether the machine runs correctly. The model reviewer `AsmetaMA` performs static analysis in order to check model quality attributes like minimality, completeness, and consistency.

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

- **Verification**: properties are verified to check whether the behavior of the model complies with the intended behavior. The `AsmetaSMV` tool supports this process in terms of model checking.
- **Testing**: the tool `ATGT` generates abstract unit tests starting from the ASM specification by exploiting the counterexamples generation of a model checker.
- **Code generation**: given the final ASM specification, the `Asm2C++` automatically translates it into C++ code. Moreover, the abstract tests, generated by the `ATGT` tool, are translated to C++ unit tests.

The framework has been applied to the formal analysis of different kinds of systems: a landing gear system [9], a haemodialysis device [4], self-adaptive systems [14], cloud systems [12], etc.

4 How We Have Addressed Usability in ASMETA

In this section, we describe how we have targeted usability when developing the ASMETA framework. First of all, in order to obtain an integrated framework in which the different tools can be used together, we developed all the tools as eclipse plugins[2].

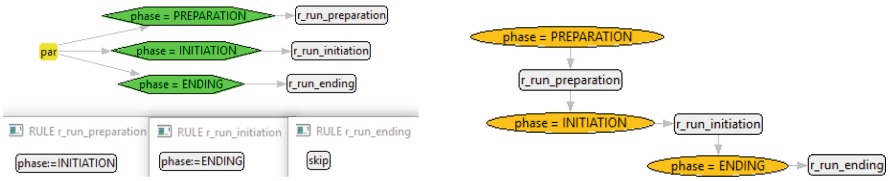
In the following, we overview the techniques of the framework that have improved it according to the three driving principles (i.e., *smoothness*, *understandability*, and *interoperability*), rather than purely improvements in terms of functionality of the framework. In the following sections, we focus on the three main phases of a formal development process: modeling, validation, and verification.

4.1 Modeling

The first step of the development process is model definition. On the top of the original parser and editor [26], we introduced a technique that provides a better visualization of the model (so improving the *understandability*), and another technique that automatically checks for common errors (so improving the *smoothness* of use).

Visualization. When a model is particularly complex, exploring it can become difficult, and so the developer does not have a proper understanding of the whole structure. In order to improve the exploration of the structure of an ASM model, in [5], we introduced the graphical visualizer `AsmetaVis`. The *basic visualization* permits to show the syntactical structure of the ASM in terms of a tree (similar to an AST); the notation is inspired by the classical flowchart notation, using green rhombuses for guards and grey rectangles for rules. The leaves of the tree are the update rules and the macro call rules. For each macro rule in the model,

[2] The update site is http://svn.code.sf.net/p/asmeta/code/code/stable/asmeta_update/.



(a) Basic visualization (b) Semantic visualization

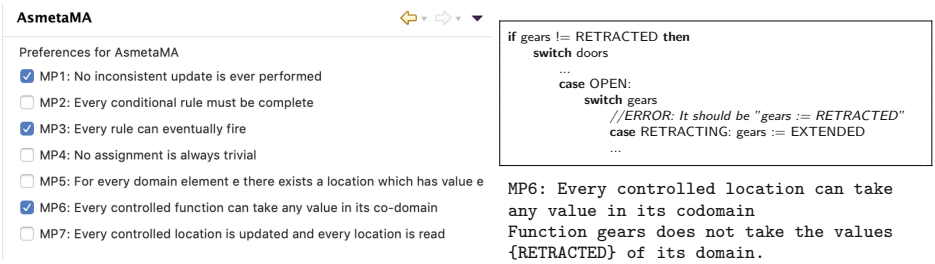
Fig. 3. Visualizer `AsmetaA` – visualization of the ASM model shown in Fig. 1

there is a tree representing the definition of the rule; double-clicking on a macro call rule shows the tree of the corresponding macro rule. Figure 3a shows the basic visualization with `AsmetaVis` (starting from rule `r_run_dialysis`) of the ASM model shown in Fig. 1.

In this case, all the macro rules are shown (i.e., the user has selected all the call rules). Note that the visualization is particularly useful when the model is big, as the user can decide which rules to visualize.

Control states ASMs [18] are a particular class of ASMs in which there is a function (called *phase function*) that identifies the current *control state*; this can be understood as a *mode* of the system. A control state is an abstraction of a set of ASM states having the same value for the phase function. The main rule of a control state ASM is a parallel of conditional rules checking the value of the phase function: in this way, the evolution of the machine depends on the current mode. The model in Fig. 1 is an example of control state ASM. A control state ASM naturally induces an FSM-like representation, where each state corresponds to one value of the phase function. Since such class of ASMs occur quite frequently, we implemented in `AsmetaVis` also a *semantic visualizer* that is able to visualize the FSM-like representation of a control state ASM. The visualization consists in a graph where control states are shown using orange ellipses. The semantic visualization of the ground model is shown in Fig. 3b. The initial control state is identified by the `PREPARATION` phase; from there, the system moves to the `INITIATION` phase by executing rule `r_run_preparation`; then, it moves to the `ENDING` phase by executing rule `r_run_initiation`. In the `ENDING` phase, rule `r_run_ending` is executed, but this does not modify the phase. Note that this visualization turned out to be quite useful, as it allows to get an understanding of the system evolution without the need of simulating the model.

Automatic Model Review. Due to a low familiarity of the formal method, during the development of a formal model, the developer can introduce different types of errors: domain specific ones (i.e., a wrong implementation of the system requirements), and non-domain specific ones that depend on the wrong usage of the method. In order to capture the former category of errors, domain specific properties derived from the requirements need to be verified; for the latter category, instead, automatic techniques can be devised.



(a) Selection of the desired meta-properties (b) Example of error found by MP6

Fig. 4. Model reviewer `AsmetaMA`

Based on our experience in modeling with the ASM method and in teaching it to students, we noticed that one of the main modelling (i.e., non-domain specific) errors is related to the computational model of the ASMs, which is based on parallel execution of function updates. If not properly guarded, they could lead to inconsistent results by simultaneously updating the same location to two different values (this is know as *inconsistent update* [18]). Such problem is usually difficult to observe by a manual review of the code, and it is usually only discovered during simulation.

Another problem that we observed frequently with our students is that, due to wrong rule guards, some transition rules can never be executed.

As a minor problem, we also observed that our students tend to write *over-specified* models containing unnecessary functions (that are never used); these could be either really unnecessary, and so removed, or they should be used in some rule that has not been implemented yet.

On the base of the previously described experience, in [7], we proposed the `AsmetaMA` tool that performs *automatic* review of ASMs. The tool checks whether the model contains typical errors that are usually done during the modeling activity using ASMs (suitable *meta-properties* specified in CTL are checked with the model checker `AsmetaSMV` [6]). Figure 4a shows the selection of the available meta-properties in the tool.

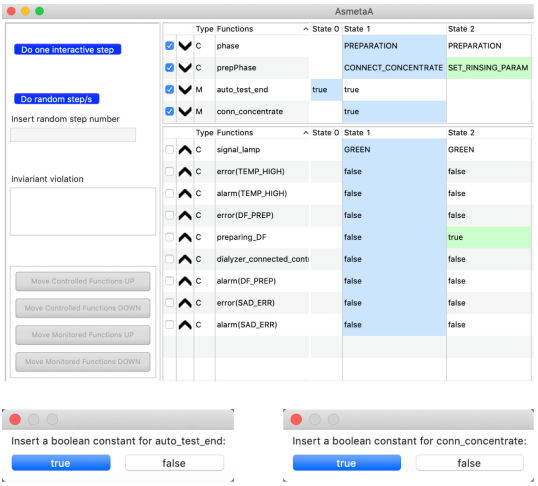
For example, MP1 checks that no inconsistent update ever happens, and MP7 that all the model locations are used somewhere in the model. Model reviewer has been extremely helpful also in our modeling of complex case studies. For example, Fig. 4b shows an error that we were able to automatically find when developing the model of a landing gear system [17]: function `gears` should become `RETRACTED` when it is `RETRACTING`, but we wrongly updated it to `EXTENDED`. The meta-property MP6, that checks that each location assumes any possible value, allowed to (indirectly) spot this error.

4.2 Validation

One of the first analysis activities that is performed while writing a formal model is *validation* to check that the model reflects the intended requirements. The

```
Insert a boolean constant for auto_test_end:
true
<State 0 (monitored)>
auto_test_end=true
</State 0 (monitored)>
<State 1 (controlled)>
alarm(DF_PREP)=false
alarm(SAD_ERR)=false
alarm(TEMP_HIGH)=false
dialyzer_connected_contr=false
error(DF_PREP)=false
error(SAD_ERR)=false
error(TEMP_HIGH)=false
phase=PREPARATION
prepPhase=CONNECT_CONCENTRATE
preparing_DF=false
signal_lamp=GREEN
</State 1 (controlled)>
Insert a boolean constant for conn_concentrate:
true
<State 1 (monitored)>
conn_concentrate=true
</State 1 (monitored)>
<State 2 (controlled)>
alarm(DF_PREP)=false
alarm(SAD_ERR)=false
alarm(TEMP_HIGH)=false
dialyzer_connected_contr=false
error(DF_PREP)=false
error(SAD_ERR)=false
error(TEMP_HIGH)=false
phase=PREPARATION
prepPhase=SET_RINSING_PARAM
preparing_DF=true
signal_lamp=GREEN
</State 2 (controlled)>
```

(a) Textual



(b) With `AsmetaA`

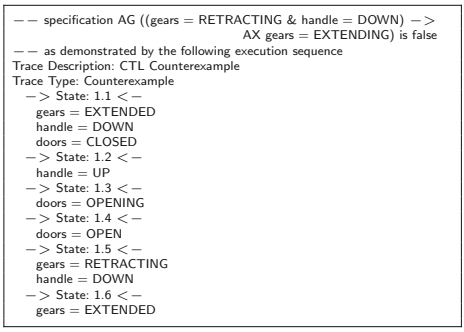
Fig. 5. Simulation

main validation technique of the ASMETA framework is simulation, in which inputs are interactively provided to the model by the user who can then check the produced state. Figure 5a shows two steps of the textual simulation (using the simulator `AsmetaS` [26]) of the second refined model of the haemodialysis case study [4].

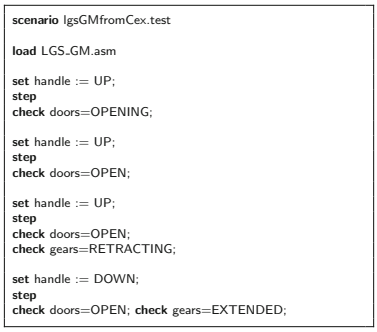
In this case, the user sets the value of monitored functions `auto_test_end` and `conn_concentrate`; the main functions of interest that the user wants to observe are `phase` and `prepPhase`. However, as shown by this small example, at every step, the whole state is printed, and checking that the simulation is as expected may become difficult as the state size grows. In order to tackle this issue and improve the *understandability* of the simulation traces, we developed the graphical animator `AsmetaA` that allows to select which functions to show, provides dialog boxes to select the values of monitored functions, and highlights the functions that have changed value in the new state. Figure 5b shows the visual simulation of the previous example, in which only the functions of interest have been selected (in the top half of the window).

4.3 Verification

The framework supports verification by model checking with the `AsmetaSMV` tool [6]. The tool translates an AsmetaL model to a model of the model checker



(a) Original counterexample



(b) Counterexample in `Avalla`

Fig. 6. Reproduction of `AsmetaSMV` counterexamples

NuSMV[3], performs the verification with NuSMV, and translates the output back in terms of AsmetaL locations. We tried to improve the usability of this tool in different directions.

Smoothness of Use. First of all, the model checker is transparent to the user who interacts with only one tool: (i) (s)he can specify the properties directly in the AsmetaL model using the AsmetaL syntax, (ii) the invocation of NuSMV is done directly only with the framework, (iii) and the output is captured and pretty-printed in terms of AsmetaL locations.

Reproducibility of Counterexamples. In model checking, when a property that should hold is falsified, the model must be fixed in order to satisfy the property (unless the property itself is wrong). To assist the developer in this activity, we developed a translator from the model checker counterexamples to `Avalla` *scenarios*. `Avalla` scenarios allow to describe simulation sequences by providing commands to **set** the values of monitored functions, to perform a **step** of simulation, and to **check** that the output is as expected; the `AsmetaV` tool is able to read `Avalla` scenarios and execute them using the simulator `AsmetaS`. By translating a counterexample in an `Avalla` scenario, the developer, while debugging the model, can rerun it as many times as needed, till the wrong behaviour is removed from the model. In this way, we achieved *interoperability* of the tools, and a better *understandability* of the verification results. Figure 6 shows the counterexample of a property for the landing gear system checking that when the gears are retracting and the handle is pushed down, the gears must start extending.

The violation occurred in a preliminary version of the model (as explained in Sect. 4.1). Figure 6b shows the `Avalla` translation of the counterexample.

[3] http://nusmv.fbk.eu/.

```
macro rule r_changeOrganization($c in Camera) =
  par
    let ($getMasterCameraOCself = getMaster($c)) in
      let ($prevGetMasterCameraOCself = prev($getMasterCameraOCself)) in
        par
          r_setMaster[$prevGetMasterCameraOCself]
          if not newSlave($prevGetMasterCameraOCself, $c) then
            newSlave($prevGetMasterCameraOCself, $c) := true
          endif
        endpar
      endlet
    endlet
    change_master($c) := false
  endpar
```

(a) Without flattener

```
macro rule r_changeOrganization($c in Camera) =
  par
    r_setMaster[prev(getMaster($c))]
    if not newSlave(prev(getMaster($c)), $c) then
      newSlave(prev(getMaster($c)), $c) := true
    endif
    change_master($c) := false
  endpar
```

(b) With flattener

Fig. 7. `AsmetaSMV` – models suitable for model checking

Supporting a Large Class of ASMs. ASMs can describe infinite state systems; however, for model checking, only finite state ASMs having finite domains are admissible. While this limitation is unavoidable, when we originally proposed the tool, we had to impose further restrictions on the class of ASMs that could be translated. Indeed, the AsmetaL language provides a powerful language that allows to describe complex systems in a concise way. While this is advantageous from a modeling point of view, it complicates the mapping to target languages such as NuSMV that have much simpler notations. Some constructs of the ASM formalism are indeed difficult to translate in the target notation, and, although possible, we did not implement such translations because too complex. For example, originally we did not support variable arguments in functions; if the user wanted to use them, (s)he had to write the model as shown in Fig. 7a (taken from [14] where we made the formalization of a self-adaptive system), where the function arguments are made explicit by means of a let rule.

This turned out to be a quite strong limitation; indeed, we noticed that our students were used to write quite compact and elegant models at first, but then this constituted a problem when they had to do model checking, as they had to refactor the ASM model in unnatural ways. Therefore, in [13] we introduced a tool that *flattens* the ASM before being translated to NuSMV; the flattened ASM is a kind of *normal form* that only contains parallel, update, and choose rules. Such kind of ASM is supported by `AsmetaSMV`; in this way, we have been able to enlarge the class of models supported by the tool, so allowing a *smoother* use of the model checker. Figure 7b shows a model equivalent to the one in Fig. 7a, in which functions are freely used as function arguments: this can be supported by the new version of `AsmetaSMV` extended with the flattener. Note that we could have achieved this also trying to modify directly the translation from ASM to NuSMV; however, not only this would have been difficult, but it would have improved only `AsmetaSMV`. The introduction of the flattener, instead, improved the capabilities of different other tools of the ASMETA framework that perform translations to other languages, namely a mapping to SPIN for test case generation [25], to SMT for proof of refinement correctness [8,10], and to C++ code [16].

5 Lessons Learned

We here provide a more critical overview of our efforts in targeting usability in the ASMETA framework. Before, we discuss which tools turned out to be useful and those that, instead, were not successful as expected. Then, we outline some of the ongoing and future efforts that should further increase the usability of the framework.

5.1 Critical Review of Previous Efforts

We should say that not all the techniques we applied for improving the usability of ASMETA have been equally successful. We started developing the visualizer `AsmetaVis` while we were developing the formal specification of a haemodialysis device [4]; indeed, the model was so big that we needed a better way to visualize its structure than the textual model. Although this was extremely helpful for us, it is not used very frequently by our students. The reasons could be different. First of all, the models they develop are not usually too big, and usually they can already have an overview of the model by scrolling once or twice the textual representation. Moreover, students are already used to code and it could be that they do not feel the need of such visualization facilities. We still believe that the visualizer has some potentials for communicating the model structure; however, we need further investigation with different stakeholders (other than students) less accustomed to code.

Among the tools that we introduced to improve the method usability, the animator `AsmetaA` has been one of the most successful. Indeed, reading long simulation traces has always been annoying both for us and our students; first, small models can already have tens of locations and their listing can be long; second, if the listing of a state is long, understanding what has changed between two states is not trivial. The animator solved these issues by allowing to customize which locations to show, and by highlighting those that have been changed.

As we discussed in Sect. 4.3, the introduction of the flattener allowed to enlarge the class of ASMs that could be model checked; the users can now write the ASM model as they wish, with any degree of nesting and compactness. While this is a clear improvement, it also introduced an unexpected drawback. Since the users have a lot of freedom in writing the model, they do not consider anymore that this will be translated for model checking and, therefore, often they write models so complicated that then their verification does not scale. From the experience with our students, we noticed that, when they were constrained by the limitation of the tool (e.g., they could not use functions as argument of other functions), they tended to write simpler models that scaled better. Our observation is that a too high-level notation could detach the user from the computational complexity of verification tasks; therefore, there is the need for some approaches that give the idea of the model complexity: these could be inspired by code metrics as cyclomatic complexity, cohesion, etc.

Being ASMETA an academic tool developed for research, most of the tools have been originally developed as complement of some research work. As such,

the implementation usually reached the point in which the research result was evident and could be published; due to deadline pressure, the usability of the tool was sometimes sacrificed. This was the case for the `AsmetaSMV` tool for which we originally restricted the class of ASMs that could be translated. We believe that, as research community, we have to promote initiatives that incentive the production of tools not only innovative from the research point of view, but also usable. Artefacts evaluations, now applied by major conferences as CAV and TACAS, are good initiatives going in this direction.

5.2 Ongoing Efforts and Future Work

As explained before, the visualizer `AsmetaVis` is not used too much because some users (as our students) are accustomed to code. However, there are still problems in managing large models. One solution could be to improve the textual editor by allowing folding/unfolding facilities as those available in main IDEs for programming languages.

CoreASM is the other major framework for ASMs [24]; ASMETA and CoreASM are somehow complementary, as CoreASM mainly provides support for model debugging, while ASMETA more focuses on simulation-based validation, and automated verification. Being able to write models that are compatible with both frameworks would highly increase the usability of the ASM method, as a user could use all the available tools. As an attempt in this direction, in [3] we proposed a uniform syntax that should be accepted by both frameworks, so that a designer can use all the available tools for ASMs. However, such integration (that is still ongoing) is not trivial, as there are different aspects that need to be merged (e.g., AsmetaL is typed, while CoreASM is not). We believe that the effort spent for this integration is worthy, as standard notations are usually beneficial for the tools that adopt them, as demonstrated by the DIMACS notation for SAT solvers and by SMT-LIB for SMT solvers.

Model refinement [1] is one of the principles of the ASM method [18], as of other methods as B [2] and Z [21]. It consists in developing models incrementally, from a high-level description of the system to more detailed ones, by adding, at each refinement step, design decisions and implementation details. The ASM notion of correct refinement is based on the correspondence of abstract and refined runs; in the framework, we provide an SMT-based tool [8] that is able to prove a particular kind of refinement correctness. However, the framework does not help the designer in deciding what to refine and does not provide support for documenting the refinement decisions; although doing a good refinement depends on the modelling skills of the developer, we believe that a proper tool support could help in obtaining more meaningful refinement steps. For example, we could allow the user to specify which abstract rule is refined in which refined rule(s); this would also help in performing more tailored refinement proofs, as we would exactly know what needs to be related in the SMT-based proof. Moreover, this would also improve incremental test generation techniques that combine refinement and conformance testing [15].

6 Related Work

Due to the lack of space, a complete survey on usability in formal methods is not possible. We only refer to some approaches that have achieved usability using approaches similar to those we proposed.

The formal method community seems to recognize the importance of having visualization techniques (similar to our visualizer `AsmetaVis`) [23,35,41], and there are positive success stories showing that the use of these visualization techniques makes the use of formal methods feasible also for non-experts [35], and also helps in teaching formal methods [34].

Some approaches perform *model visualization* [22,29] (similar to our basic visualization in `AsmetaVis`), while others provide a visual representation of the model execution (or *model animation*) [32,33]. Among these, ProB [32] is one of the most successful tools; it performs animation of B models, and can also be used for error and deadlock checking (similar to our model reviewer `AsmetaMA`), and test-case generation.

Other approaches use UML-like notation as modeling front-end. UML-B [39] uses the B notation as an action and constraint language for UML, and defines the semantics of UML entities via a translation into B. In a similar way, in [36], transforming rules are given from UML models to Object-Z constructs. In the method SPACE and its supporting tool Arctis [31], services are composed of collaborative building blocks that encapsulate behavioral patterns expressed as UML 2.0 collaborations and activities.

Regarding model review, different approaches have been proposed for different formal methods. They all automatize some checks that are usually performed manually by human reviewers; Parnas, in a report about the certification of a nuclear plant, observed that "reviewers spent too much of their time and energy checking for simple, application-independent properties which distracted them from the more difficult, safety-relevant issues" [37]. Approaches for automatic model review have been proposed, e.g., for Software Cost Reduction (SCR) models [28], software requirements specifications (SRS) [30], and UML [38].

7 Conclusions

The paper presented our efforts in addressing usability in the ASMETA framework, and a critical review of what has been more successful and what less.

Note that all our conclusions are only based on our experience; properly assessing the usability of a method/technique would need user studies that, however, are difficult and costly to conduct. Moreover, we defined *usability* according to our understanding, and not relying on notions of usability provided by the Human-Computer Interaction community [20,40]; as future work, it would be interesting to investigate which of those concepts also apply to our framework and which, instead, we are not targeting.

Moreover, all our observations come from the use of the framework by us and by our students; we do not know what would work and what wouldn't in an industrial context.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. Theoret. Comput. Sci. **82**(2), 253–284 (1991). https://doi.org/10.1016/0304-3975(91)90224-P
2. Abrial, J.R., Hallerstede, S.: Refinement, decomposition, and instantiation of discrete models: application to Event-B. Fundam. Inform. **77**(1), 1–28 (2007)
3. Arcaini, P., et al.: Unified syntax for abstract state machines. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) ABZ 2016. LNCS, vol. 9675, pp. 231–236. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33600-8_14
4. Arcaini, P., Bonfanti, S., Gargantini, A., Mashkoor, A., Riccobene, E.: Integrating formal methods into medical software development: the ASM approach. Sci. Comput. Program. **158**, 148–167 (2018). https://doi.org/10.1016/j.scico.2017.07.003
5. Arcaini, P., Bonfanti, S., Gargantini, A., Riccobene, E.: Visual notation and patterns for abstract state machines. In: Milazzo, P., Varró, D., Wimmer, M. (eds.) STAF 2016. LNCS, vol. 9946, pp. 163–178. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50230-4_12
6. Arcaini, P., Gargantini, A., Riccobene, E.: AsmetaSMV: a way to link high-level ASM models to low-level NuSMV specifications. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 61–74. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11811-1_6
7. Arcaini, P., Gargantini, A., Riccobene, E.: Automatic review of Abstract State Machines by meta property verification. In: Muñoz, C. (ed.) Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215, pp. 4–13. NASA, Langley Research Center, Hampton, April 2010
8. Arcaini, P., Gargantini, A., Riccobene, E.: SMT-based automatic proof of ASM model refinement. In: De Nicola, R., Kühn, E. (eds.) SEFM 2016. LNCS, vol. 9763, pp. 253–269. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41591-8_17
9. Arcaini, P., Gargantini, A., Riccobene, E.: Rigorous development process of a safety-critical system: from ASM models to Java code. Int. J. Softw. Tools Technol. Transfer **19**(2), 247–269 (2015). https://doi.org/10.1007/s10009-015-0394-x
10. Arcaini, P., Gargantini, A., Riccobene, E.: SMT for state-based formal methods: the ASM case study. In: Shankar, N., Dutertre, B. (eds.) Automated Formal Methods. Kalpa Publications in Computing, vol. 5, pp. 1–18. EasyChair (2018)
11. Arcaini, P., Gargantini, A., Riccobene, E., Scandurra, P.: A model-driven process for engineering a toolset for a formal method. Softw. Pract. Exp. **41**, 155–166 (2011). https://doi.org/10.1002/spe.1019
12. Arcaini, P., Holom, R.-M., Riccobene, E.: ASM-based formal design of an adaptivity component for a Cloud system. Formal Aspects Comput. **28**(4), 567–595 (2016). https://doi.org/10.1007/s00165-016-0371-5
13. Arcaini, P., Melioli, R., Riccobene, E.: AsmetaF: A flattener for the ASMETA framework. In: Masci, P., Monahan, R., Prevosto, V. (eds.) Proceedings 4th Workshop on Formal Integrated Development Environment, Oxford, England, 14 July 2018. Electronic Proceedings in Theoretical Computer Science, vol. 284, pp. 26–36. Open Publishing Association (2018). https://doi.org/10.4204/EPTCS.284.3
14. Arcaini, P., Riccobene, E., Scandurra, P.: Formal design and verification of self-adaptive systems with decentralized control. ACM Trans. Auton. Adapt. Syst. **11**(4), 251–2535 (2017). https://doi.org/10.1145/3019598

15. Bombarda, A., Bonfanti, S., Gargantini, A., Radavelli, M., Duan, F., Lei, Y.: Combining model refinement and test generation for conformance testing of the IEEE PHD protocol using Abstract State Machines. In: Gaston, C., Kosmatov, N., Le Gall, P. (eds.) ICTSS 2019. LNCS, vol. 11812, pp. 67–85. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31280-0_5
16. Bonfanti, S., Carissoni, M., Gargantini, A., Mashkoor, A.: `Asm2C++`: a tool for code generation from abstract state machines to Arduino. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NFM 2017. LNCS, vol. 10227, pp. 295–301. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57288-8_21
17. Boniol, F., Wiels, V., Aït-Ameur, Y., Schewe, K.-D.: The landing gear case study: challenges and experiments. Int. J. Softw. Tools Technol. Transfer **19**(2), 133–140 (2016). https://doi.org/10.1007/s10009-016-0431-4
18. Börger, E., Raschke, A.: Modeling Companion for Software Practitioners. Springer, Heidelberg (2018). https://doi.org/10.1007/978-3-662-56641-1
19. Bowen, J.P., Hinchey, M.G.: Seven more myths of formal methods: Dispelling industrial prejudices. In: Naftalin, M., Denvir, T., Bertran, M. (eds.) FME 1994. LNCS, vol. 873, pp. 105–117. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58555-9_91
20. Brooke, J.: SUS: a retrospective. J. Usability Stud. **8**(2), 29–40 (2013)
21. Derrick, J., Boiten, E.: Refinement in Z and object-Z: Foundations and Advanced Applications. Springer, London (2001). https://doi.org/10.1007/978-1-4471-5355-9
22. Dick, J., Loubersac, J.: Integrating structured and formal methods: a visual approach to VDM. In: van Lamsweerde, A., Fugetta, A. (eds.) ESEC 1991. LNCS, vol. 550, pp. 37–59. Springer, Heidelberg (1991). https://doi.org/10.1007/3540547428_42
23. Dulac, N., Viguier, T., Leveson, N., Storey, M.A.: On the use of visualization in formal requirements specification. In: 2012 IEEE Joint International Conference on Requirements Engineering. Proceedings, pp. 71–80. IEEE (2002)
24. Farahbod, R., Glässer, U.: The CoreASM modeling framework. Softw. Pract. Exp. **41**(2), 167–178 (2011). https://doi.org/10.1002/spe.1029
25. Gargantini, A., Riccobene, E., Rinzivillo, S.: Using spin to generate tests from ASM specifications. In: Börger, E., Gargantini, A., Riccobene, E. (eds.) ASM 2003. LNCS, vol. 2589, pp. 263–277. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36498-6_15
26. Gargantini, A., Riccobene, E., Scandurra, P.: A metamodel-based language and a simulation engine for abstract state machines. J. UCS **14**(12), 1949–1983 (2008). https://doi.org/10.3217/jucs-014-12-1949
27. Hall, A.: Seven myths of formal methods. IEEE Softw. **7**(5), 11–19 (1990). https://doi.org/10.1109/52.57887
28. Heitmeyer, C.L., Jeffords, R.D., Labaw, B.G.: Automated consistency checking of requirements specifications. ACM Trans. Softw. Eng. Methodol. **5**(3), 231–261 (1996). https://doi.org/10.1145/234426.234431
29. Kim, S.K., Carrington, D.: Visualization of formal specifications. In: Proceedings of the Sixth Asia Pacific Software Engineering Conference, APSEC 1999, p. 102. IEEE Computer Society, Washington (1999). https://doi.org/10.1109/APSEC.1999.809590
30. Kim, T., Cha, S.: Automated structural analysis of SCR-style software requirements specifications using PVS. Softw. Test. Verif. Reliab. **11**(3), 143–163 (2001). https://doi.org/10.1002/stvr.218

31. Kraemer, F.A., Slåtten, V., Herrmann, P.: Tool support for the rapid composition, analysis and implementation of reactive services. J. Syst. Softw. **82**(12), 2068–2080 (2009). https://doi.org/10.1016/j.jss.2009.06.057
32. Ladenberger, L., Bendisposto, J., Leuschel, M.: Visualising event-B models with B-motion studio. In: Alpuente, M., Cook, B., Joubert, C. (eds.) FMICS 2009. LNCS, vol. 5825, pp. 202–204. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04570-7_17
33. Leuschel, M., Bendisposto, J., Dobrikov, I., Krings, S., Plagge, D.: From Animation to Data Validation: The ProB Constraint Solver 10 Years On, pp. 427–446. Wiley (2014). https://doi.org/10.1002/9781119002727.ch14
34. Leuschel, M., Samia, M., Bendisposto, J.: Easy graphical animation and formula visualisation for teaching B. In: The B Method: From Research to Teaching (2008)
35. Margaria, T., Braun, V.: Formal methods and customized visualization: a fruitful symbiosis. In: Margaria, T., Steffen, B., Rückert, R., Posegga, J. (eds.) Services and Visualization Towards User-Friendly Design. LNCS, vol. 1385, pp. 190–207. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0053506
36. Miao, H., Liu, L., Li, L.: Formalizing UML models with object-Z. In: George, C., Miao, H. (eds.) ICFEM 2002. LNCS, vol. 2495, pp. 523–534. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36103-0_53
37. Parnas, D.L.: Some theorems we should prove. In: Joyce, J.J., Seger, C.-J.H. (eds.) HUG 1993. LNCS, vol. 780, pp. 155–162. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-57826-9_132
38. Prochnow, S., Schaefer, G., Bell, K., von Hanxleden, R.: Analyzing robustness of UML state machines. In: Workshop on Modeling and Analysis of Real-Time and Embedded Systems (MARTES 2006) (2006)
39. Snook, C., Butler, M.: UML-B: formal modeling and design aided by UML. ACM Trans. Softw. Eng. Methodol. **15**(1), 92–122 (2006). https://doi.org/10.1145/1125808.1125811
40. Speicher, M.: What is usability? A characterization based on ISO 9241–11 and ISO/IEC 25010. CoRR abs/1502.06792 (2015)
41. Spichkova, M.: Human factors of formal methods. CoRR abs/1404.7247 (2014)