Visual notation and patterns for Abstract State Machines

Paolo Arcaini[1], Silvia Bonfanti[2], 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] Department of Economics and Technology Management, Information Technology and Production, Università degli Studi di Bergamo, Italy
{silvia.bonfanti,angelo.gargantini}@unibg.it
[3] Dipartimento di Informatica, Università degli Studi di Milano, Italy
elvinia.riccobene@unimi.it

Abstract. Formal models are a rigorous way to specify informal system requirements. However, they are not widely used in practice, since they are considered difficult to develop and understand. Visualization is often considered a good means for people to communicate and to get a common understanding. We here make a proposal of a visual notation for Abstract State Machines (ASMs), and we introduce *visual trees* that visualize ASM transition rules. In addition to these graphical components that are based only on the syntactical structure of the model, we also present *visual patterns* that permit to visualize part of the behavior of the machine. A tool is also available to graphically represent ASM models using the proposed notation.

1 Introduction

Formal models are in principle accepted as the only way to specify in a precise and rigorous way the informal system requirements: they help to understand what has to be developed and to prove properties already at the early stages of the system development. However, formal specification languages are not widely used in industry, and practitioners largely consider formal methods "too hard to understand and use in practice". Limiting factors are the lack of *simplicity*, *learnability*, *readability*, *easiness of use* of formal notations [23]. All these nonfunctional properties are fundamentals to achieve easiness of development and comprehension of models, particularly for large, complex software systems. Requirement models should act as a communication medium among customers, users, designers, developers, and this common understanding is fundamental for the success of the system realization. However, since the mathematical notation is not always intuitive, and the size of the specification often consists of several pages of rules and formulas, model comprehension is threatened.

Visualization is considered as a good means for people to communicate and to get a common understanding. Indeed, the use of diagrams and graphical blocks is at the base of the mostly used notations in industry, as FSMs (and their extensions) or UML, the latter nowadays accepted as the industrial standard for system design. However, their shortcomings, as limited expressiveness for FSM w.r.t. other formal notations [6] or semantics lack for UML [7], are well-known.

Ever since UML appeared, many modeling approaches have been developed which try to use UML (or one of its profiles or domain-specific UML-like notations) as front-end of the requirements specification and formal notations as back-end of the process, to provide rigor and preciseness to lightweight models and make model validation and verification possible [22,20,21,18,12].

Abstract State Machines (ASMs) are an extension of FSMs, obtained by replacing unstructured control states by states comprising arbitrarily complex data [6]. ASMs have been widely used as requirement specification formalism. Despite of their mathematical foundation, a practitioner needs no special training to use the method since ASMs can be correctly understood as pseudo-code (or virtual machines) working over abstract data structures. Furthermore, to ease its use by non-experts, a series of integrated tools (for editing, validation, and verification) have been developed around ASMs [4].

Although the ASM textual notation [11] has been designed with readability in mind, our experience in trying to build and read very large system specifications [3,1] has shown that the complexity of the behavior being described overwhelms the reader, and most users (even the authors of the specification) need help in navigating and understanding it. This also happened while we were developing the ABZ 2016 case study [2], that motivated the current work. We tried, at first, to directly specify the ASM models from the textual description of the requirements. Although the refinement process helped us in managing the complexity of the case study, we still had some problems in discussing among us about the solution. So, we started making some drawings, whose notation was inspired by different sources: control flow graphs, UML state machines, sequence diagrams, etc. The lack of a way to graphically represent ASM models was clear.

A further observation we have made is that most of the new ASM users start developing ASM models as control state ASMs, a particular frequent class of ASMs – proposed by Börger in [6] – useful to model system modes (or control states). Control state ASMs have an intuitive graphical representation by means of FSM-like state diagrams. However, when the system to model is very complex, the resulting control state is too complicated and fails in achieving its main aim, i.e., easily communicating the behavior of the system. Moreover, a systematic use of control state ASMs is missing, and there is no algorithmic support to build or reconstruct such machines from models written in textual notations.

Starting from the motivations that (a) formality is important but also understanding and communicating among stakeholders is fundamental, (b) visualization of formal models can surely aid the understanding of model structure and behaviors, (c) visual editing is often used to help designers and developers to graphically build complex models [9], we here propose a graphical notation for ASMs. The overall visualization of a model is given in terms of a graph. In addition, we define *structural* patterns, useful to visualize the structure of a model in a more compact way, and *semantic* patterns to be used when additional information on the machine workflow can be inferred from the model.

The paper is organized as follows. Sect. 2 gives a brief background on ASMs. In Sect. 3, we introduce our proposal of a visual notation for ASMs, whose basic constituents (i.e., visual trees) are defined in Sect. 4. Sect. 5 shows that ASM models usually contain particular recurring patterns of ASM rules that can be visualized in a proper way: some patterns are simply structural, whereas others permit to infer some of the behavioral semantics of the ASM. Sect. 6 presents the prototypical implementation of a tool supporting the proposed visual notation. Sect. 7 describes a preliminary evaluation of the tool. Sect. 8 discusses some related work, and Sect. 9 concludes the paper.

2 Abstract State Machines

Abstract State Machines (ASMs) [6] are an extension of FSMs, where unstructured control states are replaced by states with arbitrary complex data. Although the method has a rigorous mathematical foundation, a practitioner can simply understand ASMs as pseudo-code working over abstract data structures.

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 container. The couple (*location*, *value*) represents a machine memory unit. Therefore, ASM states can be viewed as abstract memories.

Values of locations can be changed by firing *transition rules*. They express the modification of functions interpretation from one state to the next one. Location *updates* are the basic units of rules construction and are given as assignments of the form $loc := v$, where loc is a location and v its new value. The description of all basic ASM transition rules is given in Table 1.

An ASM *computation* is a finite or infinite sequence $S_0, S_1, \ldots, S_n, \ldots$ of states of the machine, where S_0 is an initial state and each S_{n+1} is obtained from S_n by firing the unique *main rule* which can fire other transitions rules.

There exists a classification of ASM functions that, however, is not relevant for understanding the current work and, therefore, is here skipped.

A set of tools exists to support the ASM modeling process. Tools are part of the ASMETA (ASM mETAmodeling) framework[4] [4], 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 using the AsmetaL textual syntax [11], storage, interchange, access, etc.), and supports advanced model analysis techniques (as validation, verification, testing, model review, requirements analysis, runtime verification, etc.).

3 A visual notation for ASMs

In this section, we introduce the meaning, the goals, and the possible usage scenarios for the proposed visual notation for ASM models.

The proposed visual notation is defined in terms of a set of construction rules and schemas that give a graphical representation of an ASM and its rules.

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

(a) Textual representation (b) Graphical representation

Fig. 1: Visual notation

We assume that the graphical information is represented by a visual graph in which nodes represent syntactic elements (like rules, conditions, rule invocations) or states, while edges represent bindings between syntactic elements or state transitions. We do not introduce a graphical representation for the signature (functions and domains) and properties, since we believe that they can be already easily understood from the textual model.

In the following sections, we propose a set of procedures that allow to automatically derive a *visual graph* from an ASM model. Sect. 4 introduces procedures that recursively visit the ASM rules and build a *visual tree* representing the syntactical structure of the model. In Sect. 5, we introduce some *visual patterns* that permit to identify recurring graphical schemas, and to obtain a more compact and meaningful representation, possibly capturing some behavioral information. Such representation may be no longer a tree, but a general graph.

The final goal is to have a textual representation together with a graphical visualization as shown in Fig. 1. To be more precise, we have devised two possible usage scenarios of the proposed visual notation.

Visualization – From textual to graphical representation The first usage scenario consists in writing an ASM model in a concrete syntax (AsmetaL) and then derive a graph from it. Such approach can be used when the modeler is familiar with the ASM syntax, but (s)he wants to have a graphical representation of the model for its better understanding and communication. If the ASM model is correct, also the produced graph is correct. In the visualizer, the user can activate some optimizations (presented in the following sections), in order to have different views of the same model: structural (with different levels of optimization), or semantic (behavioral).

Visual editing – From graphical to textual representation The second usage scenario consists in graphically specifying the ASM by drawing the graph. In this way, the modeler can focus on the high level structure of the model, similarly to what is done in code with control flow graphs. Note that the usage of semantic patterns allows the user to also graphically model some evolutions of the system, which are usually difficult to get by writing textual ASM models (at least without simulating it). Of course, the graph produced by the developer is not complete, as it does not specify the signature; moreover, it could also be not correct. Some trivial syntactical violations can be discovered directly on the graph by checking some consistency rules, but other faults may be more difficult to find. Once the modeler has produced the graph, a `translator` can translate the graph in an AsmetaL textual model. The produced model contains (most of) the transition rules, and the modeler is only required to add the signature (and the initialization). Then, the AsmetaL parser may find some faults that passed undetected during graph validation.

4 Visual Trees

We here introduce the relevant concepts which bring to a graphical representation of an ASM model in terms of a navigable forest of tree structures, i.e., a forest of trees connected among them by navigation links.

Definition 1. *The* visual notation *for ASMs is given by the bijective function* vis_T *between an ASM rule and a visual tree.*

Definition 2. *The function* vis_T *is given by Table 1.*
1. *For basic rules (update, skip and macro call) the function simply returns a tree with only one node (the root).*
2. *For compound rules (conditional, block, forall, choose, let), one must apply the schema given in Table 1 and recursively call the function* vis_T *on component rules.*

Table 1 describes the semantics of ASM transition rules, and shows the proposed graphical representation and the AsmetaL textual notation. The function vis_T is only based on the syntactical structure of the ASM and it can be always applied. Tree leaves are always skip, update, or call rules, and they are shown in boxes. Note that a call rule invokes a macro rule that has its own tree that, however, is not part of the main tree. At the end, one can obtain a tree for every rule declaration by applying vis_T to its definition. The visualization of an ASM is given by the forest compound of all the trees of the declared rules. To navigate this visual view, the entry point is the tree for the main rule and, from every call rule, one can navigate to the tree of the invoked macro rule by a virtual *navigation link*, which is not visualized in the graphical representation. By considering the navigation links in the visualization, the resulting structure is a graph, as a macro rule can be called by different call rules.

Example 1. For explanation purposes, we use the Hemodialysis Machine Case Study [2]. It describes a hemodialysis device which goes through three phases: the *preparation* in which the device is prepared and the patient is connected, the *initiation* in which the hemodialysis is performed (i.e., the patient's blood is cleaned), and the *ending* in which the therapy terminates and the patient is disconnected. We can abstractly describe the device using the ASM model shown

Rule	Visual tree	AsmetaL notation
`Skip rule` do nothing	skip	skip
`Update rule` update f to v	f := v	f := v
`Macro call rule`	r_rule[]	r_rule[]
invoke rule *r_rule* with arguments $\overline{v}$ (if any)	r_rule[$\overline{v}$]	r_rule[$\overline{v}$]

`Conditional rule`	guard → vis_T (rule1)	**if** guard **then**
		rule1
		endif

execute *rule1* if *guard* holds, otherwise execute *rule2* (if given)	guard —true→ vis_T (rule1)
	guard —false→ vis_T (rule2)
		if guard **then**
		rule1
		else
		rule2
		endif

`Block rule` execute *rule1* ... *rulen* in parallel	par → vis_T (rule1) par → vis_T (rule2) ... par → vis_T (rule$n$)	**par** rule1 rule2 ... rulen **endpar**
`Forall rule` execute *rule1* with all values $\overline{v} \in \overline{V}$ for which d($\overline{v}$) holds	forall —$\overline{v} \in \overline{V}$→ d($\overline{v}$) → vis_T (rule1[$\overline{v}$])	**forall** $\overline{v} \in \overline{V}$ **with** d($\overline{v}$) **do** rule1[$\overline{v}$]
`Choose rule`	choose —$\overline{v} \in \overline{V}$→ d($\overline{v}$) → vis_T (rule1[$\overline{v}$])	**choose** $\overline{v} \in \overline{V}$ **with** d($\overline{v}$) **do** rule1[$\overline{v}$]
execute *rule1* with a $\overline{v} \in \overline{V}$ for which d($\overline{v}$) holds. If no such $\overline{v}$ exists, execute *rule2* (if given)	choose —$\overline{v} \in \overline{V}$→ d($\overline{v}$) → vis_T (rule1[$\overline{v}$]) d($\overline{v}$) —ifnone→ vis_T (rule2)	**choose** $\overline{v} \in \overline{V}$ **with** d($\overline{v}$) **do** rule1[$\overline{v}$] **ifnone** rule2
`Let rule` execute *rule1* substituting $\overline{t}$ for $\overline{x}$	let —$\overline{x} = \overline{t}$→ vis_T (rule1[$\overline{x}$])	**let**($\overline{x} = \overline{t}$) **in** rule1[$\overline{x}$] **endlet**

Table 1: vis_T: Mapping from ASM transition rules to visual trees

in Code 1[5]. Using the vis_T function, the model can be represented as shown in Fig. 2. Note that the three macro rules r_preparation, r_initiation, and r_ending have their own tree representations that are not part of the tree generated from the main rule, but are connected to their corresponding call rules by navigation links (here rendered as dashed arrows only for presentation purposes).

[5] Note that the complete formalization of the case study consists of a sequence of refined models, each one specifying more details of the therapy.

```
asm Hemodialysis_GM

signature:
  enum domain PhasesTherapy = {PREPARATION |
                               INITIATION | ENDING}
  controlled phaseTherapy: PhasesTherapy

definitions:
  macro rule r_preparation =
    phaseTherapy := INITIATION

  macro rule r_initiation =
    phaseTherapy := ENDING

  macro rule r_ending =
    skip

main rule r_Main =
  par
    if phaseTherapy = PREPARATION then
      r_preparation[]
    endif
    if phaseTherapy = INITIATION then
      r_initiation[]
    endif
    if phaseTherapy = ENDING then
      r_ending[]
    endif
  endpar

default init s0:
  function phaseTherapy = PREPARATION
```

Code 1: Hemodialysis case study – AsmetaL model

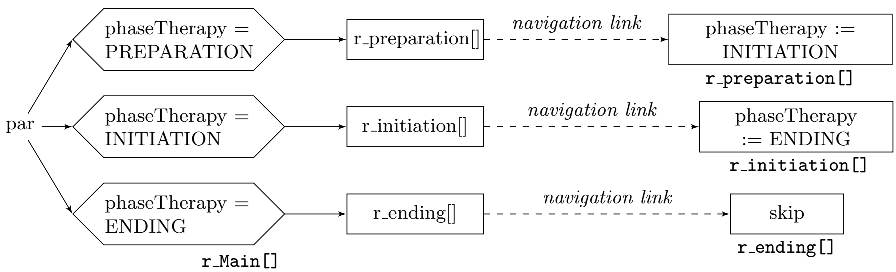


Fig. 2: Hemodialysis case study – Visual trees

5 Visual Patterns

We here introduce the notion of *visual pattern* for ASM visual trees. A pattern is a schema of connected tree nodes that is recurring and conveys a *structural* or *semantic* (i.e., behavioral) information. Therefore, identifying a pattern and substituting the entities belonging to it with a simplified structure is of interest.

5.1 Structural patterns

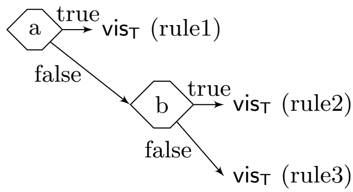
We identify the following structural pattern that permits to obtain a more compact representation of the model structure.

Nested Guards Pattern The pattern regards the use of *nested conditional rules*. Suppose that you have a conditional rule as shown in Fig. 3a. By applying the visual trees in Table 1, one would obtain the tree shown in Fig. 3b. However, one can visualize the rule in a more compact way as shown in Fig. 3c.

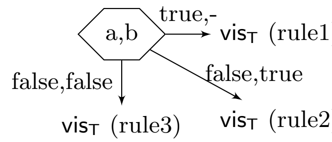
The pattern is applicable to any depth of nested conditional rules. One just has to collect all the guards $g_1, \ldots, g_n$, and create only one decision node comprising all the guards separated by commas. The decision node has as many exiting arcs as the number of conditional branches not containing another nested conditional rule, but a different rule rule$i$; each arc is labeled with the evaluations of the guards that permit to take that particular arc and fire rule rule$i$. Evaluations of guards that are not relevant for the firing of a rule rule$i$ are depicted

```
if a then
  rule1
else
  if b then
    rule2
  else
    rule3
  endif
endif
```

(a) Nested conditional rules



(b) Visual tree

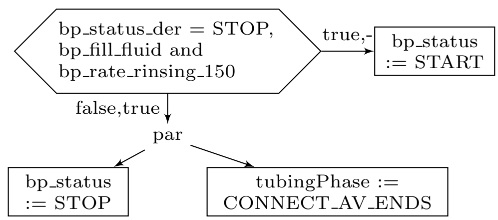


(c) Pattern

Fig. 3: Structural pattern – Nested guards pattern

```
macro rule r_priming =
  if bp_status_der = STOP then
    bp_status := START
  else
    if bp_fill_fluid and bp_rate_rinsing_150 then
      par
        bp_status := STOP
        tubingPhase := CONNECT_AV_ENDS
      endpar
    endif
  endif
```

(a) Nested conditional rules



(b) Pattern

Fig. 4: Hemodialysis case study – Nested guards pattern

with symbol "–". The decision node has up to $n + 1$ exiting arcs. Note that the pattern does not necessarily produce a tree that is more clear to understand, but it always provides a more compact representation of the nested conditional rules. For this reason, we let the modeler decide if (s)he wants to apply it.

Example 2. Fig. 4b shows the application of the pattern to macro rule `r_priming` (shown in Fig. 4a) of the hemodialysis machine case study.

5.2 Semantic Patterns

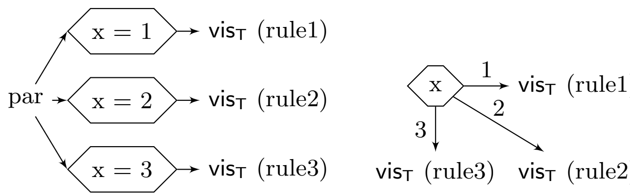
Any ASM model can be always represented using visual trees and possibly optimized by applying structural patterns. The resulting tree visualizes the structure of the ASM. However, sometimes it is possible to infer from the model also some hints on the behavior of the machine. For this reason, we introduce *semantic patterns* that can be applied when it is possible to infer from the model some information on the workflow of the machine.

We identify here three semantic patterns: *mutual exclusive guards*, *state*, and *state flow* patterns.

Mutual exclusive guards pattern In case of parallel conditional rules having mutual exclusive guards, it could be useful to represent that the workflow of the machine follows only one of the possible parallel execution paths.

```
par
  if x = 1 then
    rule1
  endif
  if x = 2 then
    rule2
  endif
  if x = 3 then
    rule3
  endif
endpar
```

(a) Parallel conditional rules



(b) Visual tree (c) Pattern

Fig. 5: Semantic pattern – Mutual exclusive guards pattern

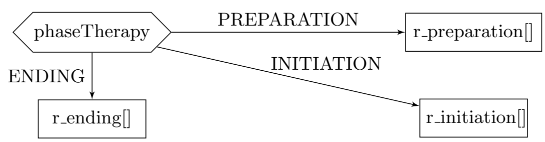


Fig. 6: Hemodialysis case study – Mutual exclusive guards pattern

The *mutual exclusive guards pattern* has been defined for this purpose. It is applicable when the rule guards check the current value of a given location that can assume disjoint values. This guarantees mutual exclusion among the guards of the conditional rules.

Consider, for example, the ASM rule in Fig. 5a. It fires the parallel execution of three conditional rules guarded by the current value of the location x. Applying the visual tree in Table 1 to this rule, we obtain the representation given in Fig. 5b. However, one can understand that the three conditions on x are mutually exclusive and, therefore, visualize the rule in a more compact way as in Fig. 5c, showing that the machine workflow follows only one of the three possible paths[6].

Example 3. The application of the mutual exclusive guards pattern to the main rule of Code 1 is shown in Fig. 6.

State pattern Often, it could be desirable to represent the machine behavior as a flow of activities along a sequence of states of control, i.e., configurations (or *modes*) in which the machine can be. Therefore, we enrich our visual notation with a special node (an ellipse) representing information about the (control) *state* in which a given rule can be executed.

Suppose the model is as shown in Fig. 7a, where `rule`i is a macro call rule that might call (directly or indirectly) the update rule `state` := `s`j. Using only the visual trees defined in Table 1, the rule would be represented by the schema

[6] Note that the pattern can be detected by a simple static analysis of the model because of the particular guard structure we consider. If we would like to handle any type of guard, detecting the pattern would require to use a logical solver.

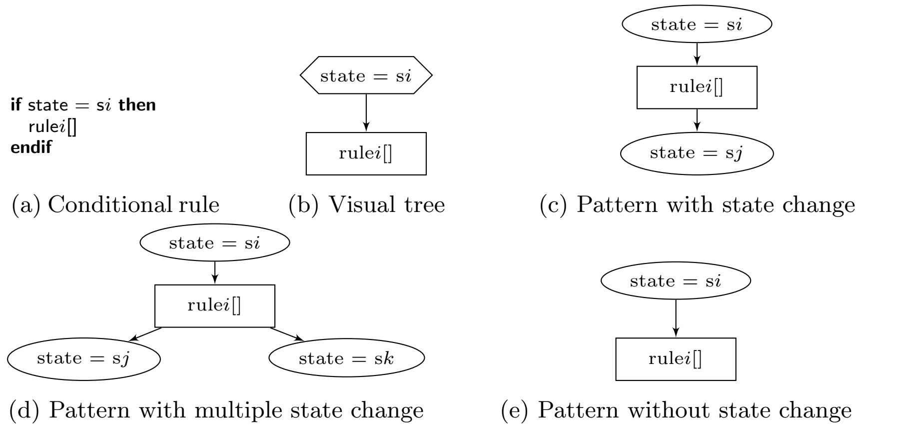


Fig. 7: Semantic pattern – State pattern

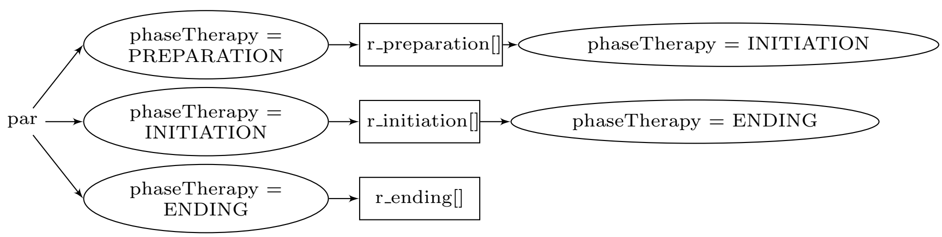
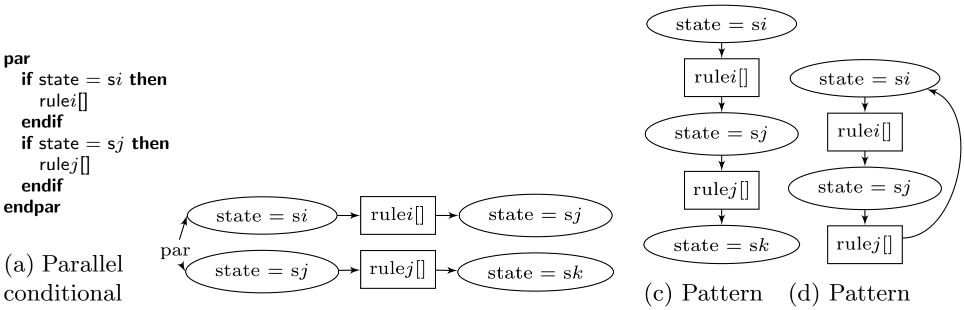


Fig. 8: Hemodialysis case study – State pattern

shown in Fig. 7b. However, supposing the modeler wants to use the function state to identify states of control, if rule$i$ changes the state from si to sj, one can build the graph as shown in Fig. 7c to explicitly represent the state change. In case rule$i$ can bring to different states (e.g., states sj and sk), the pattern is as shown in Fig. 7d. Instead, if rule rule$i$ leaves the mode unchanged, the pattern is as shown in Fig. 7e. Note that rule rule$i$ will be represented as a macro call rule, if this is not already the case.

Example 4. The application of the state pattern to the hemodialysis machine case study (see Code 1) is shown in Fig. 8.

State flow pattern The definition of the state pattern can be extended to graphically represent a flow of activities along a sequence of control states. Suppose to have the code reported in Fig. 9a and that rule$i$ contains the update rule state := sj and rule$j$ contains the update rule state := sk. By applying the state pattern explained above, one would obtain the visual graph in Fig. 9b. However, the evolution of the system from state si to sj and then to sk can be made explicit, and the graph can be rewritten as in Fig. 9c. Note that if rule rule$j$ does not update state, the flow ends with rule$j$. Instead, if rule rule$j$ updates state to si, the resulting structure is a graph as shown in Fig. 9d.



(a) Parallel conditional rules (b) State pattern (c) Pattern as tree (d) Pattern as graph

Fig. 9: Semantic pattern – State flow pattern

Fig. 10: Hemodialysis case study – State flow pattern

Example 5. The application of the state flow pattern to the hemodialysis machine case study (see Code 1) is shown in Fig. 10.

6 Tool

We have developed a prototypical tool, called `AsmetaVis`, that permits to represent the visual trees and some of the visual patterns we have presented. At the current stage of development, the tool supports the first usage we devised in Sect. 3 for our visual notation, i.e., model *visualization*, that permits to obtain the graphical representation of a specification written in AsmetaL. The tool is currently able to visualize the ASM in two modes:

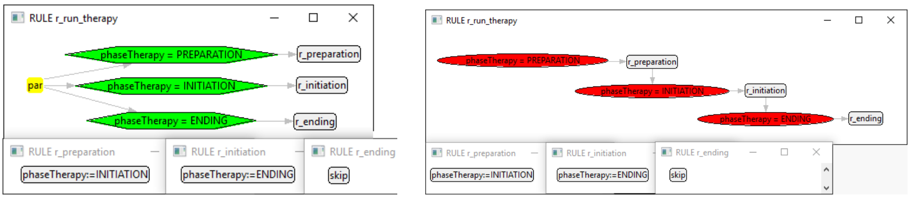
– *basic visualization* in which the ASM is visualized using only the visual trees presented in Sect. 4;
– *semantic visualization* in which information on the workflow of the model is visualized using two of the semantic patterns introduced in Sect. 5.2.

At the beginning, the tool loads the AsmetaL model and shows the graph of the main rule. A double-click on a macro call rule node causes the visualization of the corresponding macro rule graph; in this way, we provide the navigation links described in Sect. 3.

The tool is integrated in the ASMETA framework as eclipse plugin[7] and it uses Zest for implementing the visualization features[8].

Example 6. Fig. 11 shows the basic and the semantic visualizations of the model of the hemodialysis machine case study in `AsmetaVis` (see Code 1). In both cases,

---
[7] `http://asmeta.sourceforge.net/download/asmetavis.html`
[8] `https://www.eclipse.org/gef/zest/`



(a) Basic visualization (b) Semantic visualization

Fig. 11: `AsmetaVis` tool – Hemodialysis case study

Group	UQ (% correct answers)	Avg. time (sec)	SQ (% affirmative answers)
Graphical	92.3	135	100
Textual	73.0	226	7.6

Table 2: Experimental results

the main window represents the main rule and the other smaller windows depict the called macro rules.

7 Preliminary evaluation

We conducted a preliminary experiment to evaluate whether the proposed visual notation can help in understanding a model. We interviewed 15 students who attended a course on formal system modeling and verification at the University of Milan (ten lectures on ASMs), and 11 who attended a course on principles of programming languages at the University of Bergamo (six lectures on ASMs). We took the (last refined) textual model of the hemodialysis case study [2], that consists of 163 macro rules and 1880 lines of code. We gave the textual model to half of the students and its graphical representation to the other half. Then we asked them a question in order to evaluate their understanding of the model (UQ: Which are the phases of the hemodialysis treatment and in which order are they executed?). We measured the time taken for answering the question. After this experiment, we gave them also the other representation (the textual one for those having the graphical one, and vice versa) and we asked them to identify the same elements in both representations. Then we asked them a question regarding their satisfaction about the notation they used at the beginning (SQ: Are you satisfied with the notation you used at the beginning?).

Table 2 shows the results of the experiment. By UQ, we observe that the graphical notation permits to understand the model semantics better in less time than the textual notation. Regarding the level of satisfaction (SQ), all the students who used the graphical notation were satisfied and they would not have preferred using the textual one. Instead, only 7.6% of those using the textual notation were satisfied and 92.4% of them said that they would have preferred using the graphical one.

8 Related work

The need of having visualization techniques for easing the work of the modeler is felt in the formal methods community [16,24,9]. Different experiences show that the adoption of such visualization techniques makes the use of formal methods feasible also for non-experts [16], and also helps in teaching formal methods [15].

Different visualization techniques have been proposed.

Some approaches focus on the *visualization of the model*. In [12], graphical notations are used as an alternative representation of Z specifications. They also support a mechanical translation process from Z models to diagrams. They share with us the idea of using visualization in two ways.

A similar approach is proposed for VDM in [8] where the authors propose two kinds of diagrams: Entry-Structure Diagrams (ESD) modeling the system state, and Operation-State Diagrams (OSD) modeling the behavior. OSDs are similar to visual patterns and require the VDM to be in a particular style.

Other approaches, instead, try to provide a visual representation of the model execution (or *model animation*), as in [13,14,17] for the B-method. ProB [13] allows fully automatic animation of B specifications, and can be used, by means of its integrated constraint solver and model checker, to systematically check a specification for a wide range of errors, for deadlock checking, and test-case generation. In [14], B-Motion Studio, a tool that allows to create visualizations for Event-B and B models, is introduced. B-Motion Studio uses two important concepts: Controls and Observers. A control is a graphical representation of some aspects of the model, and labels, images, or buttons are used to represent informations. Observers link controls to the B-model's state and invoke the animator ProB. In [17], Event-B specifications are validated by animation with the Brama tool. The authors propose some heuristics to produce an animatable specification which exhibits the same behavior as the original specification.

We have worked in the past on the visualization of ASM behaviors [10]. In that paper, the animator was built by the user by adding, from a palette, labels (for controlled variables) and input widgets like buttons (for monitored variables). Although very powerful, that approach required a great effort in order to build the animator panel and to connect it to the model. We plan in the future to integrate the animation of behavior in `AsmetaVis`, but we would like to make the process of building animators as automatic as possible.

Our state flow pattern is a conservative generalization of the visualization for *control state machines*, which are an ASMs class with an intuitive (informally defined by examples in [5]) graphical representation in FSM-like diagrams. Our tool automatically provides a correct and precise visualization of those machines.

Other directions of model visualization concern the *use of UML notation as modeling front-end*, due also to the wide use of the UML in industry. This is, for example, the case for UML-B [22], which uses the B notation as an action and constraint language for the UML, and defines the semantics of UML entities via a translation into B. Similarly, in [18], transforming rules are given from UML models to Object-Z constructs; therefore, the semantics of UML models is directly expressed in the formal language Object-Z. The tool OZRose has been developed to automate the transforming process. Furthermore, in [20], ArchiTRIO is defined as a formal language which complements UML 2.0 concepts with a logic-based notation that allows users to state system properties, both static and dynamic, including real-time constraints.

Combined approaches have also been studied. In [19], for example, an integration of UML-B and Object-Z has been proposed to define a software development process where UML-B is used as visual modeling notation at early conceptual modeling stage, and Object-Z later when requirements are better understood.

Along this trend of combining lightweight graphical notations with formal methods, in [21], a framework has been developed for modeling and executing service-oriented applications. The framework uses the SCA (Service Component Architecture) notation to express the assembly and the architecture of service-oriented components, and the ASMs to rigorously model services behavior, interactions, orchestration, compensation, and context-awareness.

9 Conclusions

With this work we have tried to satisfy a request, felt from long time, to have a way, and a supporting tool, to graphically represent ASM models, from a structural and from a behavioral point of view. We have proposed a graphical notation for ASMs, and we have defined visual patterns that capture, in a concise way, different recurring ASM rule patterns. The representation concerns only the transition rules and not the signature of the model.

As future work, we plan to define visual trees for all the turbo rules, and identify new visual patterns. Regarding the tool, we plan to implement the second usage we devised in Sect. 3 for our visual notation, i.e., the *visual editing* that should allow a modeler to graphically specify the ASM using the visual components (visual trees and visual patterns) we have proposed. Finally, we plan to better evaluate the possible advantages of using the proposed visual notation by means of a controlled experiment.

References

1. 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.
2. P. Arcaini, S. Bonfanti, A. Gargantini, and E. Riccobene. How to assure correctness and safety of medical software: the hemodialysis machine case study. In *Proceedings of ABZ 2016*, volume 9675 of *Lecture Notes in Computer Science*. Springer International Publishing, 2016.
3. 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.
4. 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.
5. 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.

6. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.
7. B. R. Bryant, J. Gray, M. Mernik, P. J. Clarke, R. B. France, and G. Karsai. Challenges and directions in formalizing the semantics of modeling languages. *Computer Science and Information Systems*, 8(2):225–253, 2011.
8. J. Dick and J. Loubersac. Integrating structured and formal methods: A visual approach to VDM. In *Proceedings of the 3rd European Software Engineering Conference*, ESEC '91, pages 37–59, London, UK, UK, 1991. Springer-Verlag.
9. N. Dulac, T. Viguier, N. Leveson, and M.-A. Storey. On the use of visualization in formal requirements specification. In *Requirements Engineering, 2002. Proceedings. IEEE Joint International Conference on*, pages 71–80. IEEE, 2002.
10. A. Gargantini and E. Riccobene. ViBBA: A toolbox for automatic model driven animation. In *Proc. of SIMVIS 2005*, pages 101–114. SCS Publishing House, 2005.
11. A. Gargantini, E. Riccobene, and P. Scandurra. A metamodel-based language and a simulation engine for Abstract State Machines. *J. UCS*, 14(12):1949–1983, 2008.
12. S.-K. Kim and D. Carrington. Visualization of formal specifications. In *Proceedings of APSEC'99*, pages 102–109. IEEE, 1999.
13. L. Ladenberger, J. Bendisposto, and M. Leuschel. Visualising Event-B Models with B-Motion Studio. In *Formal Methods for Industrial Critical Systems*, volume 5825 of *LNCS*, pages 202–204. Springer Berlin Heidelberg, 2009.
14. M. Leuschel, J. Bendisposto, I. Dobrikov, S. Krings, and D. Plagge. *From Animation to Data Validation: The ProB Constraint Solver 10 Years On*, pages 427–446. John Wiley & Sons, Inc., 2014.
15. M. Leuschel, M. Samia, and J. Bendisposto. Easy graphical animation and formula visualisation for teaching B. In *The B Method: From Research to Teaching*, 2008.
16. T. Margaria and V. Braun. Formal methods and customized visualization: A fruitful symbiosis. In *Services and Visualization Towards User-Friendly Design*, volume 1385 of *LNCS*, pages 190–207. Springer Berlin Heidelberg, 1998.
17. A. Mashkoor, J.-P. Jacquot, J. Souquières, et al. Transformation heuristics for formal requirements validation by animation. In *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems-SafeCert 2009*, 2009.
18. H. Miao, L. Liu, and L. Li. Formalizing UML models with Object-Z. In *Formal Methods and Software Engineering*, volume 2495 of *Lecture Notes in Computer Science*, pages 523–534. Springer Berlin Heidelberg, 2002.
19. M. Najafi and H. Haghighi. An integration of UML-B and Object-Z in software development process. In *Innovations and Advances in Computer, Information, Systems Sciences, and Engineering*, volume 152 of *Lecture Notes in Electrical Engineering*, pages 633–648. Springer New York, 2013.
20. M. Pradella, M. Rossi, and D. Mandrioli. ArchiTRIO: A UML-compatible language for architectural description and its formal semantics. In *Proceedings of FORTE 2005*, pages 381–395. Springer Berlin Heidelberg, 2005.
21. E. Riccobene and P. Scandurra. A formal framework for service modeling and prototyping. *Formal Asp. Comput.*, 26(6):1077–1113, 2014.
22. C. Snook and M. Butler. UML-B: Formal modeling and design aided by UML. *ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122, Jan. 2006.
23. M. Spichkova. Design of formal languages and interfaces: "formal" does not mean "unreadable". *Emerging Research and Trends in Interactivity and the Human-Computer Interface*, pages 301–314, 2014.
24. M. Spichkova. Human factors of formal methods. *CoRR*, abs/1404.7247, 2014.