A scenario-based validation language for ASMs

A. Carioni[2] A. Gargantini[1] E. Riccobene[2] P. Scandurra[2]

[1] Dip. di Ing. Informatica e Metodi Matematici, Università di Bergamo, Italy
`angelo.gargantini@unibg.it`
[2] Dip. di Tecnologie dell'Informazione, Università di Milano, Italy
`{carioni,riccobene,scandurra}@dti.unimi.it`

Abstract. This paper presents the AValLa language, a domain-specific modelling language for scenario-based validation of ASM models, and its supporting tool, the AsmetaVvalidator. They have been developed according to the model-driven development principles as part of the asmeta(ASM mETAmodelling) toolset, a set of tools around ASMs. As a proof-of-concepts, the paper reports the results of the scenario-based validation for the well-known LIFT control case study.

1 Introduction

The success of developing complex systems depends on the use of a pertinent method for identifying the requirements on the target system and to make sure that the produced system will actually meet these requirements. Validation is intended as the process of investigating a model (intended as formal specification) with respect to its user perceptions, in order to ensure that the specification really reflects the user needs and statements about the application, and to detect faults in the specification as early as possible with limited effort. Validation should precede the application of more expensive and accurate methods, like formal requirements analysis and verification of properties, that should be applied only when a designer has enough confidence that the specification captures all informal requirements. Techniques for validation include scenarios generation, development of prototypes, animation, simulation, and also testing [28].

In [21], we defined the AsmetaL language as concrete syntax to write Abstract State Machine (ASM) models and the AsmetaS simulator to execute AsmetaL programs. In order to validate AsmetaL specifications, we here investigate the *scenario-based* approach for system validation. In this context, scenarios describe the behavior of a system from a *global* perspective by looking at the observable interactions between the system and its environment in specific situations. Scenarios are useful to ensure correct capture of informal requirements and to explore system functionalities and alternative design solutions. To make this approach effective by allowing the designer to interact with the specification, we define a language, called AValLa (ASM Validation Language), which provides suitable commands to express, at ASM model level, the interaction between a system and its environment (in the sense of UML use-cases) and the interaction between a system and an external observer who likes to play with the system model and check the system state.

AValLa has been developed according to the model-driven language engineering principles which require the abstract syntax of a language be defined in terms of an (object-oriented) model, called *metamodel*, characterizing syntax elements and their relationships. A concrete notation can be then (automatically) derived from the abstract syntax. The language semantics is given in terms of ASMs, here used as formal semantic framework to express the operational semantics of metamodel-based languages.

AValLa is supported by the AsmetaV (ASM Validator) tool to execute AValLa scenarios. Both have been developed within the asmeta(ASM mETA-modelling) tool-set [17,19,4] by exploiting the metamodelling approach.

In this paper, we first motivate in Sect. 2 our work on scenario-based system validation in relation to other similar approaches. In Sect. 3, we present our basic idea on how targeting validation in the ASM context. In Sect. 4 we present the AValLa language to build scenarios for ASM models, and we describe how AValLa has been defined following the model-driven engineering process. In Sect. 5, we provide the semantics of the AValLa constructs exploiting the ASM-based semantic framework for metamodel-based languages. Our scenario-based validator AsmetaVis presented in Sect. 6, while Sect. 7 presents a case study. Conclusions are given in Sect. 8.

2 Motivations and related work

The *scenarios* technique has been applied in different research areas and a variety of definitions, ways of use and ways of interaction with the user are given. In particular, scenarios have been used in the area of Software Engineering [33,2,32], Business-process reengineering [3], User Interface Design [9], Documentation and demonstration of software and many more. In addition, the term "script" used in Artificial Intelligence [35] and in Object-behavior Analysis [36], is very similar to the various definitions of scenarios.

Authors in [8] classify scenarios according to their use in systems development ranging from requirements analysis, user-designer communication, examples to motivate design rationale, envisionment (imagined use of a future design), software design, through to implementation, training and documentation.

The telecommunication system development is one of the main field where scenarios have been successfully applied [1]. Message Sequence Charts (MSCs) [31] is one of the most used (graphical) notation by telecommunications companies and standard bodies. MSCs can be adapted to describe embedded systems and software, although, for software, UML notations are more used. The Life Sequence Charts (LSCs) [11] extend the MSCs by providing the "clear and usable syntax and a formal semantics" MSCs lack of.

In the object-oriented community, scenarios are intended as instances of a *use case* [39] defining a goal-oriented set of interactions between external actors (i.e. parties outside the system that interact with the system) and the system under consideration. The system is treated as a *black box*, and the interactions with it, including system responses, are perceived as outside the system. A complete set of use cases specifies all different ways to use the system, and therefore defines all required behavior, bounding the scope of the system. For complex systems, the complete set of use cases might be unfeasible, and in this case it is useful to proceed in an incremental way.

The idea of using scenarios as a means for validating a specification has been extensively adopted in the past, but its application has been mostly of informal nature. [37] provides a mini tutorial explaining the concepts and process of scenario-based requirements engineering. The relationships between scenarios, specifications and prototypes are explored, and the SCRAM method (Scenario-based Requirements Analysis Method), where scenarios are used with early prototypes to elicit requirements in reaction to a preliminary design, is presented. In [26], a systematic way to analyze and validate requirements is proposed and applied to a simple PBX system. This formal-approach to scenario analysis views the user as the starting point to form scenarios and uses prototyping in order to validate the scenarios and refine the specifications. In [27], a case study is presented to show how functional requirements can be successfully decomposed and analyzed using scenarios. In [30], authors show the CineVali approach in which scenarios are formal and automatically generated by the user and by the analyst in accordance with their purposes.

The main obstacles to an effective use of scenarios for formal validation are mainly due to the non executable nature of formal models, or in the case of executable specifications, due to the lack of simulation engines and suitable tools allowing the designer to interact with the (complete or only sketched) specification in order to observe the system behavior and/or check the system states.

A method for constructing a formal description of the system from scenarios expressing the stakeholders' requirements, is presented in [25]. The authors use the Albert II formal language and scenarios are represented by MSCs. A requirements validation tool that stakeholders can use to explore different possible behaviors of the system to develop, is presented. These behaviors are automatically checked against the formal requirements specification.

In the context of ASMs, the authors in [22,5] show how SpecExplorer and its language Spec#, can be applied for scenario-oriented modelling. They describe how Spec# models can be instrumented for validation purposes by a set of instructions, which allow SpecExplorer to execute scenarios. They also describe scenarios in an algorithmic way with the ultimate goal to have a tailored notation, like MSCs, as front-end for scenarios description. Grieskamp et al. also provide an engine within the SpecExplorer tool for checking conformance of implementation against models.

Our approach is targeted to build scenarios for ASM ground models written in AsmetaL. We like to keep the algorithmic vision of building scenarios as in Spec#/SpecExplorer, since we consider this view closer to the view of programming and able to show the temporal sequence of steps between the system and its external environment. We keep the view of scenarios as paths through the use cases as inherited from the object-oriented community. Therefore, in our view a scenario will express interaction sequences of external actor actions and reactions of the machine to be analyzed.

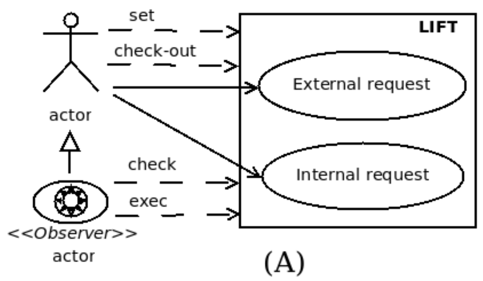
From a practical point of view, we believe that a validation activity in which the designer has a black box view of the system might be complemented by a testing activity requiring an internal view of the system. As in [10], we argue that a scenario notation should be also able to describe internal details of the system. MSCs and LSCs are very useful to describe lengthy black-box interactions between system components in a graphical way, while we want scenarios to be also able of describing, by means of a textual notation, possibly white-box interactions in a component independent way. To this regard our approach is more similar to the classical unit testing. Note that several scenario notations are derived form testing notations, for example the Use Case Maps, for which and ASM based semantics exists [24], and Use Case Trees are strongly related to the TTCN testing notation.

Therefore, in our scenario-based approach, we support two kinds of external actors: the *user*, who has a black box view of the system, and the *observer* having, instead, a gray box view. By allowing different actions to the two actors, we are able to build scenarios useful for classical validation (those including user actions and machine reactions), and scenarios useful for testing activity (those including also observer actions) requiring the inspection of the internal configurations of the machine. Therefore, our scenario-based validation approach goes behind the UML use-cases it was inspired from, and has the twofold goal of model validation and model testing.

3 Scenario-based validation of ASM models

In our approach of scenario-based validation of ASM models, we start from the idea of UML use-cases and their descriptions in terms of scenarios. A scenario is a description of external actor actions and reactions of the system. The formalization (complete or incomplete) of the system behavior is given in terms of an ASM specification. We extend the concept of actor (here called *user actor*) in UML use-cases with the concept of *observer actor* (see Fig. 1(a)). A user actor is able to interact with the system by *setting* the values of the external environment, so asking for a particular service, waits for a *step* of the machine as reaction to his/her request, and can check the values only of system outputs. A user actor has, therefore, a black box view of the system. An observer actor has the further capabilities of inspecting the internal state of the system (i.e. values of machine functions and locations), to require the *execution* of particular system (sub-)services of the machine, and to check the validity of possible invariants of a certain scenario. Therefore, an observer actor has a *gray box* view of the system under development.

Use-cases are described by a set of scenarios and each scenario represents a single path through the use case. Usually, in the UML, scenarios are depicted using sequence diagrams describing the actor(s)-system interaction, but also the



(A)

```
check floor(lift1) = 0;
// external request for UP
set existsCallFromTo(0, UP) := true;
// internal request to floor 4
set hasToDeliverAt(lift1, 4) := true;
// start moving
step
check ctlState(lift1) = MOVING;
check dir(lift1) = UP;
check existsCallFromTo(0, UP) = false;
```

(B)

Fig. 1. Use cases (A) and a scenario (B) for the `Lift` model

system components interaction to provide a certain service. We prefer to describe scenarios in an algorithmic way as interaction sequences consisting of *actions*, where each action in turn is an activity of a user actor who `set`s the environment (i.e. the values of monitored/shared functions) and `check`s for the machine outputs (i.e. the values of out functions), possibly combined with an activity of the observer actor who has the extra ability to `check` the machine (also internal) state and ask for the `exec`ution of given transition rules, and an activity of the machine which makes one `step` as reaction of the actor actions.

Fig. 1(b) shows a script relative to a scenario of the LIFT case study taken from [6] and encoded in AsmetaL. The scenario shows the interaction between a lift lying at ground floor and a user stating at the same floor and asking for the lift to go up. Once getting into, he/she asks for reaching floor 4. The observer initially checks (first **check**) that the lift is at the ground floor before the interaction takes place, and upon the machine makes a step, he/she checks that the lift is moving in the up direction and that the (external) request has been removed. Note that `existsCallFromTo(floor,dir)` specifies an external request function (reflecting the user action of pressing the `up` or `down` button outside the lift at a certain `floor`), while `hasToDeliverAt(lift,floor)` formalizes an internal request function (reflecting the user action of pressing a button inside the lift).

4 The AVALLA language

The AVALLA language has been defined as a domain-specific language (DSL) in the context of scenario-based validation of ASM models written in AsmetaL. As required for model-driven language definition, the abstract syntax of AVALLA is defined in terms of an (object-oriented) model which is usually referred in the MDE context [29] to as *Domain Definition MetaModel* (DDMM). The DDMM represents concepts and their relations of the underlying domain and allows the separation of the abstract syntax and semantics of the language constructs from their different and alternative concrete notations (textual, visual, or mixed) for various goals. A concrete syntax is usually defined by a transformation that maps the DDMM onto a "display surface" metamodel (like XML, EBNF, etc.) [29].

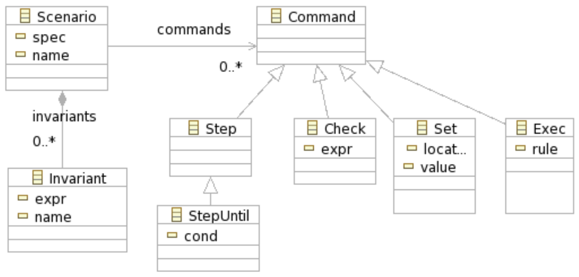


Fig. 2. AValLa metamodel (MM)

Domain Definition Metamodel (abstract syntax) The MM (Meta Model) of the AValLa is specified in EMF/Ecore [12]. Fig. 2 shows the metamodel using a UML class diagram. This metamodel captures concepts and their relations of the ASMs-based scenario modelling domain mentioned in Sect. 3.

An instance of the class `Scenario` represents a scenario of a provided ASM specification. Basically, a scenario has an attribute `name`, an attribute `spec` denoting the ASM specification to validate, and a list of target commands of type *`Command`*. Additionally, a scenario may contain the specification of some critical properties, here referred to as *scenario invariants*, that should always hold (and therefore verified) for the particular scenario – to not be confused with general *axioms* one specifies for an ASM spec as invariants over functions or domains constraining the ASM states. The composite associations between the `Scenario` class (the *whole*) and its component classes (the *parts*) `Invariant` and *`Command`* assures that each part is included in at most one `Scenario` instance.

The abstract class *`Command`* and its concrete sub-classes provide a classification of scenario commands. The `Set` command updates monitored or shared function values that are supplied by the user actor as input signals to the system. Commands `Step` and `StepUntil` represent the reaction of the system, which can execute one single ASM step and one ASM step iteratively until a specified condition becomes true. The `Check` class represents commands supplied by the user actor to inspect external property values and/or by the observer actor to further inspect internal property values in the current state of the underlying ASM. Finally, an `Exec` command executes an ASM transition rule when required by the observer actor.

Concrete Syntax A concrete syntax for AValLa has been implemented as textual notation according to the model-to-grammar mapping described in [16] and already used for deriving the AsmetaL notation [4] from the ASM Metamodel (*AsmM*) representing the abstract syntax of the ASM language as given in [4,17]. A grammar (written in JavaCC) and a parser [4] are derived from the AValLa MM to automatically parse textual scenario scripts into scenario models. Other tools, as TEF (Textual Editing Framework) [38] allowing creation of textual editors for EMF-based languages, could be exploited for the same goal.

Abstract syntax	Concrete syntax
Scenario	**scenario** name **load** spec_name Invariant* Command*
spec_name is the spec to load; invariants and commands are the script content	
Invariant	**invariant** name ':' expr ';'
expr is a boolean term made of function and domain symbols of the underlying ASM	
Command	(Set | Exec | Step | StepUntil | Check)
Set	**set** loc := value ';'
loc is a location term for a monitored function, and value is a term denoting a possible value for the underlying location	
Exec	**exec** rule ';'
rule is an ASM rule (e.g. a choose/forall rule, a conditional if, a macro call rule, ect.)	
Step	**step** ';'
StepUntil	**step until** cond ';'
cond is a boolean-valued term made of function and domain symbols of the ASM	
Check	**check** expr ';'
expr is a boolean-valued term made of function and domain symbols of the ASM	

Table 1. The AVALLA textual notation

Table 1 reports the AVALLA concrete syntax in a EBNF form, in which terminal symbols are in bold and elements in the first column represent non terminals. Examples of scenario scripts are provided in Sect. 7 for the `Lift` case study.

5 The AVALLA semantics

Currently, metamodelling environments (like Eclipse/Ecore, GME/MetaGME, AMMA/KM3, XMF-Mosaic/Xcore, etc.) allow to cope with most syntactic and transformation definition issues in the specification of a DSL, but they lack of any standard and rigorous support to provide the dynamic semantics of metamodels and metamodel-based languages, which is usually given in natural language (the most well-known example is the UML [39]). Below, we briefly present the approach we adopted to define the AVALLA semantics.

An ASM-based semantic framework for DDMMs A language has a well-defined semantics if a semantic domain S is identified and a semantic mapping M_S from the language syntax to S is provided [23]. As semantic domain S, we assume the semantic domain S_{AsmM} of the ASM language, namely the first-order logic extended with a logic for function updates and for transition rule constructors formally defined in [6]. Therefore, the semantic mapping $M_S : DDMM \rightarrow S_{AsmM}$ which associates a well-formed terminal model[3] m

[3] According to the definition in [29], a *terminal model* is a model written in the language L and *conforming* to the language metamodel.

conforming to *DDMM* with its semantic model $M_S(m)$, can be defined as

$$M_S = M_{S_{AsmM}} \circ M$$

where $M_{S_{AsmM}}$ is the semantic mapping (of the ASM language) that associates a theory conforming to the S_{AsmM} logic with a model conforming to *AsmM* (representing the abstract syntax of the ASM language), and the function

$$M : DDMM \longrightarrow AsmM$$

associating an ASM to a terminal model m. The M function ***hooks*** the semantics of a metamodel to the S_{AsmM} domain and, therefore, the problem of giving the language semantics is reduced to define the function M. Exploiting this approach, the semantics of a metamodel-based language is expressed in terms of ASM transition rules.

Language Semantics According to the approach explained above, to endow the AValLa language with a semantics we have to define a function M: $MM \longrightarrow AsmM$ associating an ASM (i.e. a model conforming to the AsmM metamodel) with a scenario model m conforming to the AValLa MM. This ASM machine is automatically induced from the elements of the source scenario model m and their typing elements in the AValLa MM, and from the original ASM to validate (which is linked to a scenario by its attribute `spec`). The resulting ASM is obtained from the original ASM in the following way.

A scenario (instance of the `Scenario` class) is mapped into the original ASM to validate (instance of the `Asm` class in AsmM), except that: monitored and shared functions are turned into controlled functions; a new 0-ary controlled function `currentRule` of `Rule` type is added to the signature to denote the current rule of the original ASM being executed; for notifying check-command's property violations, a boolean-valued 0-ary function `all_checks_OK` is added to the signature together with an axiom stating that flag `all_checks_OK` is always true; the original initial state is extended to set `currentRule` to an initial rule `r_step_0` and for setting `all_checks_OK` to true; finally, the main rule consists only into invoking the value of the `currentRule`.

Invariants and commands of the particular scenario are then taken into consideration in order to further modify the structure of the ASM (see Table 2). Scenario invariants are mapped into axioms of the final ASM. The commands list is then partitioned into groups: each group is a block of consecutive commands terminated with either a step-command, a *step-group*, or a stepUntil-command, *stepUntil-group*. For each group one or two rules are added to the final ASM according to the following directives. The i-th step-group $[C_1 \ldots C_n$ **step**$]$ is mapped into a macro rule declaration of form:

```
$r\_step\_i$ = seq                                        (1)
                $R_1 \ldots R_n$
                $old\_main[]$
                $currentRule := \ll r\_step\_i + 1 \gg$
            endseq
```

AValLa	AsmM
A `Invariant` instance	An `Axiom` instance
A `Set` instance $l{:=}v$	An `UpdateRule` instance $l{:=}v$
A `Check` instance with expression *expr*	A `ConditionalRule` with guard *expr* and then-body *all_checks_OK:=false*
A `Exec` instance for a rule R	A `Rule` instance
A step-group $C_1 \dots C_n$ **step**	A `MacroDeclaration` instance *r_step_i* as in (1)
A stepUntil-group $C_1 \dots C_n$ **step until** *cond*	Two `MacroDeclaration` instances *r_step_i* and *r_step_i_until* as in (2)

Table 2. *Mapping from* AValLa *to AsmM*

where R_i are rules generated from the corresponding commands C_i, *old_main* is the invocation of the main rule of the original ASM, and the last rule is the update for the `currentRule` variable. The i-th stepUntil-group [$C_1 \dots C_n$ **step until** *cond*] leads to two rules of form:

```
r_step_i=                    r_step_i_until =                                  (2)
    seq                          if cond then currentRule :=≪ r_step_(i+1) ≫
        R_1 ... R_n              else par
        r_step_i_until[]                 old_main[]
    endseq                               currentRule :=≪ r_step_i_until ≫
                                     endpar
                                 endif
```

where symbols R_i and *old_main* take the same meaning as above. Note that the starting rule `r_step_0` is produced by the first command group. The ASM rules R_i for the remaining commands are produced as follows. A set-command is directly mapped into an update rule. An exec-command is a request for executing a rule R of the original ASM, and therefore it is mapped into an instance of `Rule` class of *AsmM*. A check-command is mapped into a conditional rule having as guard the expression to check and as then-body an update for setting the `all_-checks_OK` flag to false (and therefore causing an axiom violation).

6 The AsmetaV validator

The AsmetaV validator is a simple Java application based on a transformation engine which automatically maps any scenario model conforming to the AValLa metamodel into an *AsmM* instance as described in Sect. 5, and on the AsmetaS simulator [18]. AsmetaV reads a scenario written by the user (see Fig. 3) in the AValLa language, builds the scenario as instance of the AValLa metamodel by means of the AValLa parser, and transforms the scenario and the AsmetaL specification which the scenario refers to, to an executable AsmM model.

Then, AsmetaV invokes the interpreter to simulate the scenario. During simulation the user can pause the simulation and observe, through a watching

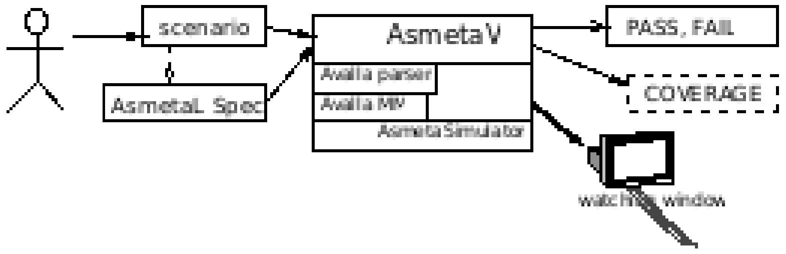


Fig. 3. AsmetaV validator

window, the current state and value of the update set produced at every step. During simulation, AsmetaV captures any check violation and if none occurs it finishes with a "PASS" verdict.

Besides a "PASS"/"FAIL" verdict, AsmetaV collects also some information about the *coverage* of the original model, obtained by running the scenario. Currently, AsmetaV keeps track of all the rules that have been called and evaluated and it prints the list of them at the end. This is useful to check if the scenario has exercised all transition rules of the original model. We plan to further refine this feature in order to monitor also which rules have actually contributed to the update set, which conditions in conditional rules have been tested true and/or false and eventually to apply the definition of coverage criteria as in [14,13].

7 The LIFT case study

To illustrate the use of the AsmetaVtool to validate an ASM specification by means of some input AValLa scenarios, we use the `Lift` example (see [6], Sect. 2.3) concerning the logic of moving n lifts between m floors.

For the sake of simplicity, we restrict to the case of one lift (`lift1`) for $m = 5$ floors. Moreover, we assume that the level of abstraction at which the `Lift` ground model is defined includes also the refinement step for the *request manipulations* (see [6], Sect. 2.3, pag. 57). In the intermediate model that we consider, the monitored function $hasToDeliverAt(L, floor)$ formalizes an internal request (reflecting requirement 1 when inside the lift a button is pressed), while an external request is modelled by the function $existsCallFromTo(floor, dir)$ (reflecting requirement 2 when on a floor outside the lift the up or down button is pressed). These two functions are shared between the lift user (who sets them, being part of the environment) and the lift control (which has to reset them in the CancelRequest macro to satisfy requirements 1 and 2). We do not consider, instead, to handle exceptional cases when the given machine either has no well-defined behavior or should be prevented from executing; we suppose therefore that the machine describes the functionality of faultless behavior.

Table 3 summarizes some of the scenarios used to validate the ASM specification of the `Lift` control. A more detailed description of these scenarios follows.

Scenario	Description	Req.	# Commands	Inv.	Coverage
s0	No requests at all	3	19	–	6/8
s1	An external request	1	24	–	7/8
s2	An external request plus an internal one	2.(a)	22	–	8/8
s3	All external buttons pushed	2.(a)	4	1	8/8

Table 3. *Some validation scenarios for the* `Lift` *control*

s0 *Description*: The lift is halted at ground floor (# 0) with no requests at all. *Requirements coverage*: 3. The lift should remain in its final position.

```
// .... setting initial state
check floor(lift1) = 0;
check ctlState(lift1) = HALTING;
check dir(lift1) = UP;
step
check floor(lift1) = 0;
check ctlState(lift1) = HALTING;
check dir(lift1) = UP;
```

# commands: set(14), check (4), step (1)
Rule coverage: 6/8 *Verdict*: PASS

s1 *Description*: The lift is halted at ground floor. A user gets into the lift and asks for reaching floor 4.
Requirements coverage: 2. The lift should move in the up direction and the external request at ground floor should be cancelled (as being satisfied). See Fig. 1(b) in Sect. 3.
# commands: set(16), check (5), step (3)
Rule coverage: 7/8 *Verdict*: FAIL (see remark below for explanation)

s2 *Description*: The lift is halted at ground floor. A user calls the lift at floor 4 and once getting into the lift he/she asks for reaching floor 2.
Requirements coverage: 2.(a) The lift satisfies the user request by reaching floor 4 and then reaching floor 2. Once satisfied, the requests must be removed:

```
// .... setting initial state
// An external request to floor 4
set existsCallFromTo(4, DOWN) := true;
// The lift goes to floor 4
step until ctlState(lift1) = HALTING
                   and floor(lift1) = 4;
// A request to floor 2
set hasToDeliverAt(lift1, 2) := true;
step
// must go down to floor 2, down dir
check dir(lift1) = DOWN;
// the request at floor 4 is cancelled
check not existsCallFromTo(4, DOWN);
// goes to floor 2
step until ctlState(lift1) = HALTING
                   and floor(lift1) = 2;
// request to floor 2 is cancelled
check not hasToDeliverAt(lift1, 2);
```

# commands: set(16), check (3), step (1), step-until (2)
Rule coverage: 8/8 *Verdict*: PASS

s3 *Description*: The lift is halted at ground floor. All external buttons (UP and DOWN) have been pushed.
Requirements coverage: 2.(a) The lift should move sequentially in the up

direction from the ground floor 0 to the top floor 4. After reaching floor 4, all UP requests should be canceled, while the DOWN ones should be still pending.
Scenario invariants: The lift should not change direction while going up: dir(lift1) != DOWN;
# commands: check (2), step-until(1), exec (1)
Rule coverage: 8/8 *Verdict*: FAIL (see remark below for explanation)

```
scenario s3
load lift.asm
invariant neverDown: dir(lift1) != DOWN;
exec //set floor requests (all external buttons UP and DOWN have been pushed)
    forall $i in {0..4} do
    par
        hasToDeliverAt(lift1, $i) := false
        if $i != top then existsCallFromTo($i, UP) := true endif
        if $i != ground then existsCallFromTo($i, DOWN) := true endif
    endpar;
//the lift goes up to floor 4, then goes down to complete existsCallFromTo(0, DOWN)
step until ctlState(lift1) = HALTING and floor(lift1) = 4;
check (forall $i in {0..4} with existsCallFromTo($i, DOWN) = true);
check (forall $i in {0..4} with existsCallFromTo($i, UP) = false);
```

Remark Scenarios s1 and s3 fail since the `Lift` specification fails to cancel an external request when it occurs at a given floor where the lift is halted, and the lift has already the same requested direction. This fault can be corrected either by constraining external requests, or by cancelling this kind of external request when the lift departs. We preferred to include a CancelRequest rule invocation within the Depart rule (see [6], Sect. 2.3), rather than to add further constraints.

8 Conclusions

This work is part of our ongoing effort in developing a set of tool around ASMs for model validation and verification. In this paper, we proposed a scenario-based approach for ASM model validation.

We have been testing our validation methodology on case studies from the embedded systems domain [20,7]. The ASMs are used as formal support to deliver formal analysis techniques for visual models developed with the UML profile for SystemC [34] – an UML extension for high-level modelling of embedded systems on chip.

In the future, we plan to integrate AsmetaV with the ATGT tool [15] in order to be able to automatically generate some scenarios by using ATGT and ask for a certain type of coverage (rule coverage, fault detection, etc.).

References

1. D. Amyot and A. Eberlein. An evaluation of scenario notations and construction approaches for telecommunication systems development. *Telecommunication Systems*, 24(1):61–94, 2003.
2. J. S. Anderson and B. Durney. Using scenarios in deficiency-driven requirements engineering. In *Proceedings of the International Symposium on Requirements Engineering*, pages 134–141. IEEE, 1993.
3. A. I. Anton, W. M. McCracken, and C. Potts. Goal decomposition and scenario analysis in business process reengineering. *Lecture Notes in Computer Science*, 811:94–104, 1994.
4. The Abstract State Machine Metamodel website. `http://asmeta.sf.net/`, 2006.
5. M. Barnett, W. Grieskamp, W. Schulte, N. Tillmann, and M. Veanes. Validating use-cases with the asmL test tool. In *3rd International Conference on Quality Software (QSIC 2003), 6-7 November 2003, Dallas, TX, USA*, pages 238–246. IEEE Computer Society, 2003.
6. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer Verlag, 2003.
7. A. Carioni, A. Gargantini, E. Riccobene, and P. Scandurra. Scenario-based Validation of Embedded Systems. In *FDL '08: Proceedings of Forum on Specification and Design Languages*, 2008.
8. J. M. Carroll. Five reasons for scenario-based design. *Interacting with Computers*, 13(1):43–60, 2000.
9. J. M. Carroll and M. B. Rosson. Getting around the task-artifact cycle: How to make claims and design by scenario. *ACM Transactions on Information Systems*, 10(2):181–212, Apr. 1992.
10. P. Chandrasekaran. How use case modeling policies have affected the success of various projects (or how to improve use case modeling). In *Addendum to the 1997 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Lanuages, and Applications*, pages 6–9, 1997.
11. W. Damm and D. Harel. LCSs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.
12. Eclipse Modeling Framework. `http://www.eclipse.org/emf/`, 2008.
13. A. Gargantini. Using model checking to generate fault detecting tests. In *International Conference on Tests And Proofs (TAP)*, number 4454 in Lecture Notes in Computer Science (LNCS), pages 189–206. Springer, 2007.
14. A. Gargantini and E. Riccobene. Asm-based testing: Coverage criteria and automatic test sequence. *J. of Universal Computer Science*, 7(11):1050–1067, 2001.
15. A. Gargantini, E. Riccobene, and S. Rinzivillo. Using spin to generate tests from ASM specifications. In *Abstract State Machines, Advances in Theory and Practice*, number 2589 in LNCS, pages 263–277. Springer, 2003.
16. A. Gargantini, E. Riccobene, and P. Scandurra. Deriving a textual notation from a metamodel: an experience on bridging Modelware and Grammarware. In *3M4MDA'06 workshop at the European Conference on MDA*, 2006.
17. A. Gargantini, E. Riccobene, and P. Scandurra. Metamodelling a Formal Method: Applying MDE to Abstract State Machines. Technical Report 97, DTI Dept., University of Milan, 2006.
18. A. Gargantini, E. Riccobene, and P. Scandurra. A metamodel-based simulator for ASMs. In A. Prinz, editor, *Proceedings of the 14th International ASM Workshop*, 2007.

19. A. Gargantini, E. Riccobene, and P. Scandurra. Ten reasons to metamodel ASMs. In *Rigorous Methods for Software Construction and Analysis - Papers Dedicated to Egon Börger on the Occasion of His 60th Birthday*, volume 5115 of LNCS. Springer, 2007.
20. A. Gargantini, E. Riccobene, and P. Scandurra. A Model-driven Validation & Verification Environment for Embedded Systems. In *Proc. of the IEEE third Symposium on Industrial Embedded Systems (SIES'08)*. IEEE, 2008.
21. A. Gargantini, E. Riccobene, and P. Scandurra. A language and a simulation engine for abstract state machines based on metamodelling. *JUCS (accepted)*, 2008.
22. W. Grieskamp, N. Tillmann, and M. Veanes. Instrumenting scenarios in a model-driven development environment. *Information & Software Technology*, 46(15):1027–1036, 2004.
23. D. Harel and B. Rumpe. Meaningful modeling: What's the semantics of "semantics"? *IEEE Computer*, 37(10):64–72, 2004.
24. J. Hassine, J. Rilling, and R. Dssouli. An ASM operational semantics for use case maps. In *13th IEEE International Conference on Requirements Engineering (RE 2005), 29 August - 2 September 2005, Paris, France*, pages 467–468. IEEE Computer Society, 2005.
25. P. Heymans and E. Dubois. Scenario-based techniques for supporting the elaboration and the validation of formal requirements. *Requir. Eng*, 3(3/4):202–218, 1998.
26. P. Hsia, J. Samuel, J. Gao, D. Kung, Y. Toyoshima, and C. Chen. Formal approach to scenario analysis. *IEEE Software*, 11(2):33–41, Mar. 1994.
27. H. Kaindl, S. Kramer, and R. Kacsich. A case study of decomposing functional requirements using scenarios. In *3rd International Conference on Requirements Engineering (ICRE '98)*, pages 156–163. IEEE Computer Society, 1998.
28. R. Kemmerer. Testing formal specifications to detect design errors. *IEEE Trans. Soft. Eng.*, 11(1):32–43, Jan. 1985.
29. I. Kurtev, J. Bézivin, F. Jouault, and P. Valduriez. Model-based DSL frameworks. In *OOPSLA Companion*, pages 602–616, 2006.
30. V. Lalioti and B. Theodoulidis. Visual scenarios for validation of requirements specification. In *SEKE'95, The 7th Int. Conference on Software Engineering and Knowledge Engineering*, pages 114–116. Knowledge Systems Institute, 1995.
31. Message sequence chart (MSC). ITU-T Recommendation Z.120, International Telecommunications Union, Nov. 1996.
32. J. Nielsen. *Scenario-Based Design*, chapter Scenarios in discount usability engineering, pages 59–83. John Wiley & Sons, 1995.
33. C. Potts, K. Takahashi, and A. I. Anton. Inquiry-based requirements analysis. *IEEE Software*, 11(2):21–32, Mar. 1994.
34. E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio. A UML 2.0 profile for SystemC: toward high-level SoC design. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 138–141. ACM Press, 2005.
35. E. Rich and K. Knight. *Artificial Intelligence*. McGraw Hill, 1991.
36. K. S. Rubin and A. Goldberg. Object behavior analysis. *Communications of the ACM*, 35(9):48–62, 1992.
37. A. Sutcliffe. Scenario-based requirements engineering. In *11th IEEE Joint Int. Conference on Requirements Engineering (RE'03)*, pages 320–329, 2003.
38. Textual Editing Framework. `http://www2.informatik.hu-berlin.de/sam/meta-tools/tef/tool.html`, 2007.
39. OMG. The Unified Modeling Language (UML), v2.1.2. `http://www.uml.org`, 2007.