The ASMETA Approach to Safety Assurance of Software Systems

Paolo Arcaini[1(✉)], Andrea Bombarda[2], Silvia Bonfanti[2], 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
{andrea.bombarda,silvia.bonfanti,angelo.gargantini,patrizia.scandurra}@unibg.it
[3] Università degli Studi di Milano, Milan, Italy
elvinia.riccobene@unimi.it

Abstract. Safety-critical systems require development methods and processes that lead to provably correct systems in order to prevent catastrophic consequences due to system failure or unsafe operation. The use of models and formal analysis techniques is highly demanded both at design-time, to guarantee safety and other desired qualities already at the early stages of the system development, and at runtime, to address requirements assurance during the system operational stage.

In this paper, we present the modeling features and analysis techniques supported by ASMETA (ASM mETAmodeling), a set of tools for the Abstract State Machines formal method. We show how the modeling and analysis approaches in ASMETA can be used during the design, development, and operation phases of the assurance process for safety-critical systems, and we illustrate the advantages of integrated use of tools as that provided by ASMETA.

1 Introduction

Failures of safety-critical systems could have potentially large and catastrophic consequences, such as human hazards or even loss of human life, damage to the environment, or economic disasters. There are many well-known examples of critical failures in application areas such as medical devices, aircraft flight control, weapons, and nuclear systems [40,41].

To assure safe operation and prevent catastrophic consequences of system failure, safety-critical systems need development methods and processes that lead to provably correct systems. Rigorous development processes require the use of formal methods, which can guarantee, thanks to their mathematical foundation, model preciseness, and properties assurance.

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

A. Raschke et al. (Eds.): Börger Festschrift, LNCS 12750, pp. 215–238, 2021.
https://doi.org/10.1007/978-3-030-76020-5_13

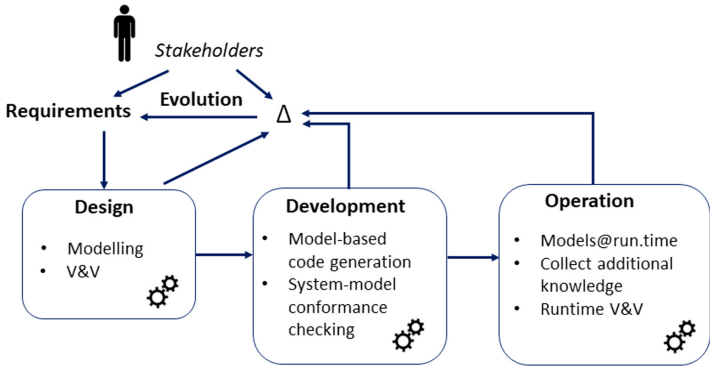


Fig. 1. Assurance process during system's life cycle

However, modern safety-critical software systems usually include physical systems and humans in the loop, as for example Cyber-Physical Systems (CPSs), and, therefore, "system safety" is not only "software safety" but may depend on the use of the software within its untrusted and unreliable environment. Reproducing and validating real usage scenarios of such systems at design- or at development- time is not always possible. Their behavior under certain circumstances cannot be completely validated without deploying them in a real environment, where all relevant uncertainties and unknowns caused by the close interactions of the system with their users and the environment can be detected and resolved [28,41]. Therefore, an important aspect of the software engineering process for safety-critical systems is providing evidence that the requirements are satisfied by the system during the *entire* system's life cycle, from inception to and throughout operation [49]. As envisioned by the Models@run.time research community, the use of models and formal analysis techniques is fundamental at design-time to guarantee reliability and desired qualities already at the early stages of the system development, but also at runtime to address requirements assurance during the system operational stage.

Providing assurances that a system complies with its requirements demands for an analysis process spanning the whole life cycle of the system. Figure 1 outlines such a process, showing the three main phases of *Design*, *Development*, and *Operation* of a system life cycle. During the system development phase, models created, validated, and verified during the design phase are eventually used to derive correct-by-construction code/artifacts of the system and/or to check that the developed system conforms to its model(s). During the operation phase, models introduced at design-time are executed in tandem with the system to perform analysis at runtime. In this assurance process, stakeholders and the system jointly derive and integrate new evidence and arguments for analysis (Δ); system requirements and models are eventually adapted according to the collected knowledge. Hence, requirements and models evolve accordingly throughout the system life cycle.

This assurance process requires the availability of formal approaches having specific characteristics in order to cover all the three phases: models should possibly be executable for high-level design validation and endowed with properties verification mechanisms; operational approaches are more adequate than denotational ones to support (automatic) code generation from models and model-based testing; state-based methods are suitable for co-simulation between model and code and for checking state conformance between model state and code state at runtime. In principle, different methods and tools can be used in the three phases; however, the integrated use of different tools around the same formal method is much more convenient than having different tools working on input models with their own languages.

This article presents, in a unified manner, the distinctive modeling features and analysis techniques supported by ASMETA (ASM mETAmodeling) [13,17], a modeling and analysis framework based on the formal method Abstract State Machines (ASMs) [26,27], and how they can be used in the three phases of the assurance process (see Fig. 1). ASMETA adopts a set of modeling languages and tools for not only specifying the executable behavior of a system but also for checking properties of interest, specifying and executing validation scenarios, generating prototype code, etc. Moreover, runtime validation and verification techniques have been recently developed as part of ASMETA to allow runtime assurance and enforcement of system safety assertions.

The remainder of this article is organized as follows. Section 2 explains the origin of the ASMETA project, recalls some basic concepts of the ASM method, and overviews the ASMETA tools in the light of the assurance process. The subsequent sections describe analysis techniques and associated tooling strategies supported by ASMETA for the safety assurance process: Sect. 3 for the design phase, Sect. 4 for the development phase, and Sect. 5 for the operation phase. Section 6 concludes the paper and outlines future research directions.

2 The ASMETA Approach

This section recalls the origin of the ASMETA project [17] and the basic concepts of the ASM method it is based on; we also overview the set of tools in the light of the assurance process.

2.1 Project Description

The ASMETA project started roughly in 2004 with the goal of overcoming the lack of tools supporting the ASMs. The formal approach had already shown to be widely used for the specification and verification of a number of software systems and in different application domains (see the *survey of the ASM research* in [27]); however, the lack of tools supporting the ASM method was perceived as a limitation, and there was skepticism regarding its use in practice.

The main goal when we started the ASMETA project, encouraged by the Egon Börger suggestion, was to develop a textual notation for encoding ASM models. We exploited the (at that time) novel *Model-driven Engineering* (MDE) approach [45] to develop an abstract syntax of a modeling language for ASMs [36] in terms of a metamodel, and to derive from it a user-facing textual notation to edit ASM models. Then, from the ASM metamodel – called *Abstract State Machine Metamodel* (AsmM) – and by exploiting the runtime support for models and model transformation facility of the open-source Eclipse-based MDE IDE EMF, ASMETA has been progressively developed till now as an Eclipse-based set of tools for ASM model editing, visualization, simulation, validation, property verification, and model-based testing [13].

In order to support a variety of analysis activities on ASM models, ASMETA integrates different external tools, such as the NuSMV model checker for performing property verification and SMT solvers to support correct model refinement verification and runtime verification. To this purpose, ASMETA mainly supports a black-box model composition strategy based on *semantic mapping* [35,39], i.e., model transformations realize semantic mappings from ASM models (edited using the textual user-facing language `AsmetaL`) to the input formalism of the target analysis tool depending on the purpose of the analysis, and then lift the analysis results back to the ASM level.

ASMETA is widely used for research purposes (also by groups different from the development teams [1,14,19,48]) and as teaching support in formal methods courses at the universities of Milan and Bergamo in Italy.

Case Studies. ASMETA has been applied to different case studies in several application domains; moreover, a wide repository of examples, many of which are benchmarks presented by Egon Börger in his dissemination work on the ASM method, are available on line[1]. Specifically, ASMETA has been applied in the context of medical devices (PillBox [20], hemodialysis device [3], amblyopia diagnosis [2], PHD Protocol [21]), software control systems (Landing Gear System [12], Automotive Software-Intensive Systems [5], Hybrid European Rail Traffic Management System [37]), cloud- [14] and service-based systems [42,43], Self-adaptive systems [15,16].

2.2 Abstract State Machines: Background Concepts

The computational model at the base of the ASMETA framework is that of the Abstract State Machines (ASMs) formal method. It was originally introduced by Yuri Gurevich as *Evolving Algebras* [38], but it was Egon Börger who renamed the approach as ASMs – viewed as an extension of Finite State Machines (FSMs) –, and disseminated it as a method for the high-level design and analysis of computing systems [26,27].

ASM *states* replace unstructured FSM control states by 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

[1] Repository https://github.com/asmeta/asmeta/tree/master/asm_examples.

the ASM concept of basic object container, and the couple (*location*, *value*) is a memory unit; an ASM state can be thus viewed as a set of abstract memories.

State transitions are performed by firing *transition rules*, which express the modification of functions interpretation from one state to the next one and, therefore, they change location values. 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. By a limited but powerful set of *rule constructors*, location updates can be combined to express other forms of machine actions as: guarded actions (`if-then`, `switch-case`), simultaneous parallel actions (`par` and `forall`), sequential actions (`seq`), non-deterministic actions (`choose`).

Functions that are not updated by rule transitions are *static*. Those updated are *dynamic*, and distinguished in *monitored* (read by the machine and modified by the environment), *controlled* (read and written by the machine), *shared* (read and written by the machine and its environment).

An ASM *computation* (or *run*) is 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 set of all transition rules invoked by a unique *main rule*, which is the starting point of the computation.

It is also possible to specify state *invariants* as first-order formulas that must be true in each computational state. A set of safety assertions can be specified as model invariants, and a model state is *safe* if state invariants are satisfied.

ASMs allow modeling different computational paradigms, from a *single* agent to distributed *multiple* agents. A *multi-agent ASM* is a family of pairs $(a, ASM(a))$, where each a of a predefined set *Agent* executes its own machine $ASM(a)$ (specifying the agent's behavior), and contributes to determine the next state by interacting synchronously or asynchronously with the other agents.

ASMs offer several advantages w.r.t. other automaton-based formalisms: (1) due to their *pseudo-code format*, they can be easily understood by practitioners and can be used for high-level programming; (2) they offer a precise system specification at any desired *level of abstraction*; (3) they are *executable models*, so they can be co-executed with system low-level implementations [43]; (4) *model refinement* is an embedded concept in the ASM formal approach; it allows for facing the complexity of system specification by starting with a high-level description of the system and then proceeding step-by-step by adding further details till a desired level of specification has been reached; each refined model must be proved to be a correct refinement of the previous one, and checking of such relation can be performed automatically [11]; (5) the concept of ASM *modularization*, i.e., an ASM without the main firing rule, facilitates model scalability and separation of concerns, so tackling the complexity of big systems specification; (6) they support synch/async multi-agent compositions, which allows for *modeling distributed and decentralized software systems* [16].

2.3 Tool-Support for Safety Assurance

Figure 2 gives an overview of the ASMETA tools by showing their use to support the different activities of the safety assurance process depicted in Fig. 1.

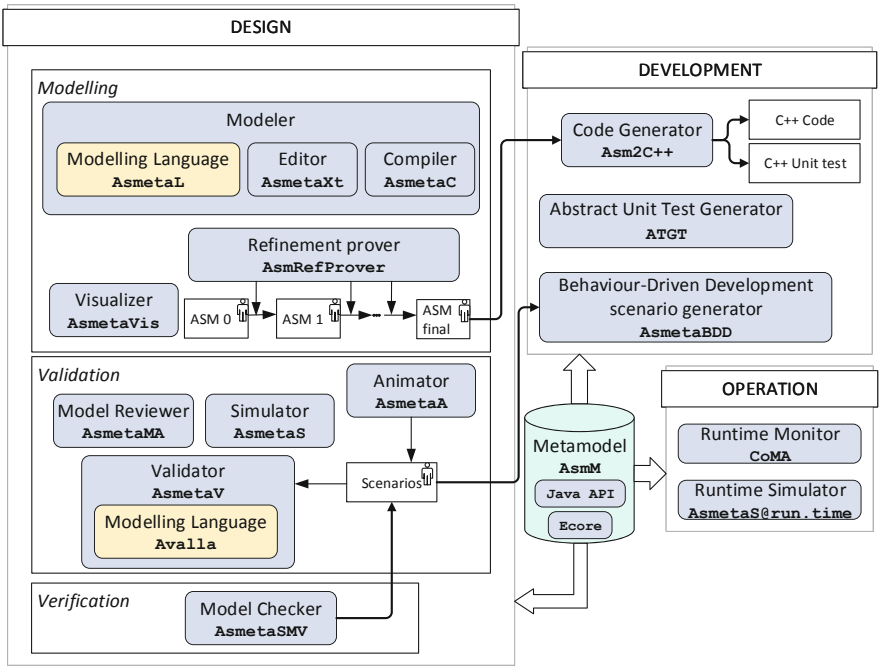


Fig. 2. ASMETA tool-set

At *design-time*, ASMETA provides a number of tools for model editing and visualization (the modeling language `AsmetaL` and its editor and compiler, plus the model visualizer `AsmetaVis` for graphical visualization of ASM models), model validation (e.g., interactive or random simulation by the simulator `AsmetaS`, animation by the animator `AsmetaA`, scenario construction and validation by the validator `AsmetaV`), and verification (e.g., static analysis by the model reviewer `AsmetaMA`, proof of temporal properties by the model checker `AsmetaSMV`, proof of correct model refinement by `AsmRefProver`).

At *development-time*, ASMETA supports automatic code and test case generation from models (the code generator `Asm2C++`, the unit test generator `ATGT`, and the acceptance test generator `AsmetaBDD` for complex system scenarios).

Finally, at *operation-time*, ASMETA supports runtime simulation (the simulator `AsmetaS@run.time`) and runtime monitoring (the tool `CoMA`).

The analysis techniques and associated tooling strategies supported by ASMETA are described in more detail in the next sections and they are applied to the *one-way traffic light* case study introduced in [27].

3 ASMETA@design-time

In order to assure the safety of software systems, system design is the first activity supported by ASMETA. During this phase, users can model the desired system

```
asm oneWayTrafficLight
import StandardLibrary
signature:
    enum domain LightUnit = {LIGHTUNIT1 | LIGHTUNIT2}
    enum domain PhaseDomain = { STOP1STOP2 | GO2STOP1 |
        STOP2STOP1 | GO1STOP2 }
    enum domain Time = {FIFTY | ONEHUNDREDTWENTY | LESS}
    dynamic controlled phase: PhaseDomain
    dynamic controlled stopLight: LightUnit -> Boolean
    dynamic controlled goLight: LightUnit -> Boolean
    dynamic monitored passed: Time
definitions:
    rule r_stop1stop2_to_go2stop1 =
        if phase=STOP1STOP2 then
            if passed = FIFTY then
                par
                    goLight(LIGHTUNIT2) :=
                        not(goLight(LIGHTUNIT2))
                    stopLight(LIGHTUNIT2) :=
                        not(stopLight(LIGHTUNIT2))
                    phase := GO2STOP1
                endpar
            endif
        endif

    rule r_go2stop1_to_stop2stop1 = ...

    rule r_stop2stop1_to_go1stop2 = ...

    rule r_go1stop2_to_stop1stop2 =
        if phase=GO1STOP2 then
            if passed = ONEHUNDREDTWENTY then
                par
                    goLight(LIGHTUNIT1) :=
                        not(goLight(LIGHTUNIT1))
                    stopLight(LIGHTUNIT1) :=
                        not(stopLight(LIGHTUNIT1))
                    phase := STOP1STOP2
                endpar
            endif
        endif

    main rule r Main =
        par
            r_stop1stop2_to_go2stop1[]
            r_go2stop1_to_stop2stop1[]
            r_stop2stop1_to_go1stop2[]
            r_go1stop2_to_stop1stop2[]
        endpar

default init s0:
    function stopLight($l in LightUnit) = true
    function goLight($l in LightUnit) = false
    function phase = STOP1STOP2
```

Fig. 3. Example of `AsmetaL` model for a one-way traffic light

using the `AsmetaL` language, exploiting its features, and refine every model which can be visualized in a graphical manner and analyzed with several verification and validation tools.

3.1 Modeling

Starting from the functional requirements, ASMETA allows the user to model the system using, if needed, model composition and refinement.

3.1.1 Modeling Language

System requirements can be modeled in ASMETA by using the `AsmetaL` language and the `AsmetaXt` editor.

Figure 3 shows the `AsmetaL` model[2] of the *one-way traffic light*: two traffic lights (`LIGHTUNIT1` and `LIGHTUNIT2`), equipped with a *Stop* (red) and a *Go* (green) light, that are controlled by a computer, which turns the lights go and stop, following a four phases cycle: for 50 s both traffic lights show *Stop*; for 120 s only `LIGHTUNIT2` shows *Go*; for 50 s both traffic lights show again the *Stop* signal; for 120 s only `LIGHTUNIT1` shows *Go*.

The model, identified by a *name* after the keyword `asm`, is structured into four sections:

- The *header*, where the signature (functions and domains) is declared, and external signature is imported (see Modularization below);
- The *body*, where transitions rules are defined (plus concrete domains and derived functions definitions, if any);
- A *main rule*, which defines the starting rule of the machine;
- The *initialization*, where a *default* initial state (among a set of) is defined.

[2] Note that `$x` denotes the variable x in the `AsmetaL` notation.

Each AsmetaL rule can be composed by using the set of *rule constructors* (see Sect. 2.2) to express the different machine action paradigms.

Modularization. ASMETA modeling supports the modularization and information-hiding mechanism, by the `module` notation. When requirements are complex or when separation of concerns is desired, users can organize the model in several ASM modules and join them, by using the `import` statement, into a single main one (also defined as *machine*), declared as `asm`, which imports the others and may access to functions, rules, and domains declared within the sub-modules. Every ASM module contains definitions of domains, functions, invariants, and rules, while the ASM machine is a module that additionally contains an initial state and the main rule representing the starting point of the execution.

3.1.2 Refinement

The modeling process of an ASM is usually based on model refinement [25]: the designer starts with a high-level description of the system and proceeds through a sequence of more detailed models each introducing, step-by-step, design decisions and implementation details. At each refinement level, a model must be proved to be a correct refinement of the more abstract one.

ASMETA supports a special case of *1−n refinement*, consisting in adding functions and rules in a way that one step in the ASM at a higher level can be performed by several steps in the refined model. We consider the refinement *correct* if any behavior (i.e., run or sequence of states) in the refined model can be mapped to a run in the abstract model.

To automatically prove the correctness of the model refinement process, users can exploit the `AsmRefProver` tool [11], which is based on a Satisfiability Modulo Theories (SMT) solver. With the execution of this software, one can specify two refinement levels and ensure that an ASM specification ASM_i is a correct refinement of a more abstract one ASM_{i-1}. Then, `AsmRefProver` confirms whether the refinement is correctly performed with two different outputs: `Initial states are conformant` and `Generic step is conformant`.

Figure 4 shows a refinement of the one-way traffic light model (see Fig. 3) in which pulsing lights (`rPulse` and `gPulse`) are introduced and a different management method for the time is used, based on a `timer` function mapping each phase to a timer duration. Thus, the behavior of the system modeled in Fig. 3 is preserved and expanded during the refinement process.

Modeling by refinement allows adding to the model requirements of increasing complexity only when the developer has gained enough confidence in the basic behaviors of the modeled system. This can be done by alternating modeling and testing activities, as presented in [21], with different refinement levels.

3.1.3 Visualization

Model visualization is a good means for people to communicate and to get a common understanding, especially when model comprehension can be threatened by the model size. ASMETA supports model visualization by a visual notation

```
asm oneWayTrafficLight_refined

import StandardLibrary

signature:
    enum domain LightUnit = {LIGHTUNIT1 | LIGHTUNIT2}
    enum domain PhaseDomain = { STOP1STOP2 | GO2STOP1
        | STOP2STOP1 | GO1STOP2 | STOP1STOP2CHANGING
        | GO2STOP1CHANGING | STOP2STOP1CHANGING
        | GO1STOP2CHANGING }
    dynamic controlled phase: PhaseDomain
    dynamic controlled stopLight: LightUnit -> Boolean
    dynamic controlled goLight: LightUnit -> Boolean
    static timer: PhaseDomain -> Integer
    dynamic monitored passed: Integer -> Boolean
    dynamic controlled rPulse: LightUnit -> Boolean
    dynamic controlled gPulse: LightUnit -> Boolean

definitions:
    function timer($p in PhaseDomain) = switch($p)
        case STOP1STOP2 : 50
        case GO2STOP1 : 120
        case STOP2STOP1 : 50
        case GO1STOP2 : 120
    endswitch

    rule r_switchToStop1 =
        par
            r_emit[rPulse(LIGHTUNIT1)]
            r_emit[gPulse(LIGHTUNIT1)]
        endpar

    rule r_switchToGo2 = ...
    rule r_switchToStop2 = ...
    rule r_switchToGo1 = ...

    rule r_stop1stop2_to_stop1stop2changing =
        if(phase=STOP1STOP2) then
            if(passed(timer(STOP1STOP2))) then
                par
                    r_switchToGo2[]
                    phase:=STOP1STOP2CHANGING
                endpar
            endif
        endif

    rule r_go2stop1_to_go2stop1changing = ...
    rule r_stop2stop1_to_stop2stop1changing = ...
    rule r_go1stop2_to_go1stop2changing = ...
    macro rule r_switch($l in Boolean) = $l := not($l)
    macro rule r_emit($pulse in Boolean) = $pulse := true

    rule r_pulses =
        forall $l in LightUnit with true do
            par
                if(gPulse($l)) then
                    par
                        r_switch[goLight($l)]
                        gPulse($l) := false
                    endpar
                endif
                if(rPulse($l)) then
                    par
                        r_switch[stopLight($l)]
                        rPulse($l) := false
                    endpar
                endif
            endpar

    macro rule r_changeState =
        par
            if(phase=STOP1STOP2CHANGING) then
                phase := GO2STOP1
            endif
            if(phase=GO2STOP1CHANGING) then ... endif
            if(phase=STOP2STOP1CHANGING) then ... endif
            if(phase=GO1STOP2CHANGING) then ... endif
        endpar

    main rule r_Main =
        par
            r_stop1stop2_to_stop1stop2changing[]
            r_go2stop1_to_go2stop1changing[]
            r_stop2stop1_to_stop2stop1changing[]
            r_go1stop2_to_go1stop2changing[]
            r_pulses[]
            r_changeState[]
        endpar

default init s0:
    function stopLight($l in LightUnit) = true
    function goLight($l in LightUnit) = false
    function phase = STOP1STOP2
    function rPulse($l in LightUnit) = false
    function gPulse($l in LightUnit) = false
```

Fig. 4. Example of a refined `AsmetaL` model for a one-way traffic light

defined in terms of a set of construction rules and schema that give a graphical representation of an ASM and its rules [4]. 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. The `AsmetaVis` tool supports two types of visualization: *basic visualization*, which represents the syntactic structure of the model and returns a visual tree obtained by recursively visiting the ASM rules; *semantic visualization*, which introduces visual patterns that permit to capture some behavioral information as control states. An example of semantic visualization of the one-way traffic light case study (see Fig. 3) is shown in Fig. 5: it displays how the four macro rules in the model change the phase of the system.

3.2 Validation and Verification

Once the `AsmetaL` model is available, the user can perform validation and verification activities.

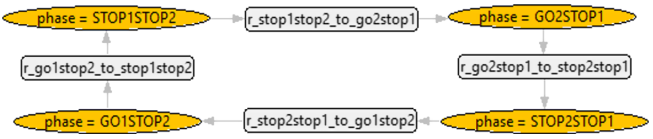


Fig. 5. `AsmetaVis` semantic visualization

```
Insert a boolean constant for passed(50):
true
<State 0 (monitored)>
passed(50)=true
</State 0 (monitored)>
<UpdateSet – 0>
goLight(lightUnit2)=true
phase=GO2STOP1
stopLight(lightUnit2)=false
</UpdateSet>
<State 1 (controlled)>
LightUnit={lightUnit1,lightUnit2}
goLight(lightUnit2)=true
phase=GO2STOP1
stopLight(lightUnit2)=false
</State 1 (controlled)>
Insert a boolean constant for passed(120):
false
<State 1 (monitored)>
passed(120)=false
</State 1 (monitored)>
<UpdateSet – 1>
</UpdateSet>
<State 2 (controlled)>
LightUnit={lightUnit1,lightUnit2}
goLight(lightUnit2)=true
phase=GO2STOP1
stopLight(lightUnit2)=false
</State 2 (controlled)>
Insert a boolean constant for passed(120):
true
<State 2 (monitored)>
passed(120)=true
</State 2 (monitored)>
<UpdateSet – 2>
goLight(lightUnit2)=false
phase=STOP2STOP1
stopLight(lightUnit2)=true
</UpdateSet>
<State 3 (controlled)>
LightUnit={lightUnit1,lightUnit2}
goLight(lightUnit2)=false
phase=STOP2STOP1
stopLight(lightUnit2)=true
</State 3 (controlled)>
Insert a boolean constant for passed(50):
```

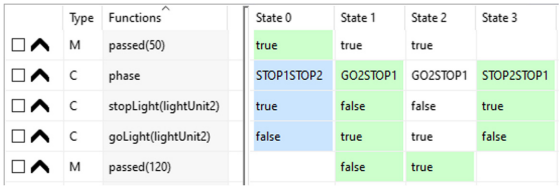
Fig. 6. Simulation of one-way traffic light using `AsmetaS`

3.2.1 Simulation

This is the first validation activity usually performed to check the `AsmetaL` model behavior during its development and it is supported by the `AsmetaS` tool [13]. Given a model, at every step, the simulator builds the update set according to the theoretical definitions given in [27] to construct the model run. The simulator supports two types of simulation: *random* and *interactive*. In random mode, the simulator automatically assigns values to monitored functions choosing them from their codomains. In interactive mode, instead, the user inserts the value of monitored functions and, in case of input errors, a message is shown inviting the user to insert again the function value. In case of invariant violation or inconsistent updates, a message is shown in the console and the simulation is interrupted. In Fig. 6, we show the result of the simulation for the one-way traffic light `AsmetaL` model (see Fig. 3). When the desired time is passed, 50 or 120 s, the phase of the system changes.

3.2.2 Animation

The main disadvantage of the simulator is that it is textual, and this makes sometimes difficult to follow the computation of the model. For this reason, ASMETA has a model animator, `AsmetaA` [22], which provides the user with complete information about all the state locations, and uses colors, tables, and figures over simple text to convey information about states and their evolution. The animator helps the user follow the model computation and understand how the model state changes at every step.



	Type	Functions	State 0	State 1	State 2	State 3
	M	passed(50)	true	true	true	
	C	phase	STOP1STOP2	GO2STOP1	GO2STOP1	STOP2STOP1
	C	stopLight(lightUnit2)	true	false	false	true
	C	goLight(lightUnit2)	false	true	true	false
	M	passed(120)		false	true	

Fig. 7. Animation of one-way traffic light using `AsmetaA`

Similarly to the simulator, the animator supports *random* and *interactive* animation. In the interactive animation, the insertion of input functions is achieved through different dialog boxes depending on the type of function to be inserted (e.g., in case of a Boolean function, the box has two buttons: one if the value is true and one if the value is false). If the function value is not in its codomain, the animator keeps asking until an accepted value is inserted. In random animation, the monitored function values are automatically assigned. With complex models, running one random step each time is tedious; for this reason, the user can also specify the number of steps to be performed and the tool performs the random simulation accordingly. In case of invariant violation, a message is shown in a dedicated text box and the animation is interrupted (as it also happens in case of inconsistent updates). Once the user has animated the model, the tool allows exporting the model run as a scenario (see Sect. 3.2.3), so that it can be re-executed whenever desired. Figure 7 shows the animation of the one-way traffic light model using the same input sequence of the simulator. The result is the same, but the tabular view makes it easier to follow the state evolution.

3.2.3 Scenario-Based Simulation

`AsmetaS` and `AsmetaA` tools require that the user executes the `AsmetaL` model step by step, each time the model has to be validated. Instead, in scenario-based simulation, the user writes a *scenario*, a description of external actor actions and reactions of the system [29], that can be executed whenever needed to check the model behavior. Scenarios are written in the `Avalla` language and executed using the `AsmetaV` tool. Each scenario is identified by its name and must `load` the ASM to be tested. Then, the user may specify different commands depending on the operation to be performed. The `set` command updates monitored or shared function values that are supplied by the user as input signals to the system. Commands `step` and `step until` represent the reaction of the system, which can execute one single ASM step and one ASM step iteratively until a specified condition becomes true. Then, the `check` command is used to inspect property values in the current state of the underlying ASM. Figure 8 shows an example of `Avalla` scenario for the one-way traffic light case study. The scenario reproduces the first two steps of the cycle: when 50 s are over, the second traffic light changes from *Stop* to *Go*; and only when 120 s are passed, the two traffic lights show *Stop* signal.

To simulate scenarios, `AsmetaV` invokes the simulator. During the simulation, `AsmetaV` captures any check violation and, if none occurs, it finishes with

```
scenario scenario1
load oneWayTrafficLight.asm

set passed(50) := true;
step
check phase = GO2STOP1;
check goLight(lightUnit2) = true;
check goLight(lightUnit1) = false;

set passed(120) := false;
step
check phase = GO2STOP1;
check goLight(lightUnit2) = true;
check goLight(lightUnit1) = false;

set passed(120) := true;
step
check phase = STOP2STOP1;
check goLight(lightUnit2) = false;
check goLight(lightUnit1) = false;
```

Fig. 8. Example of `Avalla` scenario for the one-way traffic light case study

```
</State 1 (controlled)>
check succeeded: phase = GO2STOP1
check succeeded: stopLight(lightUnit2) = false
check succeeded: goLight(lightUnit2) = true
<UpdateSet − 1>
```

```
</State 1 (controlled)>
check succeeded: phase = GO2STOP1
CHECK FAILED: stopLight(lightUnit2) = true at step 1
check succeeded: goLight(lightUnit2) = true
<UpdateSet − 1>
```

Fig. 9. `AsmetaV` output of one-way traffic light

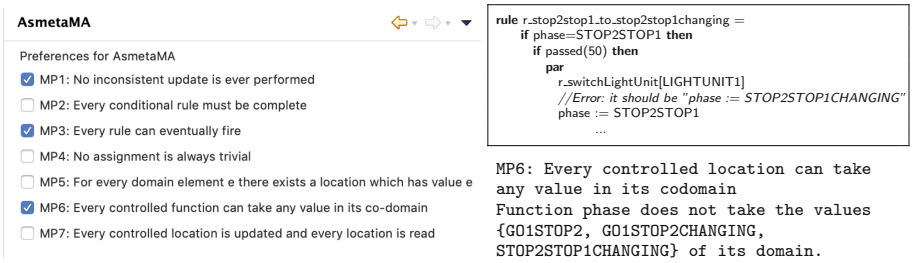
a "PASS" verdict ("FAIL" otherwise). Moreover, the tool collects information about the coverage of the `AsmetaL` model, in particular, it keeps track of all the rules that have been called and evaluated, and it lists them at the end. Figure 9 shows the output of the validator upon executing the scenario in Fig. 8: in the first column, all the functions assume the expected value, while in the second column a check is failed because the function had a different value.

The user can exploit modularization also during scenario building. Indeed, it is possible to define `blocks`, i.e., sequences of `set`, `step`, and `check`, that can be recalled using the `execblock` when writing other scenarios that foresee the same sequence of `Avalla` commands.

3.2.4 Model Reviewing

When writing a formal model, a developer could introduce some errors that are not related to a wrong specification of the requirements but are just due to carelessness, forgetfulness, or limited knowledge of the formal method. For example, a developer could use a wrong function name, or could forget to properly guard an update, and so on. An error that is commonly done in ASM development is due to its computational model, where all possible updates are applied in parallel: if a location is simultaneously updated to two different values, this is known as *inconsistent update* [26], and it is considered as an error in ASMs. Such kind of error occurs quite frequently (especially in complex models) because the developer does not properly guard all the updates. Other types of errors done using ASMs are *overspecifying* the model, i.e., adding model elements that are not needed, or writing rules that can never be triggered.

All these types of errors can be captured automatically by doing a static analysis of the model. This is the aim of the `AsmetaMA` tool [7], which performs *automatic* review of ASM models. The tool checks the presence of seven types of errors by using suitable *meta-properties* specified in CTL and verified using the model checker `AsmetaSMV` (see Sect. 3.2.5). Figure 10a shows the selection of the seven meta-properties in `AsmetaMA`. For example, MP1 checks the presence of inconsis-



(a) Meta-properties (b) Violation of MP6

Fig. 10. `AsmetaMA`

tent updates, and MP3 checks whether there are rules that can never be triggered. Figure 10b shows an example of a violation that can be found with the model review. It is an error that we discovered using `AsmetaMA` when writing the model of the traffic light; according to the requirements, when the `phase` is `STOP2STOP1` and 50 time units are passed, the `phase` should become `STOP2STOP1CHANGING` in the next state; however, we wrongly typed the value as `STOP2STOP1`. Such error was discovered by MP6 that checks if there are possible values that are never assumed by a location: the violation of MP6 allowed us to reveal our mistake.

3.2.5 Model Checking

ASMETA provides classical model checking support by the tool `AsmetaSMV` [6]. The tool translates an ASM model into a model of the symbolic model checker NuSMV [30], which is used to perform the verification. Being NuSMV a finite state model checker, the only limitation of `AsmetaSMV` is on the finiteness of the number of ASM states: only finite domains can be used, and the *extend* rule (which adds elements to a domain) is not supported.

When using `AsmetaSMV`, the NuSMV tool is transparent to the user who can specify, directly in the ASM model, Computation Tree Logic (CTL) and Linear Temporal Logic (LTL) properties defined over the ASM signature. Moreover, also the output of the model checker is pretty-printed in terms of elements of the ASM signature. Figure 11a shows CTL and LTL properties specified for the traffic light case study.

The CTL property, for example, checks that if the second traffic light shows the stop light, it will show the go light in the future.

In order to better understand the verification results, the tool allows to simulate the returned counterexample. To this aim, a translator is provided that translates a counterexample into an `Avalla` scenario (see Sect. 3.2.3). Figure 11b shows the counterexample of the violation of the CTL property shown in Fig. 11a (in a faulty version of the ASM model); the corresponding `Avalla` scenario is reported in Fig. 11c.

`AsmetaSMV` has been used in several case studies to verify the functional correctness of the specified system. `AsmetaSMV` is also used as a back-end tool for other activities supported in ASMETA, e.g., model review (see Sect. 3.2.4).

```
CTLSPEC ag(stopLight(LIGHTUNIT2) implies ef(goLight(LIGHTUNIT2)))
LTLSPEC g(phase=STOP1STOP2 implies x(phase=GO2STOP1 or phase=STOP1STOP2))
```

(a) Specification of temporal properties in the `AsmetaL` model

```
-- specification AG (stopLight(LIGHTUNIT2) ->
                     EF goLight(LIGHTUNIT2)) is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
  -> State: 7.1 <-
    stopLight(LIGHTUNIT2) = true
    goLight(LIGHTUNIT2) = false
    rPulse(LIGHTUNIT2) = false
    passed(50) = false
    phase = STOP1STOP2
    passed(120) = false
    gPulse(LIGHTUNIT2) = false
  -> State: 7.2 <-
    passed(50) = true
  -> State: 7.3 <-
    rPulse(LIGHTUNIT2) = true
    passed(50) = false
    phase = STOP1STOP2CHANGING
    gPulse(LIGHTUNIT2) = true
  -> State: 7.4 <-
    stopLight(LIGHTUNIT2) = false
    goLight(LIGHTUNIT2) = true
    rPulse(LIGHTUNIT2) = false
    phase = GO2STOP1
    passed(120) = true
    gPulse(LIGHTUNIT2) = false
  -> State: 7.5 <-
    rPulse(LIGHTUNIT2) = true
    phase = GO2STOP1CHANGING
    passed(120) = false
    gPulse(LIGHTUNIT2) = true
  -> State: 7.6 <-
    stopLight(LIGHTUNIT2) = true
    goLight(LIGHTUNIT2) = false
    rPulse(LIGHTUNIT2) = false
    phase = STOP2STOP1
    gPulse(LIGHTUNIT2) = false
```

(b) Counterexample in `AsmetaSMV`

```
scenario oneWayTrafficLight_refined.test
load oneWayTrafficLight_refined.asm

check stopLight(LIGHTUNIT2) = true;
check goLight(LIGHTUNIT2) = false;
check rPulse(LIGHTUNIT2) = false;
check phase = STOP1STOP2;
check gPulse(LIGHTUNIT2) = false;

set passed(50) := false; set passed(120) := false;
step

set passed(50) := true;
step
check rPulse(LIGHTUNIT2) = true;
check phase = STOP1STOP2CHANGING;
check gPulse(LIGHTUNIT2) = true;

set passed(50) := false;
step
check stopLight(LIGHTUNIT2) = false;
check goLight(LIGHTUNIT2) = true; check
rPulse(LIGHTUNIT2) = false;
check phase = GO2STOP1;
check gPulse(LIGHTUNIT2) = false;

set passed(120) = true;
step
check rPulse(LIGHTUNIT2) = true;
check phase = GO2STOP1CHANGING;
check gPulse(LIGHTUNIT2) = true;

set passed(120) := false;
step
check stopLight(LIGHTUNIT2) = true;
check goLight(LIGHTUNIT2) = false;
check rPulse(LIGHTUNIT2) = false;
check phase = STOP2STOP1;
check gPulse(LIGHTUNIT2) = false;
```

(c) Executable counterexample in `Avalla`

Fig. 11. `AsmetaSMV`

4 ASMETA@development time

Once the `AsmetaL` model is available, the user can automatically generate abstract tests, C++ code, and C++ unit tests. Moreover, Behavior-Driven Development scenarios in C++ can be generated from `Avalla` scenarios.

4.1 Model-Based Test Generation

Model-based testing [46] is a popular testing approach in which formal models are used for testing purposes, in particular test generation. Indeed, the model is an abstract representation of the System Under Test (SUT), from which it is possible to generate both the test inputs and the expected output (so, tackling the *oracle problem* of software testing [18]). In offline test generation, *abstract tests* are generated from the model, and then these are translated into *concrete tests* for the SUT. Coverage criteria over the model are used to define the test goals. A typical approach for generating tests achieving these goals is to use model checkers [32]: a test goal is translated into a suitable temporal property

(called *trap property*), whose counterexample (if any) is the test covering the test goal.

In ASMETA, the `ATGT` tool [34] performs model-based test generation using both model checkers SPIN and NuSMV. The generation is guided by coverage criteria defined or adapted for ASMs [33], such as rule coverage, parallel rule coverage, MCDC, etc. For example, the *rule coverage* criterion requires that for every transition rule r_i there exists at least one state in a test in which r_i fires, and another state in a test in which r_i does not fire. The abstract tests generated with `ATGT` can be later translated into concrete test cases for the implementation, as described in Sect. 4.3.

4.2 Model-Based Code Generation

According to best practices of model-driven engineering, the implementation of a system should be obtained from its model through a systematic model-to-code transformation. Thanks to `Asm2C++`, given an `AsmetaL` model, the C++ code is automatically generated [24]. This is done through a series of steps: the `AsmetaL` model is parsed and its (internal) representation in terms of Java objects as an instance of the ASMETA metamodel (AsmM) is built; then, a *model-to-text* transformation, implemented in Xtext, is applied to translate the model into C++ code. The generated code is composed of two files: header (.h) and source (.cpp). The header file contains the interface of the source file and the translation of ASM domains declaration and definition, functions and rules declaration. The rules implementation, the functions/domains initialization, and the definitions of the functions are contained in the source file. The translation of the one way traffic light case study in C++ is shown in Fig. 12.

Since an ASM run step consists in the execution of the main rule and the update of the locations, in C++ the ASM step has been implemented by two methods: `mainRule()` and `fireUpdateSet()`. The former corresponds to the translation of the ASM main rule, while the latter updates the locations to the next state values. Moreover, we have addressed two semantic ASM concepts that do not have a direct implementation in C++: parallel execution and nondeterminism. More details on their implementation in C++ and the translation of ASM rules to corresponding C++ instructions can be found in [24].

Given the translation of an `AsmetaL` model in C++ code, it is easy to adapt the code generation process for a specific platform. We have chosen Arduino since it supports C++, it is cheap and it is easily accessible. After C++ code generation, three new steps are required: HW configuration and integration, ASM runner generation, and merging of all generated files. HW configuration contains the mapping between ASM functions and Arduino input/output, and other specific hardware settings. A first draft is automatically generated, and then the user links monitored and out functions to physical hardware pins. The ASM runner automatically generates a .ino file which contains the `loop()` function to run ASM on Arduino. The `loop()` function iteratively executes the following functions: `getInputs()`—reads the data from the input devices like sensors; `mainRule()`—contains the behavior described in the `AsmetaL`

```
#ifndef  ONEWAYTRAFFICLIGHT_H
#define  ONEWAYTRAFFICLIGHT_H

#include <set>
using namespace std;


/* DOMAIN DEFINITIONS */
namespace oneWayTrafficLightnamespace{
    class LightUnit;
 enum PhaseDomain {STOP1STOP2, GO2STOP1,
       STOP2STOP1, GO1STOP2};
}

using namespace
      oneWayTrafficLightnamespace;


class oneWayTrafficLightnamespace::
      LightUnit{
 public:
 static std::set<LightUnit*> elems;
 LightUnit(){elems.insert(this);}
};


class oneWayTrafficLight {



/* DOMAIN CONTAINERS */
const set<PhaseDomain> PhaseDomain_elems;

public:


 /* FUNCTIONS */
 PhaseDomain phase[2];
 std::map<LightUnit*, bool> stopLight[2];
 std::map<LightUnit*, bool> goLight[2];
 static int timer (PhaseDomain
       param0_timer);
 static LightUnit* lightUnit1;
 static LightUnit* lightUnit2;
 std::map<int, bool> passed;


 /* RULE DEFINITION */
 void r_switch (bool _l);
 void r_switchToGo2();
 void r_switchToStop2();
 void r_switchToGo1();
 void r_switchToStop1();
 void r_stop1stop2_to_go2stop1();
 void r_go2stop1_to_stop2stop1();
 void r_stop2stop1_to_go1stop2();
 void r_go1stop2_to_stop1stop2();
 void r_Main();


 oneWayTrafficLight();


 void initControlledWithMonitored();
 void getInputs();
 void setOutputs();
 void fireUpdateSet();


};


#endif
```

```
#include "oneWayTrafficLight.h"
using namespace oneWayTrafficLightnamespace;
/* Conversion of ASM rules in C++ methods */
void oneWayTrafficLight::r_switch (bool _l){
 _l = ! (_l);}
void oneWayTrafficLight::r_switchToGo2(){
 {   r_switch(goLight[0][lightUnit2]);
  r_switch(stopLight[0][lightUnit2]);}}
void oneWayTrafficLight::r_switchToStop2(){
 { r_switch(goLight[0][lightUnit2]);
  r_switch(stopLight[0][lightUnit2]);}}
void oneWayTrafficLight::r_switchToGo1(){
 { r_switch(goLight[0][lightUnit1]);
  r_switch(stopLight[0][lightUnit1]);}}
void oneWayTrafficLight::r_switchToStop1(){
 { r_switch(goLight[0][lightUnit1]);
  r_switch(stopLight[0][lightUnit1]);}}
void oneWayTrafficLight::r_stop1stop2_to_go2stop1(){
 if ((phase[0] == STOP1STOP2)){
  if (passed[timer(STOP1STOP2)]){
   {  r_switchToGo2();
    phase[1] = GO2STOP1;}}}}
void oneWayTrafficLight::r_go2stop1_to_stop2stop1(){
 if ((phase[0] == GO2STOP1)){
  if (passed[timer(GO2STOP1)]){
   {  r_switchToStop2();
    phase[1] = STOP2STOP1;}}}}
void oneWayTrafficLight::r_stop2stop1_to_go1stop2(){
 if ((phase[0] == STOP2STOP1)){
  if (passed[timer(STOP2STOP1)]){
   {  r_switchToGo1();
    phase[1] = GO1STOP2;}}}}
void oneWayTrafficLight::r_go1stop2_to_stop1stop2(){
 if ((phase[0] == GO1STOP2)){
  if (passed[timer(GO1STOP2)]){
   {  r_switchToStop1();
    phase[1] = STOP1STOP2;}}}}
void oneWayTrafficLight::r_Main(){
 {  r_stop1stop2_to_go2stop1();
  r_go2stop1_to_stop2stop1();
  r_stop2stop1_to_go1stop2();
  r_go1stop2_to_stop1stop2();}
/* Static function definition */
int oneWayTrafficLight::timer(PhaseDomain _p){return [&](){
 if(_p==STOP1STOP2)
  return 50;
 else if(_p==GO2STOP1)
  return 120;
 else if(_p==STOP2STOP1)
  return 50;
 else if(_p==GO1STOP2)
  return 120; }();}
/* Function and domain initialization */
oneWayTrafficLight::oneWayTrafficLight(){
    //Static domain initialization
    PhaseDomain_elems:{STOP1STOP2,GO2STOP1,STOP2STOP1,GO1STOP2;};
    /* Init static functions Abstract domain */
    lightUnit1 = new LightUnit;
    lightUnit2 = new LightUnit;
    /* Function initialization */
    for(const auto& _l : LightUnit::elems){
        stopLight[0].insert({_l,true});
        stopLight[1].insert({_l,true}); }
    for(const auto& _l : LightUnit::elems){
        goLight[0].insert({_l,false});
        goLight[1].insert({_l,false}); }
        phase[0] = phase[1] = STOP1STOP2;}
void oneWayTrafficLight::initControlledWithMonitored(){}
/* Apply the update set */
void oneWayTrafficLight::fireUpdateSet(){}
/* init static functions and elements of abstract domains */
std::set< LightUnit*>LightUnit::elems;
LightUnit*oneWayTrafficLight::lightUnit1;
LightUnit*oneWayTrafficLight::lightUnit2;
```

Fig. 12. oneWayTrafficLight.h and oneWayTrafficLight.cpp

model; `fireUpdateSet()`—updates the state at the end of each loop; and `setOutputs()`—sets the output values like the current state of light-emitting diode (LED). The merging step takes care of merging all files.

```
BOOST_AUTO_TEST_SUITE( TestoneWayTrafficLight )
BOOST_AUTO_TEST_CASE( my_test_0 ){
 // instance of the SUT
 oneWayTrafficLight onewaytrafficlight;
 // state
 // set monitored variables
 onewaytrafficlight.passed[50]=false;
 ...
 BOOST_CHECK(onewaytrafficlight.phase[0]==STOP1STOP2);
 // call main rule
 onewaytrafficlight.r_Main();
 onewaytrafficlight.fireUpdateSet();
 ...
}
...
```

Fig. 13. C++ unit test

4.3 Unit Test Generation

If the C++ code is available (automatically generated or not) and the user wants to test it, C++ unit tests can be automatically generated given the `AsmetaL` model [23]. Unit tests are generated in two different ways. The first approach consists in running randomly the `AsmetaS` simulator for a given number of steps as requested by the tester, then the generated state sequence is translated into a C++ unit test. The second approach, instead, translates the abstract tests generated with `ATGT` (see Sect. 4.1) in C++ unit tests. In both cases, the C++ unit tests are written using the Boost Test C++ library.

A test suite is defined by using the `BOOST_AUTO_TEST_SUITE(testSuiteName)` macro; it automatically registers a test suite named `testSuiteName`. A test suite definition is ended using `BOOST_AUTO_TEST_END()`. Each test suite can contain one or more test cases. A test case is declared using the macro `BOOST_AUTO_TEST_CASE(testCaseName)`. An example of a test case in presented in Fig. 13.

4.4 Behavior-Driven Development Scenarios

In parallel to classical unit tests which focus more on checking internal functionalities of classes, developers and testers employ also Behavior-Driven Development (BDD) tests which should be examples that anyone from the development team can read and understand. Since the use of scenarios is common at code-level and at the level of the (abstract) model, and since there is a translator that automatically generates C++ code from `AsmetaL` model, we have introduced the `AsmetaBDD` tool which translates an abstract scenario written in the `Avalla` language to BDD code using the Catch2 framework [24]. The `AsmetaBDD` tool generates a C++ scenario that can be compiled together with the C++ code and executed. An example is shown in Fig. 14, where both scenarios check the correctness of the phase transition when 50 s are passed.

```
scenario scenario1

load oneWayTrafficLight.asm

check phase = STOP1STOP2;

set passed(50) := true;

step

check phase = GO2STOP1;
...
```

(a) `Avalla` scenario

```
#include "catch.hpp"
#include "oneWayTrafficLight.hpp"
SCENARIO("oneWayTrafficLight starts") {
 GIVEN("The traffic lights are stopped") {
  oneWayTrafficLight trafficLight;
  REQUIRE(trafficLight.phase == STOP1STOP2);
  WHEN("passed 50 sec") {
   trafficLight.passed(50);
   THEN( "the traffic light is changing state " ) {
    REQUIRE(trafficLight.phase == GO2STOP1);}}
    ....
}}
```

(b) BDD scenario using Catch2

Fig. 14. `AsmetaBDD` scenario example

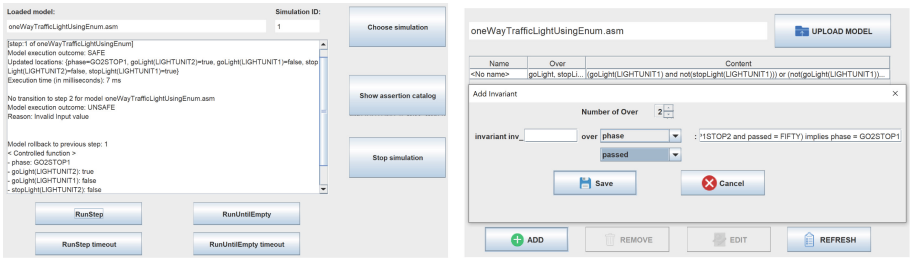
5 ASMETA@operation time

Formal validation and verification techniques usually allow the identification and resolution of problems at design time. However, the state space of a system under specification is often too large or partially unknown at design time, such as for CPSs with uncertain behavior of humans in the loop and/or endowed with self-adaptation capabilities or AI-based components. This makes a complete assurance impractical or even impossible to pursue completely at design time. Runtime assurance methods take advantage of the fact that variables that are free at design time are bound at runtime; so, instead of verifying the complete state space, runtime assurance techniques may concentrate on checking the current state of a system.

Currently, ASMETA supports two types of runtime analysis techniques: runtime simulation described in Sect. 5.1, and runtime monitoring described in Sect. 5.2. Both approaches view the model as a *twin* of the real system and use the model as *oracle* of the correct system behavior. The former exploits the twin execution to prevent misbehavior of the system in case of unsafe model behavior, while the latter exploits the twin execution to check the correctness of the system behavior w.r.t. the model behavior.

5.1 Runtime Simulation

Recently, a runtime simulation platform [44] has been developed within ASMETA to check safety assertions of software systems at runtime and support on-the-fly changes of these assertions. The platform exploits the concept of executable ASM models and it is based on the `AsmetaS@run.time` simulator to handle an ASM model as a living/runtime model [47] and execute it in tandem with a prototype/real system. To this purpose, the runtime simulation platform operates between the system model and the real running system; it traces the state of the ASM model and of the system allowing us to realize a conceivable causal relation depending on the analysis scope and on low-level implementation details. This runtime simulation mechanism, for example, could be used in conjunction with an *enforcer* component tool to concretely sanitize/filter out input events for the running system or to prevent the execution of unsafe commands by the system – *input/output sanitization* [31].



(a) Runtime simulation dashboard `SimGUI` (b) Assertion catalog GUI

Fig. 15. `AsmetaS@run.time`

`AsmetaS@run.time` supports simulation *as-a-service* features of the `AsmetaS` simulator and additional features such as model execution with timeout and model roll-back to the previous safe state after a failure occurrence (e.g., invariant violations, inconsistent updates, ill-formed inputs, etc.) during model execution. `AsmetaS@run.time` allows also the dynamic adaptation of a running ASM model to add/change/delete invariants representing, for example, system safety assertions. This mechanism could be exploited to dynamically add new assertions and guarantee a safer execution of the system after its release, in case dangerous situations have not been foreseen at design time or because of unanticipated changes or situational awareness.

The runtime simulation platform includes also UI dashboards for dynamic *Human-Model-Interaction* (both in a graphical and in a command-line way) which allow the user to track the model execution and change safety assertions. Figure 15a shows the ASM model of the one-way traffic light model through the graphical dashboard `SimGUI`.

In particular, the central panel shows the ASM runs and the simulation results. The last one produced the verdict UNSAFE due to an invalid input value read by the ASM for the enumerative monitored function `passed`. Then, the model is rolled back to its previous safe state. The running ASM model can be adapted dynamically to incorporate new safety invariants or simply modify or cancel existing ones. This can be requested by an external client program or done manually by the user through the GUI `Assertion Catalog` to the simulator engine (see Fig. 15b). Model adaptation is carried out when the model is in a quiescent state, i.e., it is not currently in execution and no other adaptation activity of it is going on. Once adapted, the ASM model execution continues from its current state. A newly added safety invariant that would be immediately violated in the current state of the ASM model is forbidden.

5.2 Runtime Monitoring

ASMETA allows to perform runtime monitoring of a Java program using the tool `CoMA` (*Conformance Monitoring through ASM*) [8]. The approach is shown in Fig. 16 and described as follows:

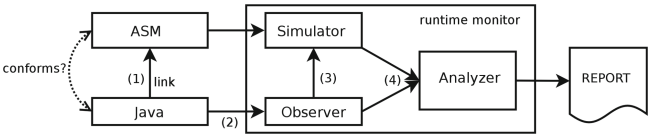


Fig. 16. CoMA: Conformance monitoring through ASM

```
import org.asmeta.monitoring.*;

@Asm(asmFile = "oneWayTrafficLight.asm")
public class OWTL {
  @FieldToLocation(func = "stopLight",
                   args={"LIGHTUNIT1"})
  boolean redLight1;
  @FieldToLocation(func = "stopLight",
                   args={"LIGHTUNIT2"})
  boolean redLight2;
  @FieldToLocation(func = "goLight",
                   args={"LIGHTUNIT1"})
  boolean greenLight1;
  @FieldToLocation(func = "goLight",
                   args={"LIGHTUNIT2"})
  boolean greenLight2;
  private boolean turn1;

  @StartMonitoring
  public OWTL() {
    redLight1 = true; redLight2 = true;
    greenLight1 = false; greenLight2 = false;
    turn1 = false;
  }

  @RunStep
  public void
  updateLights(@Param(func = "passed") Time passedTime) {
      if((passedTime == Time.FIFTY
        && redLight1 && redLight2) ||
          (passedTime == Time.ONEHUNDREDTWENTY
          && greenLight1 != greenLight2)) {
    if(turn1) {
      greenLight1 = !greenLight1;
      redLight1 = !redLight1;
    }
    else {
      greenLight2 = !greenLight2;
      redLight2 = !redLight2;
    }
    if (redLight1 && redLight2) {
      turn1 = !turn1;
    }
  }
 }
}

enum Time {FIFTY, ONEHUNDREDTWENTY, LESS;}
```

Fig. 17. CoMA – Java implementation of the one-way traffic light

- The Java program under monitoring and the ASM model are *link*ed by means of a set of Java annotations[3] (step ①). Some annotations are used to link the Java state with the ASM state; namely, they link class fields of the Java program with functions of the ASM model. Other annotations, instead, specify the methods of the Java program that produce state changes that must be monitored; Fig. 17 shows the Java implementation for the running case study, annotated for the linking with the ASM model shown in Fig. 3;
- the *observer* (step ②) monitors the Java program execution and, whenever a method under monitoring is executed, it performs a simulation step of the ASM model with the simulator (step ③);
- the *analyzer* (step ④) checks whether the Java state after the method execution is conformant with the ASM state after the simulation step. Details on the conformance definition can be found in [8].

CoMA can also check the conformance of nondeterministic systems in which multiple states can be obtained by executing a method under monitoring; namely, the tool checks whether there exists a *next* ASM state that is conformant with the obtained Java state. There are two implementations of this approach: by explicitly listing all the possible next ASM states [9], or by using a symbolic representation with an SMT solver [10].

[3] A Java annotation is a meta-data tag that permits to add information to code elements (class declarations, method declarations, etc.). Annotations are defined similarly as classes.

6 Conclusion and Outlook

This article provided an overview of the ASMETA model-based analysis approach and the associated tooling to the safety assurance problem of software systems using ASMs as underlying analysis formalism. ASMETA allows an open and evolutionary approach to safety assurance as depicted in Fig. 1.

ASMETA is an active open-source academic project. Over the years, it has been improved with new techniques and tools to face the upcoming new challenging aspects of modern systems. It has also been used as a back-end for system analysis of domain-specific front-end notations (as those for service-oriented and self-adaptive systems).

Recently, ASMETA has been extended to deal with model time features, and improvement to support the verification of *quantitative* system properties by means of probabilistic model checking is under development. Application domains under current investigations are those of IoT security, autonomous and evolutionary systems, cyber-physical systems, and medical software certification.

References

1. Al-Shareefi, F.: Analysing Safety-Critical Systems and Security Protocols with Abstract State Machines. Ph.D. thesis, University of Liverpool (2019)
2. Arcaini, P., Bonfanti, S., Gargantini, A., Mashkoor, A., Riccobene, E.: Formal validation and verification of a medical software critical component. In: 2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE), pp. 80–89. IEEE, September 2015. https://doi.org/10.1109/MEMCOD.2015.7340473
3. 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
4. Arcaini, P., Bonfanti, S., Gargantini, A., Riccobene, E.: Visual notation and patterns for abstract state machines. In: Milazzo, P., Varró, D., Wimmer, M. (eds.) Software Technologies: Applications and Foundations, pp. 163–178. Springer International Publishing, Cham (2016)
5. Arcaini, P., Bonfanti, S., Gargantini, A., Riccobene, E., Scandurra, P.: Modelling an automotive software-intensive system with adaptive features using ASMETA. In: Raschke, A., Méry, D., Houdek, F. (eds.) ABZ 2020. LNCS, vol. 12071, pp. 302–317. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-48077-6_25
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 VA 23681–2199, USA, April 2010

8. Arcaini, P., Gargantini, A., Riccobene, E.: CoMA: conformance monitoring of Java programs by Abstract State Machines. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 223–238. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29860-8_17
9. Arcaini, P., Gargantini, A., Riccobene, E.: Combining model-based testing and runtime monitoring for program testing in the presence of nondeterminism. In: Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, pp. 178–187. ICSTW 2013, IEEE Computer Society, Washington, DC, USA (2013). https://doi.org/10.1109/ICSTW.2013.29
10. Arcaini, P., Gargantini, A., Riccobene, E.: Using SMT for dealing with nondeterminism in ASM-based runtime verification. ECEASST **70**, 1–15 (2014). https://doi.org/10.14279/tuj.eceasst.70.970
11. 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
12. 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
13. Arcaini, P., Gargantini, A., Riccobene, E., Scandurra, P.: A model-driven process for engineering a toolset for a formal method. Softw. Pract. Experience **41**, 155–166 (2011). https://doi.org/10.1002/spe.1019
14. 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
15. Arcaini, P., Mirandola, R., Riccobene, E., Scandurra, P.: MSL: a pattern language for engineering self-adaptive systems. J. Syst. Softw. **164**, 110558 (2020). https://doi.org/10.1016/j.jss.2020.110558
16. Arcaini, P., Riccobene, E., Scandurra, P.: Formal design and verification of self-adaptive systems with decentralized control. ACM Trans. Auton. Adapt. Syst. **11**(4), 25:1-25:35 (2017). https://doi.org/10.1145/3019598
17. ASMETA (ASM mETAmodeling) toolset. https://asmeta.github.io/
18. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: a survey. IEEE Trans. Softw. Eng. **41**(5), 507–525 (2015). https://doi.org/10.1109/TSE.2014.2372785
19. Benduhn, F., Thüm, T., Schaefer, I., Saake, G.: Modularization of refinement steps for agile formal methods. In: Duan, Z., Ong, L. (eds.) ICFEM 2017. LNCS, vol. 10610, pp. 19–35. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68690-5_2
20. Bombarda, A., Bonfanti, S., Gargantini, A.: Developing medical devices from abstract state machines to embedded systems: a smart pill box case study. In: Mazzara, M., Bruel, J.-M., Meyer, B., Petrenko, A. (eds.) TOOLS 2019. LNCS, vol. 11771, pp. 89–103. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29852-4_7
21. Bombarda, A., Bonfanti, S., Gargantini, A., Radavelli, M., Duan, F., Lei, Yu.: 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

22. Bonfanti, S., Gargantini, A., Mashkoor, A.: AsmetaA: animator for Abstract State Machines. In: Butler, M., Raschke, A., Hoang, T.S., Reichl, K. (eds.) ABZ 2018. LNCS, vol. 10817, pp. 369–373. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-91271-4_25
23. Bonfanti, S., Gargantini, A., Mashkoor, A.: Generation of C++ unit tests from Abstract State Machines specifications. In: 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 185–193, April 2018. https://doi.org/10.1109/ICSTW.2018.00049
24. Bonfanti, S., Gargantini, A., Mashkoor, A.: Design and validation of a C++ code generator from Abstract State Machines specifications. J. Softw. Evol. Process **32**(2), e2205 (2020). https://doi.org/10.1002/smr.2205
25. Börger, E.: The ASM refinement method. Formal Aspects Comput. **15**, 237–257 (2003)
26. Börger, E., Raschke, A.: Modeling Companion for Software Practitioners. Springer, Heidelberg (2018). https://doi.org/10.1007/978-3-662-56641-1
27. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer Verlag, Heidelberg (2003). https://doi.org/10.1007/978-3-642-18216-7
28. Calinescu, R., Weyns, D., Gerasimou, S., Iftikhar, M.U., Habli, I., Kelly, T.: Engineering trustworthy self-adaptive software with dynamic assurance cases. IEEE Trans. Softw. Eng. **44**(11), 1039–1069 (2018). https://doi.org/10.1109/TSE.2017.2738640
29. Carioni, A., Gargantini, A., Riccobene, E., Scandurra, P.: A scenario-based validation language for ASMs. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 71–84. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87603-8_7
30. Cimatti, A., et al.: NuSMV 2: an opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_29
31. Falcone, Y., Mariani, L., Rollet, A., Saha, S.: Runtime failure prevention and reaction. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 103–134. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_4
32. Fraser, G., Wotawa, F., Ammann, P.E.: Testing with model checkers: a survey. Softw. Test. Verif. Reliab. **19**(3), 215–261 (2009)
33. Gargantini, A., Riccobene, E.: ASM-based testing: coverage criteria and automatic test sequence. J. Univers. Comput. Sci. **7**(11), 1050–1067 (2001). https://doi.org/10.3217/jucs-007-11-1050
34. 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
35. Gargantini, A., Riccobene, E., Scandurra, P.: A semantic framework for metamodel-based languages. Autom. Softw. Eng. **16**(3–4), 415–454 (2009). https://doi.org/10.1007/s10515-009-0053-0
36. Gargantini, A., Riccobene, E., Scandurra, P.: Ten reasons to metamodel ASMs. In: Abrial, J.-R., Glässer, U. (eds.) Rigorous Methods for Software Construction and Analysis. LNCS, vol. 5115, pp. 33–49. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-11447-2_3

37. Gaspari, P., Riccobene, E., Gargantini, A.: A formal design of the Hybrid European Rail Traffic Management System. In: Proceedings of the 13th European Conference on Software Architecture - Volume 2. pp. 156–162. ECSA 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3344948.3344993
38. Gurevich, Y.: Evolving Algebras 1993: Lipari Guide, pp. 9–36. Oxford University Press Inc., USA (1995)
39. Harel, D., Rumpe, B.: Meaningful modeling: What's the semantics of "Semantics"? Computer **37**(10), 64–72 (2004). https://doi.org/10.1109/MC.2004.172
40. Leveson, N.: Are you sure your software will not kill anyone? Commun. ACM **63**(2), 25–28 (2020). https://doi.org/10.1145/3376127
41. Lutz, R.R.: Software engineering for safety: a roadmap. In: Proceedings of the Conference on The Future of Software Engineering, pp. 213–226. ICSE 2000, Association for Computing Machinery, New York, NY, USA (2000). https://doi.org/10.1145/336512.336556
42. Mirandola, R., Potena, P., Riccobene, E., Scandurra, P.: A reliability model for service component architectures. J. Syst. Softw. **89**, 109–127 (2014). https://doi.org/10.1016/j.jss.2013.11.002
43. Riccobene, E., Scandurra, P.: A formal framework for service modeling and prototyping. Formal Aspects Comput. **26**(6), 1077–1113 (2013). https://doi.org/10.1007/s00165-013-0289-0
44. Riccobene, E., Scandurra, P.: Model-based simulation at runtime with Abstract State Machines. In: Muccini, H., et al. (eds.) Software Architecture, pp. 395–410. Springer International Publishing, Cham (2020)
45. Schmidt, D.C.: Guest editor's introduction: model-driven engineering. IEEE Comput. **39**(2), 25–31 (2006). https://doi.org/10.1109/MC.2006.58
46. Utting, M., Legeard, B., Bouquet, F., Fourneret, E., Peureux, F., Vernotte, A.: Chapter two - recent advances in model-based testing. Advances in Computers, vol. 101, pp. 53–120. Elsevier (2016). https://doi.org/10.1016/bs.adcom.2015.11.004
47. Van Tendeloo, Y., Van Mierlo, S., Vangheluwe, H.: A multi-paradigm modelling approach to live modelling. Softw. Syst. Model. **18**(5), 2821–2842 (2018). https://doi.org/10.1007/s10270-018-0700-7
48. Vessio, G.: Reasoning about properties with Abstract State Machines. In: Gogolla, M., Muccini, H., Varró, D. (eds.) Proceedings of the Doctoral Symposium at Software Technologies: Applications and Foundations 2015 Conference (STAF 2015), L'Aquila, Italy, 20 July 2015. CEUR Workshop Proceedings, vol. 1499, pp. 1–10. CEUR-WS.org (2015). http://ceur-ws.org/Vol-1499/paper1.pdf
49. Weyns, D., et al.: Perpetual assurances for self-adaptive systems. In: de Lemos, R., Garlan, D., Ghezzi, C., Giese, H. (eds.) Software Engineering for Self-Adaptive Systems III. Assurances. LNCS, vol. 9640, pp. 31–63. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-74183-3_2