








ASMETA Tool Set for Rigorous System Design

Andrea Bombarda¹(✉) , Silvia Bonfanti¹ , Angelo Gargantini¹ ,
Elvinia Riccobene² , and Patrizia Scandurra¹ 



¹ University of Bergamo, Bergamo, Italy
{andrea.bombarda,silvia.bonfanti,
angelo.gargantini,patrizia.scandurra}@unibg.it

² Università degli Studi di Milano, Milan, Italy
elvinia.riccobene@unimi.it



Abstract. This tutorial paper introduces ASMETA, a comprehensive suite of integrated tools around the formal method Abstract State Machines to specify and analyze the executable behavior of discrete event systems. ASMETA supports the entire system development life-cycle, from the specification of the functional requirements to the implementation of the code, in a systematic and incremental way. This tutorial provides an overview of ASMETA through an illustrative case study, the Pill-Box, related to the design of a smart pillbox device. It illustrates the practical use of the range of modeling and V&V techniques available in ASMETA and C++ code generation from models, to increase the quality and reliability of behavioral system models and source code.

1 Introduction

It is widely recognized that formal methods need to be supported by automated tools to be of practical use and promote their adoption, especially when they are required by critical application areas (such as security and safety) and standards for software certification and accreditation [6, 7, 22, 23]. This tutorial presents ASMETA¹, an open-source framework defining modeling notations and tools inspired by the well-known formal method of the Abstract State Machines (ASMs) [14, 15]. ASMETA supports model editing, visualization, simulation, animation, validation, verification, as well as code generation from formal models.

In the wide range of existing formal methods [17, 21], and more specifically of state-based formal methods², the ASM-based formal method supported by

¹ <https://asmeta.github.io/>.

² <https://abz-conf.org/methods/>.

The work of Andrea Bombarda is supported by PNRR - ANTHEM (Advanced Technologies for Human-centred Medicine) - Grant PNC0000003 - CUP: B53C22006700001 - Spoke 1 - Pilot 1.4. The work of Silvia Bonfanti, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

© The Author(s) 2025

A. Platzer et al. (Eds.): FM 2024, LNCS 14934, pp. 492–517, 2025.

https://doi.org/10.1007/978-3-031-71177-0_28

ASMETA offers several advantages: (1) models have a *pseudo-code format*, so practitioners can easily understand them as *high-level programs*; (2) systems can be specified at any desired *level of abstraction*, depending on the level of details one wants to achieve; (3) models are *executable*, so they are suitable also for lighter forms of model analysis such as simple simulation to check model consistency w.r.t. system requirements; (4) techniques for mapping models to code (e.g., to C++ or Java) are supported, so *correct-by-construction* development is possible; (5) *multi-agents modeling* is supported, making possible the specification of *distributed systems*. Moreover, the ASMETA framework allows for the integrated use of tools for different forms of model analysis, it is maintained and under continuous features improvement.

Through an illustrative case study from the healthcare domain, the Pill-Box system, this tutorial shows how to model in ASMETA the executable behavior of a system with a discrete state space. Then, the tutorial guides the readers through the use of the ASMETA tools to apply several model validation and verification (V&V) techniques, such as simulation, scenario-based validation, and formal verification of user-defined properties and meta-properties. A model refinement process supported by ASMETA is also presented by means of the running case study and by explaining how in formalizing the system behavior it is possible to evolve a partial specification (ground model) into a more complete model. Finally, the tutorial showcases the automatic generation of executable C++ code from the Pill-Box model, developed and verified in ASMETA.

This tutorial is intended to be a resource for software engineers and researchers that want to leverage lightweight formal methods in their projects. The hands-on approach, adopting the Pill-Box system as running example, endows the readers with the necessary skills to start adopting ASMETA for a more rigorous system design that increases the quality and reliability of behavioral system models and source code. ASMETA is distributed as an open-source solution so that other researchers can contribute to its extension.

The remainder of this tutorial is organized as follows. Section 2 introduces the ASMs, while Sect. 3 introduces the ASMETA framework together with a modeling process, and provides all useful references. Section 4 presents the running case study. Section 5 describes the user-facing modeling language *AsmetaL* to define ASM models. Sections from 6 to 9 explain the ASMETA tooling supporting all model analysis techniques for a rigorous system design. Section 10 presents model refinement applied to the running case study. Section 11 explains how to generate C++ code from the verified Pill-Box model. Finally, Sect. 12 concludes.

2 Abstract State Machines

Before introducing ASMETA, here we provide a basic introduction of the state-based formal method of ASMs [14,15]. *States* are mathematical *algebras* specifying a system configuration by means of arbitrarily complex data, i.e., domains of elements with functions defined on them. State *transitions* are expressed by named and parameterized transition rules describing how the data (function values saved into *locations*) change from one state to the next one.

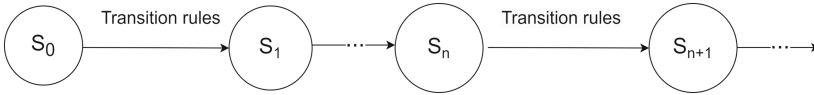


Fig. 1. An ASM run with a sequence of states and state-transitions (steps)

The functions of the algebra are classified into *dynamic* and *static* depending on whether they are updated or not by transition rules. The dynamic functions are further distinguished in *monitored* (read by the machine and modified by the environment), *controlled* (read and written by the machine), *out* (only written by the machine and read by the environment), and *shared* (read and written by the machine and its environment). In addition, functions that are defined in terms of other (dynamic) functions are called *derived*.

Dynamic functions are updated by firing transition rules. The basic transition rule is the *update* rule for function update; it has form $f(t_1, \dots, t_n) := v$, where f is an n -ary function, t_i with $i = 1..n$ are terms, and v the new value to be associated with the location $f(t_1, \dots, t_n)$ in the next state. As in structured programming, constructs for structured control flow can be used to form transition rules depending on the type of update structure they express. The main rule constructors include: guarded updates (*if-then*, *switch-case*), simultaneous parallel updates (*par*), non-determinism (*choose*), unrestricted synchronous parallelism (*for-all*), abbreviation on terms of rules (*let*), etc.

ASMs can be read as pseudocode over abstract data with a well-defined execution semantics. An ASM *run* (see Fig. 1) is a (finite or infinite) sequence $S_0, S_1, \dots, S_n, \dots$ of states. Starting from the initial state S_0 , in a computation step (*run step*) from S_n to S_{n+1} , all enabled transition rules are executed in parallel, leading to simultaneous updates of a number of locations. In case of an inconsistent update (i.e., the same location is updated to two different values) or invariant violations (i.e., some property that must be true in every state is violated), the model execution stops with error.

3 Overview of the ASMETA Toolset

The ASMETA project started in 2004 with the aim of addressing the deficiency in tools that support ASMs. Although the formal approach had demonstrated widespread application in specifying and verifying various software systems across diverse domains (as evidenced by the ASM research summary in [15]), the absence of supportive tools for the ASM method was deemed a limitation, leading to skepticism regarding its practical utility.

To address this issue, ASMETA has been developed by exploiting the Model-Driven Engineering (MDE) approach [5] for software development starting from the definition of a meta-model for an abstract notation able to capture the *working definition* (see [15, pag. 32]) of an ASM. From the metamodel, a textual notation for encoding ASM models has been derived, and has been enriched, during the years, to support many V&V activities in the rigorous design of software systems. These analysis techniques have been proven to be beneficial for

the safety assurance of safety-critical systems with event-based behavior and discrete state spaces. See [1] for further details on the case studies and application domains (including medical software, software control systems, and service-based systems, to name a few) to which ASMETA has been applied.

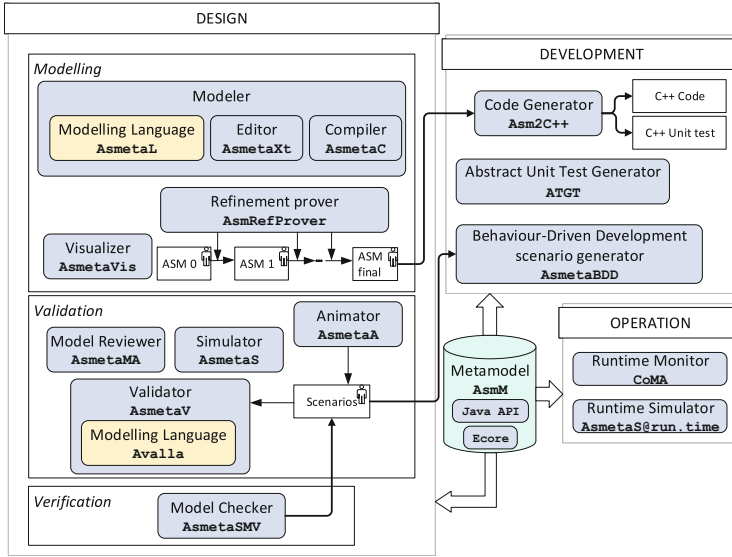


Fig. 2. ASMETA-based development process

3.1 Getting and Using ASMETA

Most ASMETA tools are integrated with the Eclipse IDE³. An Eclipse package containing the ASMETA toolset is available and released periodically at <https://github.com/asmeta/asmeta>. In the same location, the source code of all the ASMETA tools, together with examples of ASMETA specifications, is available.

Tooling. ASMETA tools support the main activities of the software development process from formal requirement specification to code generation. Figure 2 shows the tools usage in the various stages [1]. At *design* time, ASMETA provides tools for model editing and visualization (the modeling language **AsmetaL**⁴ and its editor and compiler, plus the model visualizer **AsmetaVis** for graphical visualization of ASMETA models), model validation (e.g., interactive or random simulation by the simulator **AsmetaS**⁵, animation by the animator **AsmetaA**, scenario construction and validation by the validator **AsmetaV**, and static analysis

³ An Eclipse package including all tools and models useful for this tutorial is available at <https://doi.org/10.5281/zenodo.12770854>.

⁴ It is a concrete notation for the abstract one defined by the metamodel reflecting the working definition of an ASM.

⁵ The ASMETA model simulator implements the computational paradigm (concepts and semantics) of an ASM run as defined in the previous section.

by the model reviewer **AsmetaMA**), and verification (proof of temporal properties by the model checker **AsmetaSMV**, and proof of correct model refinement by **AsmRefProver**). During software *development*, ASMETA supports automatic code and test case generation from models (the code generator **Asmeta2C++**, the unit test generator **ATGT**, and the acceptance test generator **AsmetaBDD** for complex system scenarios). If the system is available, during its *operation*, ASMETA can be used for runtime monitoring (by the tool **CoMA**) and runtime simulation (by **AsmetaS@run.time**).

Remark. Due to lack of space and to keep this tutorial simple and understandable to new and unfamiliar users, in the following sections we explain and show the application of a selected number of tools, those supporting the initial and fundamental steps of system modeling, analysis (V&V) and encoding. Focusing more on pedagogical rather than technical aspects of our modeling approach, we also skip advanced modeling features (e.g., the concepts of multi-agent ASMs or I/O ASMs, suitable to model distributed and composable systems) which require understanding the basic and preliminary concepts around ASMETA; this is what this tutorial intends to cover.

Modeling Process. ASMETA derives its foundation from the ASM theory, thus, akin to ASMs, its modeling methodology follows an iterative approach with a focus on model refinement. Concretely, ASMETA employs *stuttering refinement* [4], a specialized variant of the broader ASM refinement [13]. This refinement-based process allows users to tackle the complexity of the requirements and to bridge, in a seamless manner, specification to code. Requirements modeling begins with the creation of a high-level ASMETA model, akin to the ASM ground model [15]. This model is delineated through the analysis of informal requirements typically presented in natural language. Model signature and rule naming are set by using terms of the application domain and derived from textual requirements, thereby simplifying the process of connecting requirements to the model. This high-level model (see model ASM_0 in Fig. 2) should be *correct* and *consistent*, i.e., it should represent the intended requirements (at the desired abstraction level) and no ambiguities of initial requirements should be left. It is not necessary for ASM_0 to be *complete*, i.e., it may not specify some given requirements that are later captured during the refinement process. Indeed, the modeling process supported by ASMETA is a refinement-based one: starting from the model ASM_0 , through a sequence of *refined* models ASM_1, ASM_2, \dots , other functional requirements are specified and modeled, till the desired level of completeness is reached. At the end of this process, ASM_{final} captures all intended requirements at the desired level of abstraction. When performing refinements, it is important to prove that each refined model is a correct (stuttering) refinement of the previous one. The ASMETA framework includes the **AsmRefProver** tool [4] which supports the user in this activity and automatically performs the correctness check of refinement steps.

Starting from the very first model, the ASM_0 , during the modeling process the user should perform validation and verification (V&V) activities to assure requirements satisfaction and property validity.

4 The Pill-Box Case Study

In this section, we introduce the Pill-Box case study [8] with its informal requirements, which will be used throughout the paper as a running example to describe the modeling and analysis activities supported by the ASMETA tools.

The Pill-Box device is a medicine/pill dispenser that has a certain number of drawers (e.g., three drawers). Each drawer contains multiple slots (one for each pill) that are emptied in sequence. In each drawer, only one specific type of medicine can be placed. So, each drawer can contain multiple pills (one per slot) but all pills must be of the same drug type.

Each Pill-Box drawer has a switch and a LED. The former is used to notify whether the pill in the drawer has been taken, and the latter is used to signal relevant information to the user. When the LED is *OFF*, it is not time to take the corresponding pill, while when the LED is *ON*, it means that the patient should assume that pill. When it is time to take a pill, the LED stays *ON* for 10 minutes after the scheduled time of the pill.

For each pill type, it is possible to set several deadlines throughout the day, meaning that the same drawer might be opened multiple times. However, if two or more pills have to be taken at the same time, the Pill-Box turns on only a single LED per time, by randomly choosing the order in which to assume them. Here we introduce three models for the ASMETA specification of the Pill-Box, where each one introduces new elements for refining time and pill management:

- *Ground model* (`pillbox_ground`): here we abstract the requirement that a drawer contains multiple slots and consider only a single pill per drawer. Moreover, time is not explicitly modeled, and information on the time passed is given by an external event (a monitored Boolean-valued function).
- *Model with time* (`pillbox_time`): this specification models time passing by a timer. We still keep the abstraction of having a single pill per drawer.
- *Final model* (`pillbox_final`): it captures all requirements of the Pill-Box system, and it thus specifies multiple pills (and multiple deadlines) per drawer.

These ASMETA models and all the other related artifacts are presented in part in this paper; their complete version can be found in *Models.zip* file at <https://doi.org/10.5281/zenodo.12770854>.

5 AsmetaL: The ASMETA Language

This section introduces the textual language **AsmetaL**, the user-facing language to define ASMETA models. The main modeling constructs of **AsmetaL** are here illustrated using the ground model `pillbox_ground` introduced in Sect. 4.

An ASMETA specification is described in a text file with extension `.asm` and structured as shown in Listing 1. It has five main sections:

<pre> asm pillbox_ground import ../STDL/StandardLibrary ... signature: // DOMAINS abstract domain Drawer enum domain LedLights = {OFF ON} ... // FUNCTIONS dynamic monitored isPillTaken: Drawer -> Boolean ... dynamic controlled drawerLed: Drawer -> LedLights ... derived isOn: Drawer -> Boolean ... static drawer1: Drawer </pre>	<pre> definitions: // FUNCTIONS DEFINITIONS function isOn(\$d in Drawer) = (drawerLed(\$d) = ON) ... // RULE DEFINITIONS rule r_reset(\$drawer in Drawer) = // INVARIANTS AND PROPERTIES invariant inv_drawer1 over Drawer = // MAIN Rule main rule r_Main = // INITIAL STATE default init s0: // Turn-off all the LEDs for the Drawers function drawerLed(\$drawer in Drawer) = OFF ... </pre>
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Listing 1. Structure of an ASMETA specification

- The section **import** allows us to include all or some of the declarations and definitions given in another ASMETA model.
- The section **signature** is where domains and functions are declared.
- The section **definitions** contains the definition of static concrete domains, static or derived functions, all transition rules, and possible state invariants, i.e., first-order formulas that must be true in all states.
- The section **main rule** defines the rule that is the starting point of the computation at each state; it may, in turn, call the other transition rules (defined as *macro call rules*⁶). A *run step* of an ASMETA model is the execution of all transition rules, which are directly or indirectly called from the main rule and are enabled to fire.
- The section **default init** introduces the initial values for dynamic concrete domains and dynamic functions declared in the **signature**.

Here we provide a more detailed look at each part of an ASMETA specification.

Specification Name. The first line of the specification contains the keyword **asm** followed by the name of the specification, which must be the same as the file. For instance:

```
asm pillbox_ground
```

indicates that the specification name is **pillbox_ground** and it must be defined in the file **pillbox_ground.asm**.

A model without the main rule is called a **module**⁷. It consists of declarations and definitions of domains, functions, invariants, macro call rules, and it can be imported by other ASMETA models. Note that an ASMETA model (the model that starts with the keyword **asm**) can be imported as well, except for the initial state and the main rule.

⁶ Note that to define a macro call rule in the **definitions** section we use the syntax **macro r_rule(params)**, while the macro rule is invoked from another rule as **r_rule[params]**.

⁷ A module name corresponds to the first word used in the **.asm** file.

Import. An `AsmetaL` specification can import modules, by using the file name with its relative or absolute path. For instance, the following line imports the `StandardLibrary`:

```
import ../STDLib/StandardLibrary
```

The `StandardLibrary` is a user-ready module that defines names for basic domains and functions. This library is mandatory to import since it includes predefined names for primitive domains (like `Boolean`, `Natural`, `Integer`, etc.) and functions for the main operations over these domains and structured domains (for tuples, sequences, sets, bags, and maps). Other libraries are available⁸ as explained in the following sections.

Signature Domains. The `AsmetaL` language allows the user to specify domains of different type:

- *Basic domains*: represent primitive data values and are denoted by ready-to-use domain symbols of the standard library (`Boolean`, `Natural`, `Integer`, `Complex`, `Char`, and `String`).
- *Enum domains*: finite enumeration of elements defined by the user.
- *Abstract domains*: (non-enumerable) user-defined domain to describe abstract entities of the real word.
- *Concrete domains*: user-named domain defined as sub-domain of another domain.
- *Structured domains*: representing structured data (like finite sets, tuples, maps) over other domains; examples are the Cartesian `Product` of two or more domains, and the mathematical `Powerset` of a domain.

Examples of user-defined domains from the ground model of the Pill-Box are:

```
abstract domain Drawer
enum domain LedLights = {OFF | ON }
enum domain Drugs = {TYLENOL | ASPIRINE | MOMENT}
```

`Drawer` is an abstract domain representing the drawer objects; such objects typically do not have a precise structure and the user further characterizes them by introducing functions over them (see next paragraph). `LedLights` is the enumeration for the light status of the LEDs; `Drugs` is the enumeration of three different types of drugs (`Tylenol`, `Aspirine`, and `Moment`).

Signature Functions. Basic functions form the basic signature of the machine and are classified into `static`, which never change during any run of the machine, and `dynamic`, that may be changed by the environment or by the machine updates. Dynamic functions are further divided into `monitored`, `controlled`, `shared`,

⁸ https://github.com/asmeta/asmeta/blob/master/asm_examples/STDLib/.

and `out`. `AsmetaL` adopts appropriate keywords for declaring all these kinds of functions. Examples of declarations of static functions are the constants⁹ representing the three drawers (elements of the abstract domain `Drawer`):

```
static drawer1: Drawer
static drawer2: Drawer
static drawer3: Drawer
```

Examples of dynamic functions declaration in the ground model are¹⁰:

```
dynamic monitored isPillTaken: Drawer -> Boolean
dynamic monitored pillDeadlineHit: Drawer -> Boolean
dynamic controlled drawerLed: Drawer -> LedLights
dynamic controlled drug: Drawer -> Drugs
dynamic controlled isPillTobeTaken: Drawer -> Boolean
```

The function `isPillTaken` is a monitored function, and it is true when the user confirms he/she has taken the pill. Similarly, the monitored function `pillDeadlineHit` signals that the deadline for the pill contained in a specific drawer has come. A drawer contains a `drug` and has a `drawerLed` which is `ON` when it is time to take the pill. In addition, the Pill-Box uses the characteristic or indicator function `isPillTobeTaken` to store the drawers for which the deadline has been hit. These functions are controlled since their value is set by the machine.

In addition to basic functions, the modeler can introduce *derived* functions, i.e., those coming with a specification or computation mechanism defined in terms of other basic functions. Examples of declarations of derived functions are as follows:

```
derived isOn: Drawer -> Boolean
derived isOff: Drawer -> Boolean
derived areOthersOn: Drawer -> Boolean
```

These functions are used as guards in the transition rules and are defined (see below) in terms of the `drawerLed` controlled function.

Definitions Functions. Once declared, static and derived functions must also be defined explicitly in the definitions section. The notation to define functions is as in the following examples:

```
function isOn($d in Drawer) = (drawerLed($d) = ON)
function isOff($d in Drawer) = (drawerLed($d) = OFF)
function areOthersOn($d in Drawer) = switch($d)
  case drawer1 : isOn(drawer2) or isOn(drawer3)
  case drawer2 : isOn(drawer1) or isOn(drawer3)
  case drawer3 : isOn(drawer2) or isOn(drawer1)
endswitch
```

⁹ The domain is optional. Functions of arity 0 are common *variables* of programming; 0-ary static functions are constants.

¹⁰ The keyword **dynamic** is optional.

The right-hand term specifies the function law. In the case of the derived function `areOthersOn`, the right-end term is a logical map that associates domain elements to codomain elements. The target domains of the formal parameters are to be the same as those specified in the function declaration, and the domain type of the right-end term must be compatible with the function codomain. Note that, as exception to this explanation, static 0-ary functions (constants) over an abstract domain (such as `drawer1`) do not need to be defined.

Definitions Rules. An update rule is the basic form of a transition rule. Typically, an ASM transition system appears as a set of *guarded updates* or *conditional rules* of form **if** *cond* **then** *updates*, where function *updates* are simultaneously executed when the condition *cond* (also called “guard”) evaluates to true. An example of a conditional rule is as follows:

```
if pillDeadlineHit($drawer) then isPillTobeTaken($drawer) := true endif
```

It sets to true the value of the function `isPillTobeTaken` for a given drawer `$drawer`¹¹ when it is time to take the drug of that drawer (denoted by the monitored function `pillDeadlineHit`).

In `AsmetaL`, the transition rules can be defined after the definition of concrete domains (if any) and functions. A rule definition starts with the keywords **macro** (it is optional) and **rule**, followed by the name of the rule with the fixed prefix `r_`, the list of free variables and their typing domains, and the rule body (containing occurrences of the free variables). As an example of rule definition, consider the rule `r_reset` that uses a **par** rule to reset the status of a given drawer (in parallel it sets the led to OFF and `isPillTobeTaken` to false):

```
rule r_reset($drawer in Drawer) = par
  drawerLed($drawer) := OFF
  isPillTobeTaken($drawer) := false endpar
```

Once defined, a named rule can be invoked (like in structured programming) within the rule body of another rule by using the rule name followed by the list of actual arguments (if any)¹² surrounded within square brackets (e.g., `r_reset[$drawer]`). When the rule is invoked, it is expanded by replacing every variable freely occurring within the rule body with the actual argument of the invocation (the association is positional).

The **par** rule and the **forall** rule are rule constructors realizing *synchronous parallelism* since both allow the synchronous parallel execution of multiple transition rules. The only difference is that the **par** rule expresses *bounded parallelism*, while the **forall** rule expresses potentially *unbounded parallelism*. An example of a **forall** rule in the Pill-Box ground model is in the rule definition:

¹¹ In `AsmetaL` the name of a variable freely occurring in a rule starts with the prefix `$`.

¹² The number of actual parameters must be equal to the number of the formal parameters of the rule to invoke and be domain-compatible with them. Invocations of rules of arity 0 is also allowed; in this case the list of parameters is empty.

```

rule r_setOtherDrawers = forall $drawer in Drawer do par
  if pillDeadlineHit($drawer) and isOff($drawer) then isPillTobeTaken($drawer) := true endif
  if isOn($drawer) and isPillTaken($drawer) then r_reset[$drawer] endif endpar

```

Such a rule, in parallel for all potential drawers, sets the status of a drawer if it is time to take the drawer's pill (pillDeadlineHit is true), or resets it in case the drawer's LED is on and the drawer's pill has already been taken. ASMETA supports non-deterministic operations, which are implemented by selecting a domain and picking a random element from it. This concept is realized by means of the **choose** rule. An example of rule definition that uses the **choose** rule is for the non-deterministic choice of one pill to take when there are more to take at a certain time.

```

rule r_choosePillToTake = choose $drawer in Drawer with
  isPillTobeTaken($drawer) and isOff($drawer) and not areOthersOn($drawer) do drawerLed($drawer) := ON

```

Since only a single red LED is to be on at a time, at each step the Pill-Box chooses randomly one still off among those of the drawers containing a pill to be taken, but only if all the other drawers' LEDs are off, and turns it (if any) on.

Definitions Invariants. Invariants allow users to specify first-order logic formulas that must be true in each computational state during model execution. In **AsmetaL**, invariants are defined after rule definitions but precede the main rule definition (see Sect. 6 for further details).

Definitions Properties. After the invariants, Computation Tree Logic (CTL) and Linear Temporal Logic (LTL) properties can be defined in an ASMETA model (see Sect. 9 for further details).

Main Rule. The main rule designates the initial transition rule to execute (the entry point of the machine's program) at each computational step. Its definition follows those of invariants and properties, and has no formal parameters (its arity is 0). The main rule for the Pill-Box ground model is:

```

main rule r_Main = par
  r_choosePillToTake[]
  r_setOtherDrawers[] endpar

```

It is in charge of simultaneously (i) choosing one drawer with a pill to take (if any) and (ii) managing the state of the other drawers.

Initial State. An ASMETA specification may contain the initialization of controlled functions to the value that they must assume when the execution of the model starts. The syntax and rules to assign an initial value to a controlled

function is the same for defining static/derived functions. For instance, in the following model fragment, the `drawerLed` function, for all drawers, is set to `OFF`, as well as the `isPillTobeTaken` function, which is set to `false`. Finally, the `drug` function associates a different type of drug to each drawer.

```

function drawerLed($drawer in Drawer) = OFF
function isPillTobeTaken($drawer in Drawer) = false
function drug($drawer in Drawer) = switch($drawer)
    case drawer1 : TYLENOL
    case drawer2 : ASPIRINE
    case drawer3 : MOMENT
endswitch
    
```

If a function is not initialized, all its locations take the special value `undef`¹³.

6 Model Simulation

Simulation is the first validation activity usually performed to check an ASMETA model's behavior during its development, and it is supported by the **AsmetaS** tool [5]. Given a model, at every step, the simulator builds the update set according to the theoretical definitions given in [15] 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. **AsmetaS** can be executed from the command line¹⁴ and from the Eclipse interface. By using the Eclipse UI, the **AsmetaS** toolbar has three buttons (see Fig. 3) with three actions:

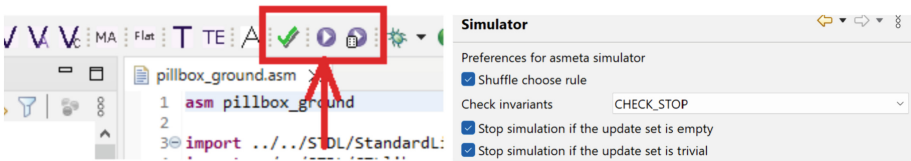





Fig. 3. AsmetaS commands and options panel

-  Parse the model and type check it
-  Execute the model in interactive mode
-  Execute the model with random inputs

¹³ Although the parser does not force you to initialize all the controlled functions, it is strongly suggested to avoid run-time errors due to a missing initialization.

¹⁴ More details are available in the Appendices file at <https://doi.org/10.5281/zenodo.12770854>.

1	Running interactively pillbox_ground.asm	<UpdateSet - 1>	29
2	INITIAL STATE:Drawer={drawer1,drawer2,drawer3}	drawerLed(drawer1)=ON	30
3	Insert a boolean constant for pillDeadlineHit(drawer1):	</UpdateSet>	31
4	true	<State 2 (controlled)>	32
5	Insert a boolean constant for pillDeadlineHit(drawer2):	drawerLed(drawer1)=ON	33
6	false	drawerLed(drawer2)=OFF	34
7	Insert a boolean constant for pillDeadlineHit(drawer3):	drawerLed(drawer3)=OFF	35
8	false	isPillTobeTaken(drawer1)=true	36
9	<State 0 (monitored)>	isPillTobeTaken(drawer2)=false	37
10	pillDeadlineHit(drawer1)=true	isPillTobeTaken(drawer3)=false	38
11	pillDeadlineHit(drawer2)=false	</State 2 (controlled)>	39
12	pillDeadlineHit(drawer3)=false	Insert a boolean constant for isPillTaken(drawer1):	40
13	</State 0 (monitored)>	true	41
14	<UpdateSet - 0>	<State 2 (monitored)>	42
15	isPillTobeTaken(drawer1)=true	isPillTaken(drawer1)=true	43
16	</UpdateSet>	</State 2 (monitored)>	44
17	<State 1 (controlled)>	<UpdateSet - 2>	45
18	drawerLed(drawer1)=OFF	drawerLed(drawer1)=OFF	46
19	drawerLed(drawer2)=OFF	isPillTobeTaken(drawer1)=false	47
20	drawerLed(drawer3)=OFF	</UpdateSet>	48
21	isPillTobeTaken(drawer1)=true	<State 3 (controlled)>	49
22	isPillTobeTaken(drawer2)=false	drawerLed(drawer1)=OFF	50
23	isPillTobeTaken(drawer3)=false	drawerLed(drawer2)=OFF	51
24	</State 1 (controlled)>	drawerLed(drawer3)=OFF	52
25	Insert a boolean constant for isPillTaken(drawer1): false	isPillTobeTaken(drawer1)=false	53
26	<State 1 (monitored)>	isPillTobeTaken(drawer2)=false	54
27	isPillTaken(drawer1)=false	isPillTobeTaken(drawer3)=false	55
28	</State 1 (monitored)>	</State 3 (controlled)>	56

Fig. 4. Output of the interactive Simulation of the Pill-Box using *AsmetaS*

In the simulator option panel (see Fig. 3), the user can set the preferences regarding the **choose** rule, when to stop the random simulation (until the update set becomes empty or trivial), and how to handle invariants and axioms.

In Fig. 4, we show the result of the interactive simulation for the Pill-Box when the pill in drawer 1 hits the deadline (in State 0 - line 10), so the pill becomes to be taken (State 1 - line 21), the led becomes ON (State 2 - line 33), the user takes the pill, and the led becomes OFF (State 3 - line 50). Note that the update set is computed in the current state and is applied only in the next one. For instance, when the monitored location `pillDeadlineHit(drawer1)` is set true by the user in the initial state:

State 0 (**monitored**): `pillDeadlineHit(drawer1)=true`

the following rule:

if `pillDeadlineHit($d)` **and** `isOff($d)` **then** `isPillTobeTaken($d) := true` **endif**

checks the current state (State 0) and since the deadline is hit and the LED is off, the update set will contain the update of the location `isPillTobeTaken(drawer1)`, which is updated only in the next state (State 1):

```

<UpdateSet - 0>
isPillTobeTaken(drawer1)=true
</UpdateSet>
<State 1 (controlled)>
isPillTobeTaken(drawer1)=true
...
</State 1 (controlled)>

```

Invariant Checking. AsmetaS implements an invariant checker, which (optionally) checks in every state reached during the computation if the invariants (if any) declared in the specification are satisfied or not. If an invariant is not satisfied, AsmetaS throws an `InvalidInvariantException`, which keeps track of the violated invariants and of the update set which has caused such violation. The invariant checker is particularly useful during the first phase of the model development to validate the specification. The designer adds model invariants, activates the invariant checker from the simulator options, and runs the model with some critical inputs. For example, with the following invariant:

```

invariant inv_drawer1 over Drawer: (forall $d in Drawer with isOff($d))

```

As soon as a led becomes ON, the computation stops:

```

<State 2 (controlled)>
drawerLed(drawer2)=ON
...
</State 2 (controlled)>
INVARIANT violations
FINAL STATE: ....
run terminated

```

Consistent Updates Checking. AsmetaS is able to reveal inconsistent updates by throwing an `UpdateClashException`. The `UpdateClashException` records the location being inconsistently updated and the two different values assigned to it. The user, analyzing this error, can detect the fault in the specification. As the invariant checker, this feature is useful for model validation. For example, suppose to modify the `r_setOtherDrawers` rule by removing the strike-through condition in the first conditional term as shown in the following code:

```

rule r_setOtherDrawers = forall $drawer in Drawer do par
  if pillDeadlineHit($drawer) and isOff($drawer) then isPillTobeTaken($drawer) := true endif
  if isOn($drawer) and isPillTaken($drawer) then r_reset[$drawer] endif endpar

```

The simulator signals an inconsistent update on the `isPillTobeTaken(drawer1)` location with the following message:

```

INCONSISTENT UPDATE FOUND !!! : location isPillTobeTaken(drawer1) updated to true != false

```

The screenshot shows the AsmetaA animation interface. On the left, there are controls for 'Do one interactive step', 'Do random step/s', 'Insert random step number' (set to 1), and 'Invariant violation / exceptions'. Below these are buttons for 'Controlled Functions', 'Monitored Functions', and 'export to Avalla'. The main part of the interface is a table with columns for 'Type', 'Functions', and four states: 'State 0', 'State 1', 'State 2', and 'State 3'. The table lists various functions and their values across the states.

Type	Functions	State 0	State 1	State 2	State 3
<input type="checkbox"/> ^ C	drawerLed(drawer1)	OFF	OFF	ON	OFF
<input type="checkbox"/> ^ C	drawerLed(drawer2)	OFF	OFF	OFF	OFF
<input type="checkbox"/> ^ C	drawerLed(drawer3)	OFF	OFF	OFF	OFF
<input type="checkbox"/> ^ M	isPillTaken(drawer1)			true	
<input type="checkbox"/> ^ C	isPillTobeTaken(drawer1)	false	true	true	false
<input type="checkbox"/> ^ C	isPillTobeTaken(drawer2)	false	false	false	false
<input type="checkbox"/> ^ C	isPillTobeTaken(drawer3)	false	false	false	false
<input type="checkbox"/> ^ M	pillDeadlineHit(drawer1)	true	false	false	
<input type="checkbox"/> ^ M	pillDeadlineHit(drawer2)	false	false	false	
<input type="checkbox"/> ^ M	pillDeadlineHit(drawer3)	false	false	false	

Fig. 5. AsmetaA Animation of the Pill-Box

Indeed, if the `isOff($drawer)` condition is removed and the deadline of the pill in the first drawer has passed, the first conditional rule sets `isPillTobeTaken(drawer1)` to true for all the following execution steps. However, if the LED for the drawer is ON and the user signals that the pill has been taken, the rule `r_reset` is executed and the location `isPillTobeTaken(drawer1)` is set to false. Thus, in the same step, the location is updated to two different values, leading to an inconsistent update.

6.1 Model Animation

The main disadvantage of the simulator is that it is textual, and this sometimes makes it difficult to follow the computation of the model. For this reason, ASMETA has a model animator, *AsmetaA* [11], which provides the user with complete information about all 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. A screenshot of *AsmetaA* is shown in Fig. 5. To execute the animator, the user clicks on the Δ icon in Eclipse.

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 the 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 the 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. 7), so that it can be re-executed whenever desired. Figure 5 shows the animation of the Pill-Box

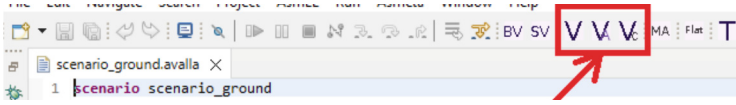


Fig. 6. AsmetaV commands

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.

7 Scenario-Based Validation

The **AsmetaS** and **AsmetaA** tools presented in the previous section require that the user executes the ASMETA model step by step or, at least, inserts some value to start the model simulation. In this section, we present the **AsmetaV** tool, which allows for performing scenario-based validation. Each *scenario* is a description of external actor actions and reactions of the system [18], which can be used to check the correct behavior of the model. Scenarios can be launched by using the button **V** shown in Fig. 6. Additionally, if the button **V_C** is pressed, **AsmetaV** keeps track of the rules covered by the scenario.

Scenarios are written in the **Avalla** language, and saved as **.avalla** files, as for the example reported in Listing 2 for the ASMETA ground model of the Pill-Box reported in Listing 1. The scenario models a simple assumption cycle for the pill in the first drawer. Initially, the Pill-Box has all the LEDs OFF, so no pill has to be taken (line 8-10). In the second step, we set the deadline for the pill in the first drawer as hit (line 17) and, after the execution of a step, the scenario checks whether the pill has been marked as one of those to be taken (line 20). Then, after a new execution step, we check that the LED corresponding to the first drawer is ON (line 25). Finally, after the patient has taken the pill, the scenario verifies whether all the LEDs have been turned OFF (line 35-37).

1	scenario scenario_ground	check isPillTobeTaken(drawer2) = false;	21
2	load pillbox_ground.asm	check isPillTobeTaken(drawer3) = false;	22
3		step	23
4	<i>// Initially all deadlines are not hit</i>	<i>// Check that the led for the drawer 1 is on</i>	24
5	set pillDeadlineHit(drawer1) := false;	check drawerLed(drawer1) = ON;	25
6	set pillDeadlineHit(drawer2) := false;	check drawerLed(drawer2) = OFF;	26
7	set pillDeadlineHit(drawer3) := false;	check drawerLed(drawer3) = OFF;	27
8	set isPillTaken(drawer3) := false;	check isPillTobeTaken(drawer1) = true;	28
9	set isPillTaken(drawer1) := false;	check isPillTobeTaken(drawer2) = false;	29
10	set isPillTaken(drawer2) := false;	check isPillTobeTaken(drawer3) = false;	30
11	step	<i>// Now, take the pill</i>	31
12	<i>// Check that all leds are off</i>	set isPillTaken(drawer1) := true;	32
13	check drawerLed(drawer1) = OFF;	step	33
14	check drawerLed(drawer2) = OFF;	<i>// Check that the led is reset</i>	34
15	check drawerLed(drawer3) = OFF;	check drawerLed(drawer1) = OFF;	35
16	<i>// Now, the time for the pill in the drawer 1 comes</i>	check drawerLed(drawer2) = OFF;	36
17	set pillDeadlineHit(drawer1) := true;	check drawerLed(drawer3) = OFF;	37
18	step	check isPillTobeTaken(drawer1) = false;	38
19	<i>// Check that pill is ready to be taken</i>	check isPillTobeTaken(drawer2) = false;	39
20	check isPillTobeTaken(drawer1) = true;	check isPillTobeTaken(drawer3) = false;	40

Listing 2. Example of Avalla scenario

Scenario Name. The first line of the scenario defines its name. For instance:

```
scenario scenario_ground
```

Unlike the ASMETA specification, the scenario name is not required to match the file name.

Loading AsmetaL Specifications. Each *Avalia* scenario is executed against an ASMETA spec. Thus, after having defined the scenario name it is essential to specify which ASMETA model to load. This is done by using the `load` command, followed by the relative or absolute path of the `.asm` file (including its extension):

```
load pillbox_ground.asm
```

Setting Monitored Functions. Monitored functions are read by the machine from the environment. When performing scenario-based validation, the user may supply the values for monitored or shared functions through the `set` command. These functions are then used as input signals to the system. For instance:

```
set pillDeadlineHit(drawer1) := false;
```

is used to set the monitored function `pillDeadlineHit` for the `drawer1` to `false`.

Step Execution. After having set the value for the monitored functions of interest, an ASMETA computation step (i.e., the reaction of the system) can be launched by using the `step` command. Additionally, *Avalia* supports the execution of multiple steps using the `stepUntil` command, until a specified Boolean condition becomes true.

Checking Controlled Functions. Executing an ASMETA specification step will lead to the update of the internal state of the ASMETA model. The `check` command is used to inspect property values in the current state of the underlying model. For instance:

```
check drawerLed(drawer1) = OFF;
```

checks that the controlled function `drawerLed` for the `drawer1` is `OFF`. When executing an *Avalia* scenario, the *AsmetaV* validator captures any check violation, and, if none occurs, it finishes with a “PASS” verdict (“FAIL” otherwise).

AsmetaL Code in Avalla Scenarios. Avalla scenarios support basic `set` commands. However, users may want to set ASMETA functions by using a more complex set of instructions, e.g., rules previously defined in the ASMETA specification or by parallelizing the update. Thus, scenarios allow for including **AsmetaL** commands with the `exec` keyword. For instance, the following **Avalla** code

```
set pillDeadlineHit(drawer1) := false;
set pillDeadlineHit(drawer2) := false;
set pillDeadlineHit(drawer3) := false;
```

can be replaced by

```
exec forall $drawer in Drawer do pillDeadlineHit($drawer) := false;
```

Note that this command would have been wrong if written in an **AsmetaL** specification, as `pillDeadlineHit` is a monitored function and it should not be set by the system. However, when **AsmetaV** simulates the scenario, a new ASMETA spec is created, and the monitored functions are converted to controlled ones, whose value is set to that specified in the **Avalla** scenario (either with a `set` command or with the `exec` command).

Scenario Modularization. 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` command when writing other scenarios that foresee the same sequence of Avalla commands.

Exporting and Animating Scenarios. Avalla scenarios can be exported from the **AsmetaA** tool, so that an animation session can automatically be repeated multiple times (see the “export to Avalla” button in Fig. 5). Similarly, **AsmetaV** supports the execution of scenarios through animation, by using the button V_A shown in Fig. 6. This allows users to control execution, enabling step-by-step scenario execution.

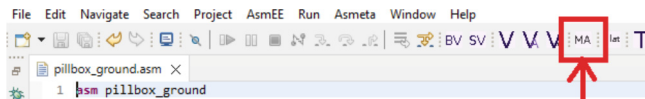


Fig. 7. AsmetaMA command

8 Model Review

ASMETA supports a form of static analysis of a model to automatically capture typical modeling errors such as inconsistent updates or dead specification parts

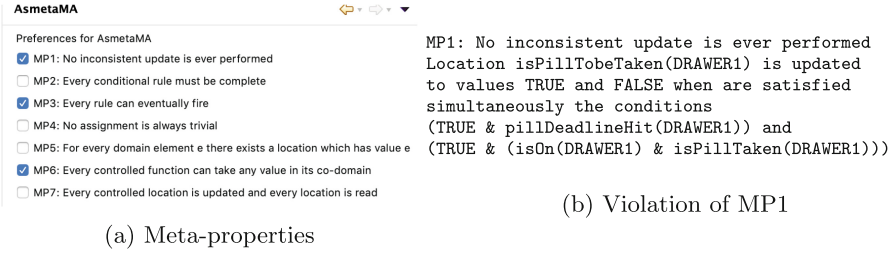


Fig. 8. AsmetaMA usage

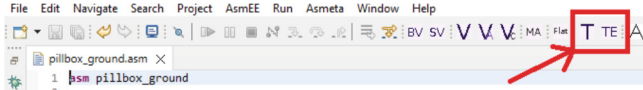


Fig. 9. AsmetaSMV command

(transition rules that are never triggered) due to overspecification. We called such a kind of static analysis about model quality *automatic model review* and it is carried out by the **AsmetaMA** tool [3], which can be executed by clicking on the button shown in Fig. 7. This 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. 9). Figure 8a shows the selection of the seven meta-properties in **AsmetaMA**. An example of meta-property is MP1, which checks the presence of inconsistent updates. Figure 8b reports an example of inconsistent update revealed by **AsmetaMA** on the same example reported in Sect. 6.

9 Formal Verification Through Model Checking

Besides validation, the ASMETA toolset supports the user in the properties' verification activity by the tool **AsmetaSMV** [2]. Properties are written in terms of propositional formulas over the machine's signature, preceded by the keyword `ctlspec` or `ltlspec`. For this purpose the libraries `CTLLibrary.asm` and `LTLLibrary.asm` must be imported, so for each CTL/LTL operator an equivalent **AsmetaL** Boolean-valued function is defined. The following example shows a CTL property (with the temporal operator $AG \Phi$ - globally Φ) for the Pill-Box ground model, i.e. a propositional formula that must hold in all reachable states:

```
ctlspec ag((forall $d in Drawer with isOn($d) implies (not areOthersOn($d))))
```

These properties are then automatically translated into a model of the symbolic model checker NuSMV [20], used to perform the verification. If the ASMETA model contains infinite or time domains, the NuXmv [19] model checker is preferred. The choice of the model checker is performed in Eclipse from the `ASMETA` \rightarrow **AsmetaSMV** preferences. The buttons shown in Fig. 9 are

used to verify the specification: the first button translates the specification into a model for the model checker without executing it, and the second translates and executes the specification using the selected model checker. The output of the model checker is pretty printed in terms of elements of the ASMETA signature. If the property is positively verified, the **AsmetaSMV** tool prints out on the Eclipse console that the property is **true**:

```
-- specification AG (((drawerLed(DRAWER1) = ON -> !areOthersOn(DRAWER1)) &
(drawerLed(DRAWER2) = ON -> !areOthersOn(DRAWER2))) & (drawerLed(DRAWER3) = ON
-> !areOthersOn(DRAWER3))) is true
```

Otherwise, assuming the property is false, it returns a counterexample. If we want to verify that a pill in **drawer1** is always taken when the pill deadline hits, we can write the following property:

```
ctlspec ag(pillDeadlineHit(drawer1) implies af(isOn(drawer1)))
```

When running the model checker, the property is false because it can happen that the pill in **drawer1** will never be taken (the function **isPillTaken(drawer1)** is never set to true), and the counterexample in Listings 3 is printed.

<pre>-- specification AG (pillDeadlineHit(DRAWER1) -> AF drawerLed(DRAWER1) = ON) is false -- as demonstrated by the following execution sequence Trace Description: CTL Counterexample Trace Type: Counterexample -> State: 1.1 <- pillDeadlineHit(DRAWER1) = false drawerLed(DRAWER1) = OFF drawerLed(DRAWER2) = OFF isPillTobeTaken(DRAWER2) = false drawerLed(DRAWER3) = OFF isPillTobeTaken(DRAWER3) = false ... -> State: 1.2 <- pillDeadlineHit(DRAWER1) = true pillDeadlineHit(DRAWER2) = true</pre>	<pre>--> State: 1.3 <- pillDeadlineHit(DRAWER1) = false isPillTobeTaken(DRAWER2) = true isPillTobeTaken(DRAWER1) = true pillDeadlineHit(DRAWER2) = false -> State: 1.4 <- ... pillDeadlineHit(DRAWER2) = true areOthersOn(DRAWER3) = true areOthersOn(DRAWER1) = true -> State: 1.5 <- drawerLed(DRAWER2) = OFF isPillTaken(DRAWER2) = false pillDeadlineHit(DRAWER2) = false areOthersOn(DRAWER3) = false areOthersOn(DRAWER1) = false</pre>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Listing 3. Counterexample generated by **AsmetaSMV**

10 Model Refinement

After having performed the activities presented in the previous sections, the model can be refined and the desired level of detail can be achieved. Here we report details of the two model refinements introduced in Sect. 4, and we highlight the differences between the ground model and the refined models.

10.1 Time Handling: **pillbox_time**

Modeling. The first model refinement we propose consists in explicitly modeling time passing, which is left abstract in the **pillbox_ground** model, by introducing the timer **tenMinutes** to capture the requirement stating that *the LED*

stays on for 10 min after the scheduled time to take the pill. Dealing with timers requires importing the predefined time library and setting a suitable timer as an element of the abstract domain `Timer`:

```
import ../STDLib/TimeLibrarySimple
static tenMinutes: Timer
```

The time library provides the user with several features to check whether a timer is expired (see the use of the predicate `expired(tenMinutes)` in rule `r_take` in the `pillbox_time` model to control the expiration of timer `tenMinutes`) and to reset a timer (see the use of the predefined rule `r_reset_timer[tenMinutes]` in rule `r_pillToBeTaken` of `pillbox_time` model). Using a timer always requires initializing the timer's duration and its starting time; in the `pillbox_time` model, the duration of the timer `tenMinutes` is set to 600 time unit (seconds, in this case) and its starting time is equal to the current time (e.g., taken as monitored value from the Java virtual machine).

```
function duration($t in Timer) = 600 // Timer initialization
function start($t in Timer) = currentTime($t)
// From the Time library
function currentTime($t in Timer) = mCurrTimeSecs
```

This model is an example of *vertical* model refinement, where concepts or behaviors previously left abstract are modeled in detail. Here the monitored function `pillDeadlineHit` is refined by the homonymous derived function that relates the time of a pill consumption with the current time. The value of the function `time_consumption` is set, for each pill/drawer, in the initialization section of the `pillbox_time` model, as follows:

```
function time_consumption($drawer in Drawer) = switch($drawer) // Initialization of the time consumption
case drawer1 : 60
case drawer2 : 2400
case drawer3 : 180
endswitch
```

The behavior of the rule `r_choosePillToTake` is refined by adding the new rule `r_pillToBeTaken` to turn on the led and reset the timer `tenMinutes` if the led is off. The behavior of rule `r_setOtherDrawers` is also refined by marking a pill to be taken if its time of consumption is reached and by resetting the timer (of a drawer with red led) if the pill has been taken or the timer of ten minutes waiting has expired.

Validation and Verification. As explained in the previous sections, V&V activities can be performed on this refinement level. Since this refinement step considers also the time during the simulation, the simulator (as well as the animator) handles the time using three different approaches for setting the monitored `mCurrTimeSecs` [10] (see Fig. 10): 1. time is read from the machine using the Java `TimeAPI`; 2. the user enters the value for time (like for monitored functions); 3.

Fig. 10. AsmetaS time simulation preferences

time is automatically increased at each step by a predefined value. Additionally, ASMETA allows the user to set the preferred time unit.

Regarding property verification, the NuSMV model checker does not support infinite domains (such as in the case of times), so the NuXmv [19] model checker must be used. However, its integration with ASMETA is still under development and not stable, thus, here we do not discuss its use.

10.2 Managing Multiple Pills: `pillbox_final`

Modeling. A further (and the last that we propose) vertical model refinement specifies the complete Pill-Box functionalities, allowing modeling the requirement that *Each drawer contains multiple slots (one for each pill) that are emptied in sequence*. To model this requirement we introduce the following controlled functions `time_consumption` and `drugIndex`:

dynamic controlled `time_consumption`: Drawer \rightarrow Seq(Integer)
dynamic controlled `drugIndex`: Drawer \rightarrow Natural

The former maps each drawer into a sequence of integers, containing the time deadlines expressed in seconds. The latter associates with each drawer an integer indicating the next slot to be emptied in the corresponding drawer. The two functions are initialized accordingly:

```
function time_consumption($drawer in Drawer) = switch($drawer) // Initialization of the time consumption
case drawer1 : [60, 1200, 1800]
case drawer2 : [2400, 3000, 3600]
case drawer3 : [180, 1200, 1800]
endswitch
function drugIndex($drawer in Drawer) = 0n // Every drawer has an index starting from 0
```

The derived function `pillDeadlineHit` is refined to check the pill's deadline in the drawer's current slot to be emptied¹⁵. The newly derived function `isThereAnyOtherDeadline` indicates if there is any other pill in the drawer to be taken. This information is used to refine the rule `r_setOtherDrawers`, which leads to suitably updating the drawer state (led status and drug index) by invoking the (nested and refined) macro call rule `r_reset`.

¹⁵ The function `at(sequence, i)` yield the value of the *i*th element of the sequence.

Listing 4. Header file

```

#define ANY String
#include <string.h>
#include <iostream>
#include <vector>
#include <set>
#include <map>
#include <list>
#include <chrono>
#include ".../STDL/TimeLibrarySimple.h"
using namespace std;
/* DOMAIN DEFINITIONS */
namespace pillbox_finalnamespace {
    class Drawer;
    enum LedLights { OFF, ON };
    enum Drugs { TYLENOL, ASPIRINE, MOMENT };
}
using namespace pillbox_finalnamespace;
class pillbox_finalnamespace::Drawer {
public:
    static set<Drawer*> elems;
    Drawer() { elems.insert(this); }
};
class pillbox_final : public virtual TimeLibrarySimple {
    /* DOMAIN CONTAINERS */
    const set<LedLights> LedLights_elems;
    const set<Drugs> Drugs_elems;
public:
    /* FUNCTIONS */
    map<Drawer*, bool> isPillTaken;
    map<Drawer*, LedLights> drawerLed[2];
    map<Drawer*, vector<int>> time_consumption[2];
    ...
    static Timer* tenMinutes;
    bool isOn (Drawer* param0.isOn);
    bool isOff (Drawer* param0.isOff);
    ...
    static Drawer* drawer1;
    ...
    /* RULE DEFINITION */
    void r_reset (Drawer* _drawer);
    void r_pillToBeTaken (Drawer* _drawer);
    ...
    void r_Main();
    pillbox_final();
    void initControlledWithMonitored();
    void getInputs();
    void setOutputs();
    void fireUpdateSet();
};

```

Listing 5. Cpp file

```

#include "pillbox_final.h"
using namespace pillbox_finalnamespace;
/* Conversion of ASM rules in C++ methods */
void pillbox_final::r_reset (Drawer* _drawer) {
    drawerLed[1][_drawer] = OFF;
    drugIndex[1][_drawer] = (drugIndex[0][_drawer] + 1);
    isPillTobeTaken[1][_drawer] = false;
}
void pillbox_final::r_pillToBeTaken (Drawer* _drawer) { ... }
void pillbox_final::r_ON (Drawer* _drawer) { ... }
void pillbox_final::r_choosePillToTake() { ... }
void pillbox_final::r_setOtherDrawers() { ... }
void pillbox_final::r_Main() {
    r_choosePillToTake();
    r_setOtherDrawers();
}
/* Static function definition */
bool pillbox_final::isOn(Drawer* _d) {
    return (drawerLed[0][_d] == ON);
}
bool pillbox_final::isOff(Drawer* _d) { ... }
bool pillbox_final::areOthersOn(Drawer* _d) { ... }
bool pillbox_final::pillDeadlineHit(Drawer* _d) { ... }
bool pillbox_final::isThereAnyOtherDeadline(Drawer* _d) { ... }
/* Function and domain initialization */
pillbox_final::pillbox_final(): LedLights_elems({ OFF, ON }),
    Drugs_elems({ TYLENOL, ASPIRINE, MOMENT }) {
    /* Init static functions Abstract domain */
    tenMinutes = new Timer;
    ...
    /* Function initialization */
    for(const auto& _drawer : Drawer::elems) {
        drawerLed[0].insert( { _drawer, OFF } );
        drawerLed[1].insert( { _drawer, OFF } );
    } ...
}
/* Apply the update set */
void pillbox_final::fireUpdateSet() {
    drawerLed[0] = drawerLed[1];
    time_consumption[0] = time_consumption[1];
    drug[0] = drug[1];
    drugIndex[0] = drugIndex[1];
    ...
}
/* init static functions and elements of abstract domains */
set< Drawer*> Drawer::elems;
Timer* pillbox_final::tenMinutes;
Drawer* pillbox_final::drawer1;
...

```

Remark. Model refinement must be proved to be *correct*, i.e., at each refinement step, a refined model must be proved to be a correct refinement of the abstract one. Due to lack of space and to keep this presentation easy to follow, here we skip the proof of correct refinement of models and the application of the **Asm-RefProver** supporting automatic proof of a particular form of model refinement.

11 From an ASMETA Model to Code

As requested by the best practices of model-driven engineering [16], the implementation of a system should be obtained from its model through a systematic model-to-code transformation. ASMETA features a set of tools allowing the automatic generation of C++ code [12] and C++ unit tests, and Java code [9].

In the following, we focus on using the **Asmeta2C++** tool. It generates C++ code (which is meant to be integrated with other artifacts or directly embedded in the final device) starting from an ASMETA model and, in particular, it produces two files: header (.h) and source (.cpp). The former contains the interface

of the source file and the translation of model domain declarations and definitions, function and rule declarations. The latter includes rules implementation, the functions and domain initialization, and the definitions of the functions. *Asmeta2C++* is only available as a command line tool and can be executed, in the case of the last refinement, with the following command:

```
java -jar Asmeta2Cpp.jar pillbox_final.asm
```

Additional options for the previous command are available in the Appendices file at <https://doi.org/10.5281/zenodo.12770854>.

An excerpt of the translation of the Pill-Box case study in C++ is shown in Listings 4 and 5, while the complete version of the source code is available in the replication package. An ASM run step involves executing the main rule and updating the locations. In C++, this is realized through two methods: `mainRule()` for translating the ASMETA main rule and `fireUpdateSet()` for updating locations to their next state values. *Asmeta2C++* can generate two additional files allowing to embedding the generated class into an Arduino program. Further insights into the translation of ASMETA rules and constructs into corresponding C++ instructions are given in [12].

12 Conclusion

This tutorial provides an overview of ASMETA, an integrated set of tools to describe the behavior of discrete event systems using the ASM formalism. The hands-on approach adopted in this tutorial shows how to combine all the model analysis techniques offered by ASMETA in order to start from a ground or partial specification of the system behavior, and then refine it incrementally into more complete models till leading to transformation to other external analysis models or code. Thanks to the adoption of a set of integrated and easy-to-use tools, like ASMETA, the effort for modeling and analysis with a formal method, like ASM, may be reduced and more software engineers may be convinced of applying the formal method for richer system design and more reliable systems.

Data Availability Statement. The artifacts for the tutorial paper are available at <https://doi.org/10.5281/zenodo.12770854>.

References

1. Arcaini, P., Bombarda, A., Bonfanti, S., Gargantini, A., Riccobene, E., Scandurra, P.: The ASMETA Approach to Safety Assurance of Software Systems, pp. 215–238. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-76020-5_13
2. 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.) Abstract State Machines, Alloy, B and Z, pp. 61–74. Springer, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11811-1_6

3. 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 (2010)
4. Arcaini, P., Gargantini, A., Riccobene, E.: SMT-based automatic proof of ASM model refinement. In: De Nicola, R., Kühn, E. (eds.) *Software Engineering and Formal Methods*, pp. 253–269. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-41591-8_17
5. Arcaini, P., Gargantini, A., Riccobene, E., Scandurra, P.: A model-driven process for engineering a toolset for a formal method. *Softw. Pract. Exper.* **41**, 155–166 (2011). <https://doi.org/10.1002/spe.1019>
6. ter Beek, M.H.: Formal methods and tools applied in the railway domain. In: Bonfanti, S., Gargantini, A., Leuschel, M., Riccobene, E., Scandurra, P. (eds.) *Rigorous State-Based Methods - 10th International Conference, ABZ 2024, Bergamo, Italy, June 25–28, 2024, Proceedings. Lecture Notes in Computer Science*, vol. 14759, pp. 3–21. Springer (2024). https://doi.org/10.1007/978-3-031-63790-2_1
7. ter Beek, M.H., et al.: Formal methods in industry. *Form. Asp. Comput.* (2024)
8. 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.) *Software Technology: Methods and Tools*, pp. 89–103. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-29852-4_7
9. Bombarda, A., Bonfanti, S., Gargantini, A.: From concept to code: unveiling a tool for translating abstract state machines into java code. In: *Rigorous State-Based Methods 10th International Conference, ABZ 2024, Bergamo, Italy, June 25–28, 2024, Proceedings, Lecture Notes in Computer Science*, vol. 14759. Springer (2024). https://doi.org/10.1007/978-3-031-63790-2_10
10. Bombarda, A., Bonfanti, S., Gargantini, A., Riccobene, E.: Extending ASMETA with time features. In: Raschke, A., Méry, D. (eds.) *Rigorous State-Based Methods*, pp. 105–111. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-77543-8_8
11. Bonfanti, S., Gargantini, A., Mashkoor, A.: ASMETA: animator for abstract state machines. In: Butler, M., Raschke, A., Hoang, T.S., Reichl, K. (eds.) *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pp. 369–373. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-91271-4_25
12. 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>
13. Börger, E.: The ASM refinement method. *Form. Asp. Comput.* **15**, 237–257 (2003)
14. Börger, E., Raschke, A.: *Modeling Companion for Software Practitioners*. Springer, Berlin, Heidelberg (2018). <https://doi.org/10.1007/978-3-662-56641-1>
15. Börger, E., Stärk, R.: *Abstract State Machines*. Springer, Berlin, Heidelberg (2003). <https://doi.org/10.1007/978-3-642-18216-7>
16. Brambilla, M., Cabot, J., Wimmer, M.: *Model-Driven Software Engineering in Practice*. Springer International Publishing (2017). <https://doi.org/10.1007/978-3-031-02549-5>
17. Broy, M., et al.: Does every computer scientist need to know formal methods? *Form. Asp. Comput.* (2024). <https://doi.org/10.1145/3670795>
18. 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.)

- Abstract State Machines, B and Z, pp. 71–84. Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87603-8_7
19. Cavada, R., et al.: The NUXMV symbolic model checker. In: Biere, A., Bloem, R. (eds.) *Computer Aided Verification*, pp. 334–342. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_22
 20. Cimatti, A., et al: NuSMV 2: an opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) *Computer Aided Verification*, pp. 359–364. Springer, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_29
 21. Garavel, H., Beek, M.H.t., Pol, J.V.D.: The 2020 expert survey on formal methods. In: *Formal Methods for Industrial Critical Systems: 25th International Conference, FMICS 2020, Vienna, Austria, September 2–3, 2020, Proceedings 25*, pp. 3–69. Springer (2020). https://doi.org/10.1007/978-3-030-58298-2_1
 22. Gleirscher, M., Marmsoler, D.: Formal methods in dependable systems engineering: a survey of professionals from Europe and North America. *Empir. Softw. Eng.* **25**(6), 4473–4546 (2020). <https://doi.org/10.1007/s10664-020-09836-5>
 23. Gleirscher, M., van de Pol, J., Woodcock, J.: A manifesto for applicable formal methods. *Softw. Syst. Model.* **22**(6), 1737–1749 (2023). <https://doi.org/10.1007/s10270-023-01124-2>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

