A Compositional Simulation Framework for Abstract State Machine Models of Discrete Event Systems

SILVIA BONFANTI, University of Bergamo, Bergamo, Italy
ANGELO GARGANTINI, University of Bergamo, Bergamo, Italy
ELVINIA RICCOBENE, Università degli Studi di Milano, Milano, Italy
PATRIZIA SCANDURRA, University of Bergamo, Bergamo, Italy

Modeling complex system requirements often requires specifying system components in separate models, which can be validated and verified in isolation from each other, and then integrating all components' behavior in order to validate the operation of the whole system. If models are executable, as for state-based formal specifications, engines to orchestrate the simulation of separate component operational models are extremely useful.

This paper presents an approach for the co-simulation, according to predefined orchestration schemas, of state-based models of separate components of a Discrete Event System. More precisely, we exploit the Abstract State Machine (ASM) formal method as state-based formalism, and we (i) define a set of operators to compose ASMs that communicate with each other through I/O events, and (ii) present an engine to execute the compositional simulation of the ASMs as a whole assembly.

As proof of concepts, we use a set of model examples of Discrete Event Systems of increasing complexity to show the application of our approach and to evaluate its effectiveness in co-simulating models of real systems.

CCS Concepts: • **Software and its engineering → Software as a service orchestration system**; *Formal methods*; • **Computing methodologies → Discrete-event simulation;**

Additional Key Words and Phrases: Models composition, models co-simulation, abstract state machines, ASMETA

ACM Reference Format:
Silvia Bonfanti, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. 2025. A Compositional Simulation Framework for Abstract State Machine Models of Discrete Event Systems. *Form. Asp. Comput.* 37, 2, Article 13 (February 2025), 26 pages. https://doi.org/10.1145/3652862

1 INTRODUCTION

Modern software-intensive systems are reaching unprecedented levels of complexity. Their modeling requires managing large-scale models, and the analysis and simulation of such models is often unmanageable and quite infeasible. Thus, to keep requirements complexity under control and to effectively manage large system specifications, it is necessary to decompose the conceptual

Authors' addresses: S. Bonfanti, A. Gargantini, and P. Scandurra, University of Bergamo, via Salvecchio 19, 24129, Bergamo, Italy; e-mails: silvia.bonfanti@unibg.it, angelo.gargantini@unibg.it, patrizia.scandurra@unibg.it; E. Riccobene, Università degli Studi di Milano, via Festa del Perdono 7, 20122, Milano, Italy; e-mail: elvinia.riccobene@unimi.it.

ACM 1433-299X/2025/02-ART13
https://doi.org/10.1145/3652862

model into components, each of which can be separately modeled, validated, and verified in isolation from the rest. Then all component models have to be integrated and possibly simulated as a whole [7, 27]. This decomposition approach is widely used for modeling and prototyping **Discrete Event Systems (DESs)**, and is the only alternative available in practice for understanding the behavior of such complex and large systems [5, 29].

Towards this research line and inspired by the well-known principles of decentralization and separation of concerns, this paper presents an approach for the co-simulation of state-based models of independent and interacting components of a DES. Specifically, we exploit the **Abstract State Machine (ASM)** [4, 11] formal method as state-based formalism. We define a set of coordination/orchestration operators, which allow the definition of patterns for compositional simulation of ASM models that communicate with each other through I/O events. We present the `AsmetaComp` simulation composer engine, which executes the compositional simulation of the I/O ASMs as *living models* at runtime [6, 25, 28].

The contributions of this paper can be summarized as follows:

- we propose a rigorous compositional modeling and simulation method for DESs through formal state-based models (as ASMs) supported by **V&V (validation and verification)** tools (like the ASMETA toolset for ASMs [1, 4]);
- we provide the concept of I/O ASM and define a compositional execution semantics of orchestrated I/O ASMs;
- we present the simulation composer engine `AsmetaComp` for the compositional simulation of I/O ASMs;
- we present the application of the proposed framework to safety-critical case studies;
- we compare the proposed compositional approach with a monolithic one where different component models are imported and scheduled in a unique model; comparison is given in terms of well-known principles of software architectural design and simulation results.

The paper extends the results presented in [8]. More precisely, (i) we expand the definition of the composition operators, enriching how I/O ASM models can be assembled, and (ii) we present the architecture and the usage of the `AsmetaComp` tool. Moreover, besides our experience reported in [8] in validating the proposed approach on the **Mechanical Ventilator Milano (MVM)**, a mechanical lung ventilator developed for Covid-19 patients, (iii) we here present the compositional model simulation by a set of examples spanning from simple models used to show the application of the composition operators, to a classical software engineering case study, the traffic light system, which is easy to follow by the reader, to another real case study from the medical domain, to show the effectiveness of our framework. Finally, (iv) we discuss the advantages of a compositional model simulation w.r.t. the simulation of a monolithic composition of models.

This paper is organized as follows. Section 2 provides basic concepts about ASMs and the ASMETA toolset. Section 3 presents the proposed compositional model-based simulation technique. Section 4 introduces the architecture and the usage of the `AsmetaComp` tool. Section 5 shows the application of the proposed technique to three case studies. Section 6 discusses the comparison between monolithic and composed simulations, and reports the threats to validity of our approach. Section 7 discusses the related works, and finally Section 8 concludes the paper.

2 BACKGROUND OF THE ASM FORMAL METHOD AND THE ASMETA TOOL-SET

The Abstract State Machine (ASM) is a formal method to specify behaviour in a state-based manner [10]. ASMs extend **Finite State Machines (FSMs)** where unstructured FSM control states are replaced by mathematical *algebras* comprising arbitrary complex data (i.e., domains of objects with functions defined on them), and state *transitions* are expressed by transition rules describing how

```
asm asmInc
import StandardLibrary
signature:
 monitored funcMulti: Integer
 monitored funcDec: Integer
 controlled maxFunction: Integer
 out funcInc: Integer
definitions:
 macro rule r_max =
  par
   if funcMulti >= funcDec then
       maxFunction := funcMulti endif
   if funcDec > funcMulti then
       maxFunction := funcDec endif
  endpar
 main rule r_Main =
  par
   funcInc := (funcMulti + funcDec) + 2
   r_max[]
  endpar

default init s0:
 function maxFunction = 0
```

Code 1. ASMETA model asmInc.asm.

```
asm asmDec
import StandardLibrary
signature:
 monitored funcInc: Integer
 monitored funcMulti: Integer
 out funcDec: Integer
 derived sumOfFunctions: Prod(Integer,Integer)-> Integer
definitions:
 function sumOfFunctions($f1 in Integer, $f2 in Integer) =
        $f1 + $f2
 main rule r_Main =
  funcDec := sumOfFunctions(funcInc, funcMulti) - 1
```

Code 2. ASMETA model asmDec.asm.

```
asm asmMulti
import StandardLibrary
signature:
 monitored myinput: Integer //unbound input
 out funcMulti: Integer
definitions:
 main rule r_Main =
  funcMulti := myinput *2
```

Code 3. ASMETA model asmMulti.asm.

the data (state function values saved into *locations*) change from one state to the next. **ASMETA (ASM mETAmodeling)** [1, 4] is a modeling tool-set based on the ASMs. In the following, we refer to an ASMETA model as an ASM encoded in the ASMETA tool-set.

Codes 1, 2, and 3 show three examples of simple ASMETA models that are here used to introduce the necessary preliminary concepts of ASMETA models and, in the following sections, to describe the compositional simulation of ASMETA models. The first model `asmInc.asm` (see Code 1) increases the sum of inputs by two and saves the maximum value between the two inputs, the second model `asmDec.asm` (see Code 2) decreases the sum of inputs by one, and the last model `asmMulti.asm` (see Code 3) multiplies the sum of inputs by two.

From these examples, you can see that an ASMETA model is composed by the following sections:

- The `signature` section, where domains and functions are declared.[1] The model interface with its environment is specified by *monitored* functions (e.g., function funcMulti in Code 1) that are written by the environment and read by the machine, and by *out* functions (e.g., function funcDec in Code 2) that are written by the machine and read by the environment; *controlled* functions (e.g., function maxFunction in Code 1) are the internal functions used by the machine (read in the current state and updated by the machine in the next state); *derived* functions (e.g., function sumOfFunctions in Code 2) are defined in terms of other (dynamic) functions.
- The `definitions` section where derived functions, all transition rules, and possible invariants are specified. *Transition rules* have different constructors depending on the structure of the updates they express, e.g., guarded updates (`if-then`, `switch-case`), simultaneous parallel updates (`par`), sequential updates (`seq`), and so on. E.g., Code 1 makes use of `if-then` and `par` transition rules: at the same time all the guards (mutually exclusive) are evaluated, and the corresponding update rule is executed when the condition is true. The *update* rule $f(t_1, \dots, t_n) := v$, being f an n-ary function, t_i terms, and v the new value of $f(t_1, \dots, t_n)$ in

[1] External libraries of a predefined (part of the) signature can be imported into a model as module, like the StandardLibrary containing built-in domains and functions to simplify the user declarations.

the next state, is the basic unit of rules construction. State *invariants* are first-order formulas that must be true in each computational state.

- The `main rule` that is, at each state, the starting point of the computation; it, in turns, calls all the other transitions rules, if any. For example, as shown in Code 1, the rule r_max, called by the main rule (in parallel to the update rule that assigns a new value to funcInc), assigns to the function maxFunction the maximum between the functions functMulti and funcDec, zero if they are equal.
- The `default init` section where initial values for the *controlled* functions are defined. An example is shown in Code 1, where the controlled function maxFunction is initialized to 0.

An ASMETA model can be understood as executable pseudo-code or virtual machine working over abstract data structures at any desired level of abstraction.

A model *run* is a finite or infinite sequence $S_0, S_1, \ldots, S_i, \ldots$ of states: starting from an initial state S_0, a *run step* from S_i to S_{i+1} consists of the parallel execution of all transition rules, which are directly or indirectly called from the main rule and are enabled to fire, and leads 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 by firing transition rules) or invariant violations, the model execution fails, but the model is kept alive by restoring the state in which it was before the failing step (*model roll-back*). In the sequel, we shortly write *model succeeds* or *fails* to mean a model performing either a successful run step or a failing one.

The development process from formal requirement specification to code generation is supported by the ASMETA set of tools [1], which provides the user with a modeling notation (as already shown in the code examples), different analysis (V&V) techniques, and automatic source code and test generators for models to be applied at design-, development-, and operation- time [4]. In particular, a runtime simulation engine, `AsmetaS@run.time` [26], has been developed within ASMETA as an extension of the offline simulator `AsmetaS` [14] to handle an ASM as a living model [6, 28] to run in tandem with a real software system.

3 COMPOSITIONAL SIMULATION OF ASM MODELS

When considering the decomposition of a conceptual system model into separate subsystems or components that interact to share resources, one has to analyze input/output events, since they represent the communication mechanism among components.

ASMs are a state-based formalism that relies on the current state and inputs of a machine to determine outputs. To define behavioral models of subsystems/components of a whole system, one can specify a set of ASMs, each with its own input (representing the monitored locations of the ASM), current state (representing the controlled locations of the ASM), and output (representing the out locations of the ASM). These component models, denoted here as I/O ASMs, are defined as follows:

Definition 3.1 (I/O ASM). An *I/O ASM* is an ASM model m with a set I_m of input (or monitored) functions and a set O_m of output (or out) functions in its signature, where I_m and O_m cannot be both empty. We denote by (I_m, m, O_m) an I/O ASM, and by curr_state(m) the set of its locations values.

We assume that I/O ASMs can interact in a black-box manner by binding input and output functions with the same symbol name and interpretation. Take, for instance, the standard arrangement or progression of two machines A and B, wherein the output of machine A serves as the input for machine B. We may view the cascade of these two machines as a single compound machine that reacts to an external input x by propagating the effect of x through the cascade of A and B at each

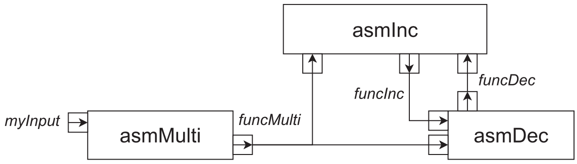


Fig. 1. I/O ASM assembly of simple components.

step (A reacts to x, then B reacts to the output of A). So we focus on ASMs having a well-defined I/O interface represented by (possibly parameterized) input and output ASM functions, which are, in fact, the interaction ports (or points) with the environment or other ASMs. We formally define the binding between I/O ASMs as follows:

Definition 3.2 (I/O ASM Binding). Given two I/O ASMs, m_i and m_j, an *I/O binding* exists from m_i to m_j iff $O_{m_i} \cap I_{m_j} \neq \emptyset$. We denote by $m_i \overset{B_{i,j}}{\Longrightarrow} m_j$ the I/O binding from m_i to m_j, where $B_{i,j} = O_{m_i} \cap I_{m_j}$ is the set of binding functions.

We assume that if an I/O ASM binding exists for the model m_i to m_j, then at least one function symbol f must occur in both the two models' signatures and is used as output function for m_i and as input function for m_j, with the same domain and codomain and same interpretation.

The `asmMulti` specification shown in Code 3 is an example of I/O ASM, having input function $I = \{\text{myInput}\}$ and output function $O = \{\text{funcMulti}\}$. Figure 1 provides a graphical view of the bindings among all component models, `asmInc.asm`, `asmDec.asm`, and `asmMulti.asm`, as wires (the solid lines with arrows) labeled with the name of the interface representing the binding functions (the exchanged signals values): an arrow incoming is the input for the component, an arrow outgoing is an output for the component.

Several I/O ASMs can execute and communicate over I/O bindings to form a whole ASM assembly.

Definition 3.3 (I/O ASM Assembly). An I/O ASM assembly is a set of I/O ASMs bound together by I/O ASM bindings.

Figure 1 illustrates, for example, an I/O ASM assembly consisting of the I/O ASM models `asmInc.asm`, `asmDec.asm`, and `asmMulti.asm`.

The execution of an assembly of I/O ASMs can be orchestrated (or coordinated) in accordance with a workflow expressible through different types of coordination constructs, defined below, with specific semantics. Intuitively, a pipe connection $m_1 \mid m_2$ means that the output of m_1 is used as input to m_2, assuming a directional I/O binding exists between the two models, namely some input functions of m_2 are a subset of the output functions of m_1. Similarly, a bidirectional pipe $m_1 <|> m_2$ is like having two pipes where one is used for the reverse direction, i.e., the output (or a subset) from m_2 becomes the input of m_1 in addition to external input from the environment or other machines bound to m_1. In both these two series compositions, we assume a cascade synchrony in reacting to external input from the environment. In a parallel connection $m_1 \parallel m_2$, both the two models react to external input from the environment or from other machines separately. Such coordination constructs allow for the following (recursive) definition of a *composition formula* of I/O ASMs.

Definition 3.4 (I/O ASM Composition Formula). A *composition* formula c over an ASM assembly A is a single I/O ASM m belonging to A or $c_1 \mid c_2$ or $c_1 <|> c_2$ or $c_1 <\|> c_2$ or $c_1 \parallel c_2$ or $if\ \phi\ then\ c_1\ else\ c_2$ or $while\ \phi\ c_1$, where c_1 and c_2 are composition formulas and $\{|, <|>, <\|>, \|, if-then-else, while\}$ are composition operators.

The composition formula $c = (m_1 <|> (m_2 \parallel m_3)) \mid m_4$ denotes, for example, the execution schema of four ASM models, where the output of the bidirectional pipe between *m1* and the parallel of *m2* and *m3*, is given as input to *m4* connected through a pipe. As another example, the I/O ASM assembly shown in Figure 1 can be executed using a composition formula made by a pipe between `asmMulti` and a half-duplex bidirectional pipe between `asmInc` and `asmDec`: $asmMulti.asm \mid (asmInc.asm <|> asmDec.asm)$. The intuitive meaning of this composition formula is that `asmMulti` is executed first, then it passes its output to both `asmInc` and `asmDec`. `asmInc` starts and passes its output to `asmDec` which executes its rules and gives back the result to `asmInc` for the next step.

Note that in case of a composition formula of the form $c_i op\, c_j$, we assume that if a model m occurs in c_i and also in c_j, two different instances of the same model m are instantiated.

Before providing the operational semantics of the composition operators, we extend the definition of I/O ASM binding at the level of a composition formula, as follows:

Definition 3.5 (I/O ASM Composition Binding). Given two I/O ASM composition formulas, c_1 and c_2, for a given I/O ASM assembly, an *I/O composition binding* $c_1 \overset{B_{1,2}}{\Longrightarrow} c_2$ exists from c_1 to c_2 iff $\exists$ at least an I/O ASM m_{1i} occurring in c_1 and an I/O ASM m_{2j} occurring in c_2 such that an I/O binding $m_{1i} \overset{B_{1i,2j}}{\Longrightarrow} m_{2j}$ exists from m_{1i} to m_{2j}. $B_{1,2}$ is the union of all existing binding functions between models m_{1i} in c_1 and models m_{2j} in c_2:

$$B_{1,2} = \bigcup_{m_{1i} \in c_1,\, m_{2j} \in c_2} B_{1i,2j}$$

As an example, consider the two composition formulas $c_1 = asmMulti.asm$ and $c_2 = asmInc.asm <|> asmDec.asm$ for the assembly shown in Figure 1. c_1 and c_2 are bound by the binding $B_{1,2} = B_{asmMulti,asminc} \cup B_{asmMulti,asmDec}$ where $B_{asmMulti,asminc} = \{\text{funcMulti}\}$ and $B_{asmMulti,asmDec} = \{\text{funcMulti}\}$.

Definition 3.6 (I/O ASM Composition Operators). Let c be an I/O ASM composition formula. The operational semantics of c is defined as follows:

Single model: $c = (I_m, m, O_m)$. A step of c is an execution step of the ASM m on the inputs I_m provided by the I/O bindings and by the environment.

(Simplex) pipe or sequence: $c = c_1 \mid c_2$. We assume $c_1 \overset{B}{\Rightarrow} c_2$. First execute c_1 on inputs I_{c_1} provided by the I/O bindings and by the environment, and if c_1 succeeds, subsequently execute c_2 on the inputs I_{c_2} provided by the I/O binding B with c_1, and by the environment, and return the results as outputs.

Half-duplex bidirectional pipe: $c = c_1 <|> c_2$. We assume $c_1 \overset{B_{1,2}}{\Longrightarrow} c_2$ and $c_2 \overset{B_{2,1}}{\Longrightarrow} c_1$. First execute c_1 on the inputs I_{c_1} provided by its I/O bindings and by the environment, and if c_1 succeeds, subsequently execute c_2 on the inputs I_{c_2} provided by the I/O binding $B_{1,2}$ with c_1 and by the environment; then, return the outputs for the I/O binding $B_{2,1}$ with c_1.

Full-duplex bidirectional pipe: $c = c_1 <\parallel> c_2$. We assume $c_1 \overset{B_{1,2}}{\Longrightarrow} c_2$ and $c_2 \overset{B_{2,1}}{\Longrightarrow} c_1$. First execute both c_1 and c_2 simultaneously on their inputs I_{c_1} and I_{c_2} provided by their I/O bindings and by the environment; if both succeed, then return their outputs for their I/O bindings.

Synchronous parallel split (or fork-join): $c = c_1 \parallel c_2$ Execute both c_1 and c_2 separately on their inputs I_{c_1} and I_{c_2}, respectively, provided by their I/O bindings and by the environment, then return their outputs.

Conditional execution: $c =$ **if** ϕ **then** c_1 **else** c_2 First evaluate the boolean expression ϕ on the current state of the I/O ASMs c_1 and c_2. If ϕ is true then execute c_1 on the inputs I_{c_1}

provided by its I/O bindings and by the environment, and if c_1 succeeds, return its result; otherwise, if c_1 fails, return fail and backtrack c_1 to its previous state. Else (ϕ is false) execute the I/O ASM c_2 on the inputs I_{c_2} provided by its I/O bindings and by the environment, and if c_2 succeeds, return its result; otherwise, if c_2 fails, return fail and backtrack c_2 to its previous state.

Iterative execution: $c =$ **while** $\phi\ c_1$ Execute repeatedly the I/O ASM c_1 on the inputs I_{c_1} provided by its I/O bindings and by the environment, so long as the boolean expression ϕ is true on the current state of c_1. If c_1 fails, return fail and backtrack c_1 to its original state before executing the loop.

In case an I/O ASM model fails, the composition expression fails and the faulty ASM model and all models already executed in the composition are rolled back.

The simulation of an I/O ASM assembly is the result of the compositional simulation of its I/O ASM components according to the execution semantics of a precise composition formula. Concretely, it can be represented and managed in memory in terms of an expression tree, as given by the following definition.

Definition 3.7 (Compositional Simulation Tree of I/O ASMs). Given a composition formula c of I/O ASMs m_i, $i = 1, \ldots, n$, a ***Compositional Simulation Tree (CST)*** of c is a binary tree $T_c = (V_c, E_c)$, where a leaf node in V_c is labelled by an ASM m_i together with its *curr_state(m_i)*, and an internal node in V_c is labeled by a composition operator chosen from the set {|, <|>, <||>, ||, if-then-else, while}.

Intuitively, a step of an I/O ASM composition (i.e., a single compositional simulation step) is a recursive pre-order traversal of the corresponding CST to visit nodes (from the root to leaves) and evaluates them according to their type. The *execCST* in Algorithm 1 is the pseudocode of a simplified version of a CST traversal without considering the roll-back of models in case a failure occurs during model execution. Given a CST T_c for a composition c, the output of the algorithm is the set O_c of ASM out functions values in the (final) current state of the I/O ASM models (executed at each leaf node of T_c). The recursive traversal *exec* in Algorithm 2 is initially invoked (see line 2 of Algorithm 1) on the root node of a not empty CST with an empty set I_c[2] of ASM input functions values for the I/O ASM models occurring in c. I_c will be populated during the tree traversal (when a model at a leaf node is executed) with function values provided externally by the environment or computed by other previously executed models in the tree. Algorithm 2 uses the subroutine $put(c_1, c_2, B_{1,2})$ to copy the values of the binding functions in $c_1 \overset{B_{1,2}}{\Longrightarrow} c_2$ from $B_{1,2} \subseteq O_{c_1}$ to $B_{1,2} \subseteq I_{c_2}$. An intuitive graphical representation of the execution steps of the various composition operators is depicted on the left side of Algorithm 2.

Definition 3.8 (Runstep of an I/O ASM Composition). Given a composition formula c for an assembly A of I/O ASMs m_i, $i = 1, \ldots, n$, a *runstep of the composition c* is the depth (pre-order) traversal, given by Algorithm 1: *execCST*, of the compositional simulation tree T_c, which updates the current states of the component models m_i at the leaves of T_c.

As an example, Figure 2 depicts the depth visit of the CTS of the composition $c = (m_1 < | > (m_2 \parallel m_3)) \mid m_4$: an execution step of the ASM models at leaf nodes of the tree updates the model's current state on the inputs provided by the I/O bindings (plus those provided by the environment but not shown in the picture). Labels 1, 4, and 7 denote the steps of the models to update their location

[2] We assume I_c is concretely realized as a map (or dictionary) that associates ASM function symbols (the keys) with their values.

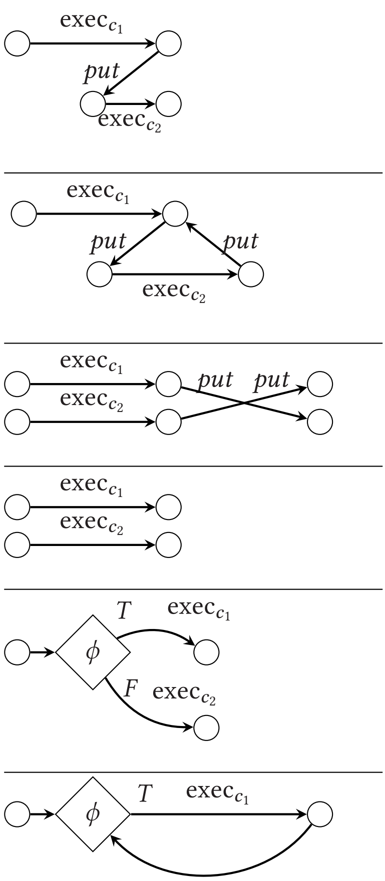
values, while labels 2, 3, 5, 6, and 8 denote copying of the binding function values according to the composition operators.

ALGORITHM 1: *execCST*

```
Input   : T: a CST
Function execCST(T):
    if $T.root \neq NULL$ then
        exec(T.root, $\emptyset$);
```

ALGORITHM 2: *exec*

```
Input   : node: a CST node
Input   : $I$: object that maps ASM monitored functions to their values.
Output  : $O$: an object that maps ASM out functions to their values.
Function exec(node, $I$):
    if node.isLeaf() then
        run a step of node.model on input $I$;
        return node.model.getOutLocations();
    else
        $c_1$ = node.left;
        if node.operator != while then
            $c_2$ = node.right;
        switch node.operator do
            case | do
                $O_1$ = exec($c_1$,$I$);
                put($c_1$,$c_2$,$B_{1,2}$);
                $O_2$ = exec($c_2$,$I$);
                return $O_1 \cup O_2$;
            case < | > do
                $O_1$ = exec($c_1$,$I$);
                put($c_1$,$c_2$,$B_{1,2}$);
                $O_2$ = exec($c_2$,$I$);
                put($c_2$,$c_1$,$B_{2,1}$);
                return $O_1 \cup O_2$ ;
            case < || > do
                $O_1$ = exec($c_1$,$I$);
                $O_2$ = exec($c_2$,$I$);
                put($c_1$,$c_2$,$B_{1,2}$);
                put($c_2$,$c_1$,$B_{2,1}$);
                return $O_1 \cup O_2$;
            case || do
                $O_1$ = exec($c_1$,$I$);
                $O_2$ = exec($c_2$,$I$);
                return $O_1 \cup O_2$ ;
            case if_then_else do
                if $\phi$ then $O$ = exec($c_1$,$I$);
                else $O$ = exec($c_2$,$I$);
                return $O$ ;
            case while do
                while $\phi$ do
                    $O$ = exec($c_1$,$I$);
                return $O$ ;
```



4 TOOL SUPPORT

AsmetaComp is a simulation composer engine for I/O ASMs. It has been developed using the Java OO programming language and is available as part of the ASMETA (ASM mETAmodeling) analysis toolset [1] for ASMs.

In this section, we show the general architecture and the use of the AsmetaComp tool.

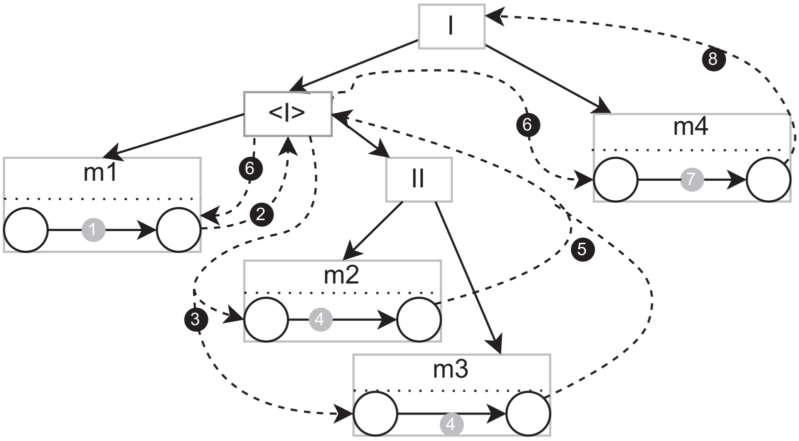


Fig. 2. Visit of a compositional simulation tree for $(m_1 <|> (m_2 \parallel m_3)) \mid m_4$.

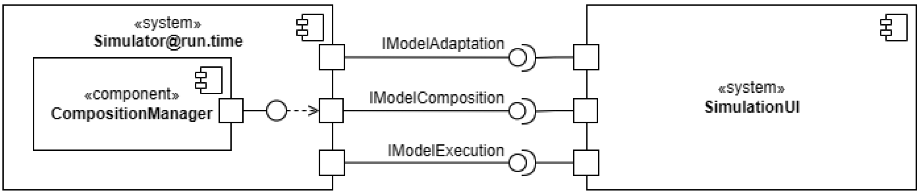


Fig. 3. Simulator@run.time and SimulationUI component diagram.

4.1 `AsmetaComp` Software Architecture

`AsmetaComp` is part of the `AsmetaS@run.time` [26] (see Figure 3). `AsmetaS@run.time` supports simulation as-a-service features (the interface `IModelExecution`) of the conventional ASMETA simulator `AsmetaS`, including model roll-back to the previous safe state after a failure of the model execution (e.g., invariant violations, inconsistent updates, ill-formed inputs, etc.) while processing an input event. It also allows the dynamic adaptation of a running ASMETA model (the interface `IModelAdaptation`) to change model invariants (representing, for example, system safety assertions). These functionalities are conveniently accessible by the user/modeler through the provided interfaces of `AsmetaS@run.time` and the dashboards for the Human-Model-Interaction (both in a graphical and in a command-line way) (the subsystem `SimulationUI`).

`AsmetaComp` is realized by the sub-component `CompositionManager` and by a CLI (see Section 4.2). The sub-component `CompositionManager` is responsible for implementing the new interface `IModelComposition` exposed by `AsmetaS@run.time` for allowing compositional simulation of ASMETA models. Figure 4 shows the class diagram for the internal details of such a component. `CompositionManager` is the core class of the component implementing the provided interface with all methods for managing the execution of a composition of ASMETA models (the attribute `compositionTree` of type `Composition`), and eventual exceptions that may arise during the compositional simulation, such as the rollback in case of a model execution fails (as represented by classes `CompositionException` and `CompositionRollbackException`).

Figure 5 shows an excerpt[3] of the class hierarchy introduced for representing the entities of a CST as (Java) OOP objects. The abstract class `Composition` is the root class for a generic node

[3] For the sake of simplicity, Figure 5 does not report the classes for the conditional and iterative operators.

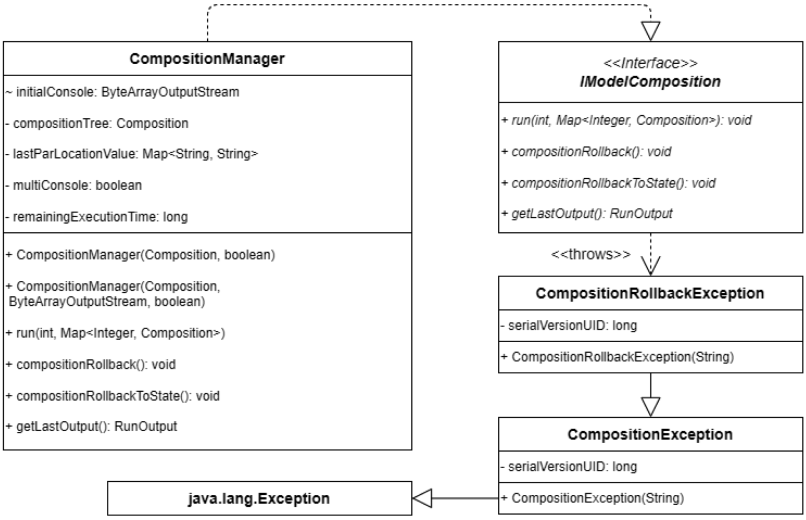

CompositionManager
~ initialConsole: ByteArrayOutputStream
- compositionTree: Composition
- lastParLocationValue: Map<String, String>
- multiConsole: boolean
- remainingExecutionTime: long
+ CompositionManager(Composition, boolean)
+ CompositionManager(Composition, ByteArrayOutputStream, boolean)
+ run(int, Map<Integer, Composition>)
+ compositionRollback(): void
+ compositionRollbackToState(): void
+ getLastOutput(): RunOutput
<<Interface>>
IModelComposition
+ run(int, Map<Integer, Composition>): void
+ compositionRollback(): void
+ compositionRollbackToState(): void
+ getLastOutput(): RunOutput
<<throws>>
CompositionRollbackException
- serialVersionUID: long
+ CompositionRollbackException(String)
CompositionException
- serialVersionUID: long
+ CompositionException(String)
java.lang.Exception

Fig. 4. CompositionManager class diagram.

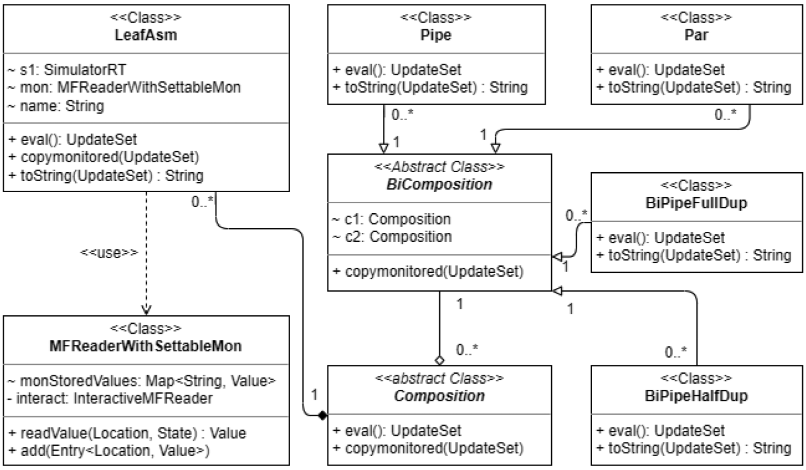


Fig. 5. Composition class diagram.

of the CST representing a composition (sub-)formula. Its direct subclass `BiComposition` is abstract and represents a generic node for a binary composition operator; its concrete subclasses (`BiPipeHalfDup`, `BiPipeFullDup`, etc.) represent the nodes for the binary composition operators. The class `LeafAsm` represents a leaf node of a CST, namely a living I/O ASM; it includes a `name`, a reference object to the run-time simulator instance (attribute of type `SimulatorRT`) where the ASM is running, and a driver (attribute object of type `MFReaderWithSettableMon`) for the interaction with the environment during the input reading. All these classes expose the methods `eval()` and `copyMonitored(UpdateSet)` representing the recursive traversal *exec* and the subroutine *put*, respectively, used in the Algorithm 2. So, according to the GoF *Visitor* design pattern, each of these concrete classes implements part of the visit algorithm operating on the object structure and maintains the state of the algorithm locally.

```
1 init −n 3
2 setup comp as asmMulti.asm | (asmInc.asm <|> asmDec.asm)
3 run(comp, {myinput=2})
4 run(comp, {myinput=3})
5 run(comp, {myinput=5}) ...
```

Code 4. Composition script.

```
1  ========= Compositional step 1 =========
2  [step:1 of asmMulti]
3  Model execution outcome: SAFE
4  Updated locations: {funcMulti=4}
5  Out locations: {funcMulti=4}
6  [step:1 of asmInc]
7  Model execution outcome: SAFE
8  Updated locations: {funcInc=6; maxFunction=4}
9  Out locations: {funcInc=6}
10 [step:1 of asmDec]
11 Model execution outcome: SAFE
12 Updated locations: {funcDec=9}
13 Out locations: {funcDec=9}
14 ========= Compositional step 2 =========
15 [step:2 of asmMulti]
16 Model execution outcome: SAFE
17 Updated locations: {funcMulti=6}
18 Out locations: {funcMulti=6}
19 [step:2 of asmInc]
20 Model execution outcome: SAFE

Updated locations: {funcInc=17; maxFunction=9}        21
Out locations: {funcInc=17}                           22
[step:2 of asmDec]                                    23
Model execution outcome: SAFE                         24
Updated locations: {funcDec=22}                       25
Out locations: {funcDec=22}                           26
                                                      27
========= Compositional step 3 =========              28
[step:3 of asmMulti]                                  29
Model execution outcome: SAFE                         30
Updated locations: {funcMulti=10}                     31
Out locations: {funcMulti=10}                         32
[step:3 of asmInc]                                    33
Model execution outcome: SAFE                         34
Updated locations: {funcInc=34; maxFunction=22}       35
Out locations: {funcInc=6}                            36
[step:3 of asmDec]                                    37
Model execution outcome: SAFE                         38
Updated locations: {funcDec=43}                       39
Out locations: {funcDec=43}                           40
```

Code 5. Output of the compositional simulation of the three simple models.

4.2 AsmetaComp Usage

AsmetaComp can be used through a command line, as explained below. An orchestration language allows the definition of composition schemas for simulating local or remote models[4] according to the compositional semantics of the basic operators.

AsmetaComp *Shell.* The user can provide input interactively or via a script containing a sequence of commands for setting and running the composition formula. In this example, we have adopted the second option as shown in Code 4. After the number of involved models is defined (Code 4, line 1), we have built a pipe from the asmMulti to the half-duplex bidirectional pipe of asmInc and asmDec (Code 4, line 2). Then, in the following lines, the run command denotes when a composition step is performed and which are the values of inputs (monitored functions).

The output of the composition is shown in Code 5, and the result reflects the expected behavior. Given the input myinput = 2 to asmMulti (Code 4, line 3), the value of funcMulti is the input value multiplied by 2 (funcMulti = 4 - Code 5, line 5). Then, asmInc is executed with the output of asmMulti specification, funcMulti = 4. The value of funcDec is the output of asmDec specification in the previous step (initially, it assumes the value assigned in the initialization section). After the first execution step, the output of asmInc is funcInc = 6 (Code 5, line 9). Finally, to conclude the first composition step, the asmDec specification is executed, its output is funcDec = 9 (Code 5, line 13). The outputs of the first composition steps are the inputs of the next composition step.

[4] Model files within a domain are located through a http-based URL path.

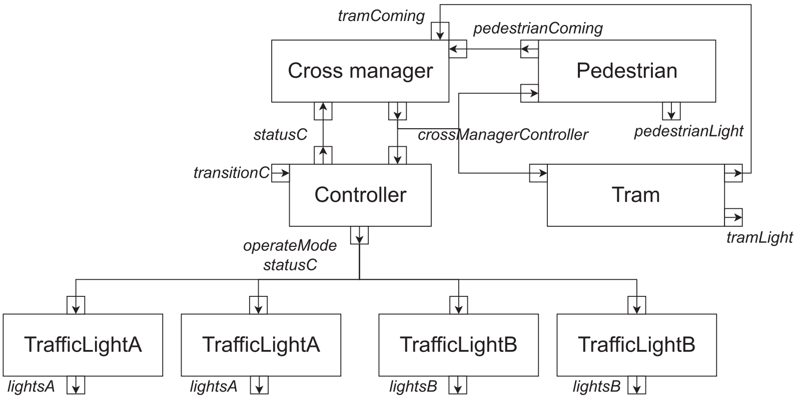


Fig. 6. Main components of the Traffic Light Cross Manager system and signals exchanged.

5 ASMETACOMP VALIDATION

In this section, we present the validation of the AsmetaComp tool by three use cases[5]: the Traffic Light Cross Manager, the **Medicine Reminder and Monitoring System (MRM)**, and the **Mechanical Ventilator Milano (MVM)**. This latter case study [3], which motivated us to develop the tool, has already been presented in the conference paper [8] and here is reported to point out the main findings.

5.1 Traffic Light Cross Manager

The traffic light cross manager presented in this paper manages a two-way traffic street intersection made up of four traffic lights for cars, a traffic light for pedestrians, and a traffic light for trams. The cross manager can be into three distinct states: PEDESTRIAN when a pedestrian is required to cross the intersection, TRAM when a tram is required to cross the intersection, NORMAL when other vehicles are allowed to cross the intersection. A controller manages the four traffic lights for cars. The traffic lights can be *off*, *yellow blinking*, *red light on*, *green light on* or *yellow light on*. Initially, the controller and the traffic lights are *off*. When the *turn on* command is received by the controller, the traffic lights are set to *yellow blinking* until the *operate* command is received. When the controller is in *operate* state, six modes are identified: the traffic lights are *red*, two are *red*, and the others are *green*, two are *red*, and the others are *yellow*. The transition between red, *green* and *yellow* is managed by a timer. When the *standby* command is received, the traffic lights are set to *yellow blinking* and when the controller is *turned off*, the traffic lights are turned *off* as well.

If a pedestrian or a tram has to cross the intersection, the cross manager registers the event and then sets its status to PEDESTRIAN or TRAM. Then the controller sets the traffic lights for cars to *red*. After that, the pedestrian/tram traffic light is set to *green* and the pedestrian/tram can cross the intersection (in all directions). The pedestrian/tram events are managed by the cross manager only when the controller is in *operate* state, otherwise, the corresponding traffic light is set to *red*.

Figure 6 shows the I/O ASM assembly of the overall traffic light cross manager system using an informal graphical notation. Code 6 reports, instead, an excerpt of one of the component models: the ASM model of the traffic light. For example, transitionC (see Figure 6) is the input command received by the controller to change its status, while the controller status and mode (statusC and

[5] AsmetaComp and all model examples presented here are available as part of the experimental project AsmetaS@run.time of ASMETA [1].

```
asm trafficlightA
signature:
 enum domain Lights = {ALL_OFF | RED | BLINK_YELLOW |
        GREEN | YELLOW}
 enum domain ControllerStatus = {CONTR_OFF | STANDBY |
        OPERATE}
 enum domain ControllerOperateMode = {RED_AB |
        GREEN_A_RED_B | YELLOW_A_RED_B | RED_BA |
        GREEN_B_RED_A | YELLOW_B_RED_A}
 out lightsA: Lights
 monitored statusC: ControllerStatus
 monitored operateMode : ControllerOperateMode

definitions:
 rule r_operateSubState =
    par
        if operateMode = RED_AB or operateMode = RED_BA or
              operateMode = GREEN_B_RED_A or operateMode =
              YELLOW_B_RED_A then
            lightsA := RED
        endif
        if operateMode = GREEN_A_RED_B then
            lightsA := GREEN
   endif
   if operateMode = YELLOW_A_RED_B then
    lightsA := YELLOW
   endif
  endpar

main rule r_Main =
  par
   if statusC = CONTR_OFF then
    lightsA := ALL_OFF
   endif
   if statusC = STANDBY then
    lightsA := BLINK_YELLOW
   endif
   if (statusC = OPERATE) then
     r_operateSubState[]
   endif
  endpar

default init s0:
 function lightsA = ALL_OFF
```

Code 6. ASM model of the Traffic Light.

```
1  init −n 8
2  setup comp as (pedestrian || tram) <|> crossManager <||> controller | (trafficlightA || trafficlightA || trafficlightB || trafficlightB)
3  run(comp, {crossManagerController=NORMAL;newPedestrianComing=false;newTramComing=false;transitionC=TURN_O N})
4  run(comp, {newPedestrianComing=false;newTramComing=false;transitionC=OPERATE_T;mCurrTimeSecs=2})
5   ...
6  run(comp, {newPedestrianComing=true;newTramComing=false;mCurrTimeSecs=6;transitionC=OPERATE_T})
7  run(comp, {newTramComing=false;mCurrTimeSecs=7;transitionC=OPERATE_T}) ...
```

Code 7. Composition script of the Traffic Light Cross Manager system.

operateMode) are the input for the traffic light models (see Code 6). The outputs of the controller are the input for the traffic lights, and the output lights values are respectively lightsA and lightsB for the two traffic lights (see Figure 6). The `definitions` section in Code 6 contains the transition rules r_operateSubState and r_Main.

Compositional model simulation. By the `AsmetaComp` shell script reported in Code 7 we establish the composition formula (using the setup command at line 2) made by four operator types (pipe, half-duplex, full-duplex and synchronous parallel split):

$$(pedestrian||tram) < | > crossManager < || > controller|(trafficlightA||trafficlightA||trafficlightB||trafficlightB)$$

The cross manager takes as input the controller status, the pedestrian and tram call, and determines the status of the crossing. Based on the crossing status, the pedestrian and tram traffic lights are set to the expected color. In case both calls arrive at the same time, the pedestrian has priority. Then, the traffic light controller, based on defined rules, sets simultaneously the traffic lights. Then, we express some compositional steps of the interacting models. This includes inputs from the environment for the unbound ASM monitored functions of models (through the run commands). The resulting composition output is shown in Code 8. At each step, the user sets the value of monitored functions, and the models are executed by following the composition formula. When a pedestrian has to cross the road, i.e., newPedestrianComing=true (see Code 7, line 6), the cross manager moves to state PEDESTRIAN (see Code 8, line 60). In the next step, the pedestrian traffic light is set to GREEN (see Code 8, line 79), while the traffic lights are set to RED (see Code 8, line 88).

```
========= Compositional step 1 =========
[step:1 of pedestrian]
Model execution outcome: SAFE
Out locations: {pedestrianComing=false, pedestrianLight=RED}
[step:1 of tram]
Model execution outcome: SAFE
Out locations: {tramLight=RED, tramComing=false}
[step:1 of crossManager]
Model execution outcome: SAFE
Out locations: {crossManagerController=NORMAL}
[step:1 of controller]
Model execution outcome: SAFE
Out locations: {statusC=STANDBY}
[step:1 of trafficlightA]
Model execution outcome: SAFE
Out locations: {lightsA=BLINK_YELLOW}
[step:1 of trafficlightA]
Model execution outcome: SAFE
Out locations: {lightsA=BLINK_YELLOW}
[step:1 of trafficlightB]
Model execution outcome: SAFE
Out locations: {lightsB=BLINK_YELLOW}
[step:1 of trafficlightB]
Model execution outcome: SAFE
Out locations: {lightsB=BLINK_YELLOW}
========= Compositional step 2 =========
[step:2 of pedestrian]
Model execution outcome: SAFE
Out locations: {pedestrianComing=false, pedestrianLight=RED}
[step:2 of tram]
Model execution outcome: SAFE
Out locations: {tramLight=RED, tramComing=false}
[step:2 of crossManager]
Model execution outcome: SAFE
Out locations: {crossManagerController=NORMAL}
[step:2 of controller]
Model execution outcome: SAFE
Out locations: {operateMode=RED_AB;statusC=OPERATE}
[step:2 of trafficlightA]
Model execution outcome: SAFE
Out locations: {lightsA=RED}
[step:2 of trafficlightA]
Model execution outcome: SAFE
Out locations: {lightsA=RED}
[step:2 of trafficlightB]
Model execution outcome: SAFE
Out locations: {lightsB=RED}
[step:2 of trafficlightB]
Model execution outcome: SAFE
Out locations: {lightsB=RED} ...
========= Compositional step 6 =========
[step:6 of pedestrian]
Model execution outcome: SAFE
Out locations: {pedestrianComing=true, pedestrianLight=RED}
[step:6 of tram]
Model execution outcome: SAFE
Out locations: {tramLight=RED, tramComing=false}
[step:6 of crossManager]
Model execution outcome: SAFE
Out locations: {crossManagerController=PEDESTRIAN}
[step:6 of controller]
Model execution outcome: SAFE
Out locations: {operateMode=GREEN_A_RED_B;statusC=
        OPERATE}
[step:6 of trafficlightA]
Model execution outcome: SAFE
Out locations: {lightsA=GREEN}
[step:6 of trafficlightA]
Model execution outcome: SAFE
Out locations: {lightsA=GREEN}
[step:6 of trafficlightB]
Model execution outcome: SAFE
Out locations: {lightsB=RED}
[step:6 of trafficlightB]
Model execution outcome: SAFE
Out locations: {lightsB=RED}
========= Compositional step 7 =========
[step:6 of pedestrian]
Model execution outcome: SAFE
Out locations: {pedestrianComing=true, pedestrianLight=
        GREEN}
[step:6 of tram]
Model execution outcome: SAFE
Out locations: {tramLight=RED, tramComing=false}
[step:6 of crossManager]
Model execution outcome: SAFE
Out locations: {crossManagerController=PEDESTRIAN}
[step:6 of controller]
Model execution outcome: SAFE
Out locations: {operateMode=RED_BA;statusC=OPERATE}
[step:6 of trafficlightA]
Model execution outcome: SAFE
Out locations: {lightsA=RED}
[step:6 of trafficlightA]
Model execution outcome: SAFE
Out locations: {lightsA=RED}
[step:6 of trafficlightB]
Model execution outcome: SAFE
Out locations: {lightsB=RED}
[step:6 of trafficlightB]
Model execution outcome: SAFE
Out locations: {lightsB=RED}
```

Code 8. Composition output of the Traffic Light Cross Manager system.

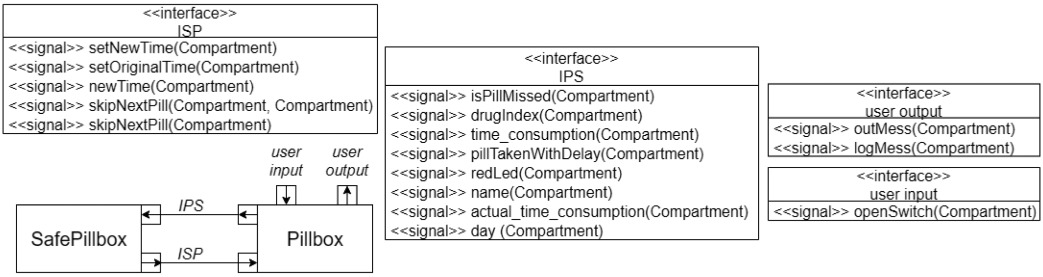
5.2 The Medicine Reminder and Monitoring System

Medicine Reminder and Monitoring (MRM) System controls a smart medicine dispenser for medicine administration. To secure patient safety, a component is responsible to manage critical situations by applying corrective actions if possible. The ASM I/O assembly of the MRM system is shown in Figure 7, where all exchanged signals are specified between the component `Pillbox` that represents the medicine dispenser, and the component `SafePillbox` in charge of managing missed pills.

The medicine dispenser is manually filled with the medicines prescribed (one medicine type per each compartment) based on the doctor's prescription. Then, the caregiver uploads to the medicine dispenser a drug file record containing information about the medicines prescribed by the doctor: medicine name, number of doses per day, time schedule, minimum separation (in terms of time) from the medicine M to the interferer N and between the same medicine, and delta time added to the original time schedule to remember the medicine again if a dose is missed. Once everything is correctly set up, the patient is notified by the `Pillbox` when a medicine has to be taken: an audible

```
rule r_pillOnTime($compartment in Compartment)=
    if isPillMissed($compartment) then
       if rescheduleNotOverlap($compartment) then
          par
             setNewTime ($compartment, drugIndex($compartment)):= true
             newTime ($compartment, drugIndex($compartment)):= at(time_consumption($compartment),drugIndex($compartment))+
                     deltaDelay(name($compartment))
          endpar endif endif
```

Code 9. Small snippet of SafePillbox: reschedule the pill when missed, if possible.



<<interface>>
ISP
<<signal>> setNewTime(Compartment)
<<signal>> setOriginalTime(Compartment)
<<signal>> newTime(Compartment)
<<signal>> skipNextPill(Compartment, Compartment)
<<signal>> skipNextPill(Compartment)
<<interface>>
IPS
<<signal>> isPillMissed(Compartment)
<<signal>> drugIndex(Compartment)
<<signal>> time_consumption(Compartment)
<<signal>> pillTakenWithDelay(Compartment)
<<signal>> redLed(Compartment)
<<signal>> name(Compartment)
<<signal>> actual_time_consumption(Compartment)
<<signal>> day (Compartment)
<<interface>>
user output
<<signal>> outMess(Compartment)
<<signal>> logMess(Compartment)
<<interface>>
user input
<<signal>> openSwitch(Compartment)
user input
user output
IPS
ISP
SafePillbox
Pillbox

Fig. 7. Main components of the MRM system and signals exchanged.

alarm of the pill box sounds, a red LED, corresponding to the compartment where the medicine to be taken is located, is turned on, and the compartment is unlocked. The patient/caregiver has to open and then close the compartment to report that the medicine was effectively taken. If the pill is not taken after 20 minutes, it is considered missed. A missed pill is rescheduled by the `SafePillbox` by adding the delta time to the original time schedule only if the minimum time window from the next medicine is observed, otherwise it is considered definitely missed. Code 9 reports a snippet of the `SafePillbox` component: when a pill is missed (the condition isPillMissed is true) and there is guaranteed to be no overlap with the next medicine (the condition rescheduleNotOverlap is true), the current pill is rescheduled by computing its new assumption time.

Compositional model simulation. Once the interfaces and the specification have been defined, the composition formula set for the MRM system is: `pillbox.asm <|> safePillbox.asm`, as reported in Code 10 (we perform an initialization step of the `SafePillbox` before starting the composition). This trace simulates the scenario of the pill missed (the compartment is never opened - openSwitch(compartment)=false) and rescheduled. The output of the `AsmetaComp` is shown in Code 11. At step 6 the pill is missed (outMess(comparment2)=moment missed - Code 11, line 53), and in step 7 it is rescheduled by the `SafePillbox` component, which updates the assumption time of the missed pill (Code 11, line 59).

5.3 Mechanical Ventilator Milano

MVM is a mechanical lung ventilator composed of two main software components: the controller and the supervisor. The *controller* sets the input and output valves to allow the patient to breathe and manages the lung ventilation based on user inputs (acquired through the *GUI* component) and patient parameters. The *supervisor* software monitors the overall system behavior and ensures that the machine *HW* operates safely. As explained in [8], when composing the controller and the supervisor, when the supervisor detects a failure in the controller it leads the system in a fail-safe state (the input valve is closed, and the output valve is opened) to put the patient in a safe situation as required by the specification.

```
init −n 2
setup Pillbox.asm SafePillbox.asm
run(SafePillbox.asm)
setup pillboxComposition as Pillbox.asm <|> SafePillbox.asm
run(pillboxComposition, {openSwitch(compartment1)=false;openSwitch(compartment2)=false})
run(pillboxComposition, {openSwitch(compartment1)=true;openSwitch(compartment2)=false})
run(pillboxComposition, {openSwitch(compartment1)=true;openSwitch(compartment2)=false})
run(pillboxComposition, {openSwitch(compartment1)=false;openSwitch(compartment2)=false})
run(pillboxComposition, {openSwitch(compartment1)=false;openSwitch(compartment2)=false})
run(pillboxComposition, {openSwitch(compartment1)=false;(openSwitch(compartment2)=false})
run(pillboxComposition, {openSwitch(compartment1)=false;openSwitch(compartment2)=false})
```

Code 10. Composition script of the MRM system.

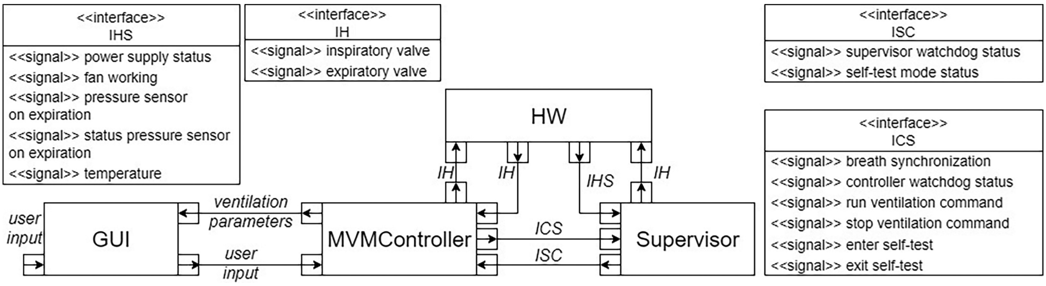

<<interface>>
IHS
<<signal>> power supply status
<<signal>> fan working
<<signal>> pressure sensor on expiration
<<signal>> status pressure sensor on expiration
<<signal>> temperature
<<interface>>
IH
<<signal>> inspiratory valve
<<signal>> expiratory valve
<<interface>>
ISC
<<signal>> supervisor watchdog status
<<signal>> self-test mode status
<<interface>>
ICS
<<signal>> breath synchronization
<<signal>> controller watchdog status
<<signal>> run ventilation command
<<signal>> stop ventilation command
<<signal>> enter self-test
<<signal>> exit self-test
HW
IH
IH
IHS
IH
user input
GUI
ventilation parameters
user input
MVMController
ICS
ISC
Supervisor

Fig. 8. Main components of the MVM system and signals exchanged.

Figure 8 shows a high-level overview of the main architecture components of the MVM system using an informal graphical notation.

Compositional model simulation. The script reported in Code 12 is used to set the compositional formula (see line 2) made by two bidirectional pipes: Hardware.asm <|> (MVMController .asm <|> Supervisor.asm) and compositional steps of the interacting MVM models. The hardware sends values to both the controller and supervisor and both entities respond by providing the hardware configuration based on the ventilation status (bidirectional pipe between hardware and controller/-supervisor). Additionally, the controller communicates information such as its status and alarms raised to the supervisor, which returns its status to it (bidirectional pipe between controller and supervisor). The script provides input functions for each run to simulate a normal operation.

Code 13 reports the output of the simulation trace. In this simulation, after starting the ventilation, the supervisor detects an error in the input valve status when entering in expiration phase (step 5). The valve is set to OPEN instead of CLOSED. To protect the patient, the supervisor sets the ventilator into fail-safe mode by setting the input valve closed and the output valve open (step 6). To detect this malfunction in the controller, we injected an error manually in the model, and we found that the supervisor was able to detect it and act properly.

6 DISCUSSION

In this section, we provide a more extended analysis of the potential advantages of the proposed compositional approach over the traditional *monolithic* one, i.e., importing the different submachines in a unique model and scheduling them by suitable ASM rule constructors. Comparison is given in terms of well-known architectural design principles [19] (Section 6.1) and simulation results (Section 6.2). We also discuss major strengths and limitations of our approach as we perceived and observed concretely during the development of the presented case studies (Section 6.3), as well as how threats to validity have been mitigated (Section 6.4).

```
========= Compositional step 1 =========
[step:1 of SafePillbox]
Model execution outcome: SAFE
Updated locations: {}
[step:1 of pillbox]
Model execution outcome: SAFE
Updated locations: {outMess(compartment1)=Take fosamax, redLed(
        compartment2)=OFF, time_consumption(compartment1)=[360],
        time_consumption(compartment2)=[730,1140], redLed(
        compartment1)=ON, drugIndex(compartment1)=0, isPillMissed(
        compartment1)=false, pillTakenWithDelay(compartment2)=
        false, systemTime=361, pillTakenWithDelay(compartment1)=
        false, drugIndex(compartment2)=0, name(compartment2)="
        moment", name(compartment1)="fosamax", isPillMissed(
        compartment2)=false, day=0}
Out locations: {outMess(compartment1)=Take fosamax, redLed(
        compartment2)=OFF, time_consumption(compartment1)=[360],
        time_consumption(compartment2)=[730,1140], redLed(
        compartment1)=ON, drugIndex(compartment1)=0, isPillMissed(
        compartment1)=false, pillTakenWithDelay(compartment2)=
        false, systemTime=361, pillTakenWithDelay(compartment1)=
        false, drugIndex(compartment2)=0, name(compartment2)="
        moment", name(compartment1)="fosamax", isPillMissed(
        compartment2)=false, day=0}

========= Compositional step 2 =========
[step:2 of SafePillbox]
Model execution outcome: SAFE
Updated locations: {}
[step:2 of pillbox]
Model execution outcome: SAFE
Updated locations: {outMess(compartment1)=Close fosamax in 10
        minutes, redLed(compartment1)=BLINKING}
Out locations: {outMess(compartment1)=Close fosamax in 10 minutes,
        redLed(compartment1)=BLINKING, time_consumption(
        compartment1)=[360], time_consumption(compartment2)
        =[730,1140], redLed(compartment1)=ON, drugIndex(
        compartment1)=0, isPillMissed(compartment1)=false,
        pillTakenWithDelay(compartment2)=false, systemTime=361,
        pillTakenWithDelay(compartment1)=false, drugIndex(
        compartment2)=0, name(compartment2)="moment", name(
        compartment1)="fosamax", isPillMissed(compartment2)=false,
        day=0}

========= Compositional step 3 =========
[step:3 of SafePillbox]
Model execution outcome: SAFE
Updated locations: {}
[step:3 of pillbox]
Model execution outcome: SAFE
Updated locations: {outMess(compartment1)=fosamax taken, redLed(
        compartment1)=OFF, actual_time_consumption(compartment1)
        =[366], systemTime=366, pillTakenWithDelay(compartment1)=
        true}
Out locations: {outMess(compartment1)=fosamax taken, redLed(
        compartment1)=OFF, pillTakenWithDelay(compartment1)=true,
         time_consumption(compartment1)=[360], time_consumption(
        compartment2)=[730,1140], redLed(compartment1)=OFF,
        drugIndex(compartment1)=0, isPillMissed(compartment1)=false,
         pillTakenWithDelay(compartment2)=false,
        actual_time_consumption(compartment1)=[366], systemTime
        =366, pillTakenWithDelay(compartment1)=true, drugIndex(
        compartment2)=0, name(compartment2)="moment", name(
        compartment1)="fosamax", isPillMissed(compartment2)=false,
        day=0}

========= Compositional step 4 =========
[step:4 of SafePillbox]
Model execution outcome: SAFE
Updated locations: {}
[step:4 of pillbox]
Model execution outcome: SAFE
Updated locations: {outMess(compartment2)=Take moment, redLed(
        compartment2)=ON, systemTime=731}

Out locations: {outMess(compartment2)=Take moment, outMess(
        compartment1)=fosamax taken,redLed(compartment2)=ON,
        time_consumption(compartment1)=[360], time_consumption(
        compartment2)=[730,1140], redLed(compartment1)=OFF,
        drugIndex(compartment1)=0, isPillMissed(compartment1)=false,
         pillTakenWithDelay(compartment2)=false,
        actual_time_consumption(compartment1)=[366], systemTime
        =731, pillTakenWithDelay(compartment1)=false, drugIndex(
        compartment2)=0, name(compartment2)="moment", name(
        compartment1)="fosamax", isPillMissed(compartment2)=false,
        day=0}

========= Compositional step 5 =========
[step:5 of SafePillbox]
Model execution outcome: SAFE
Updated locations: {}
[step:5 of pillbox]
Model execution outcome: SAFE
Updated locations: {outMess(compartment2)=Take moment in 10
        minutes, redLed(compartment2)=BLINKING}
Out locations: {outMess(compartment2)=Take moment in 10 minutes,
        outMess(compartment1)=fosamax taken, redLed(compartment2)
        =BLINKING, time_consumption(compartment1)=[360],
        time_consumption(compartment2)=[730,1140], redLed(
        compartment1)=OFF, drugIndex(compartment1)=0, isPillMissed(
        compartment1)=false, pillTakenWithDelay(compartment2)=
        false, actual_time_consumption(compartment1)=[366],
        systemTime=742, pillTakenWithDelay(compartment1)=false,
        drugIndex(compartment2)=0, name(compartment2)="moment",
        name(compartment1)="fosamax", isPillMissed(compartment2)=
        false, day=0}

========= Compositional step 6 =========
[step:6 of SafePillbox]
Model execution outcome: SAFE
Updated locations: {}
[step:6 of pillbox]
Model execution outcome: SAFE
Updated locations: {outMess(compartment2)=moment missed, redLed(
        compartment2)=OFF, isPillMissed(compartment2)=true}
Out locations: {outMess(compartment2)=moment missed, redLed(
        compartment2)=OFF, time_consumption(compartment1)=[360],
        time_consumption(compartment2)=[730,1140], redLed(
        compartment1)=OFF, drugIndex(compartment1)=0, isPillMissed(
        compartment1)=false, pillTakenWithDelay(compartment2)=
        false, actual_time_consumption(compartment1)=[366],
        systemTime=753, pillTakenWithDelay(compartment1)=false,
        drugIndex(compartment2)=0, name(compartment2)="moment",
        name(compartment1)="fosamax", isPillMissed(compartment2)=
        true, day=0}

========= Compositional step 7 =========
[step:7 of SafePillbox]
Model execution outcome: SAFE
Updated locations: {setOriginalTime(compartment2)=false,
        newTime(compartment2)=790, setNewTime(compartment2)=
        true}
Out locations: {setOriginalTime(compartment2)=false, newTime(
        compartment2)=790, setNewTime(compartment2)=true}
[step:7 of pillbox]
Model execution outcome: SAFE
Updated locations: {outMess(compartment2)=moment
        rescheduled, redLed(compartment2)=OFF,
        time_consumption(compartment2)=[790,1140]}
Out locations: {outMess(compartment2)=moment rescheduled, redLed(
        compartment2)=OFF, time_consumption(compartment1)=[360],
        time_consumption(compartment2)=[790,1140], redLed(
        compartment1)=OFF, drugIndex(compartment1)=0, isPillMissed(
        compartment1)=false, pillTakenWithDelay(compartment2)=
        false, actual_time_consumption(compartment1)=[366],
        systemTime=753, pillTakenWithDelay(compartment1)=false,
        drugIndex(compartment2)=0, name(compartment2)="moment",
        name(compartment1)="fosamax", isPillMissed(compartment2)=
        false, day=0}
```

Code 11. Composition output of the MRM system.

```
init −n 3
setup comp as Hardware.asm <|> (Controller.asm <|> Supervisor.asm)
run(comp, {adc_reply_m=RESPONSE;fan_working_m=true;pi_6_m=25;pi_6_reply_m=RESPONSE;temperature_m=25;state= STARTUP;
        insp_valve=CLOSED;exp_valve=OPEN;mCurrTimeSecs=1})
run(comp, {adc_reply_m=RESPONSE;fan_working_m=true;pi_6_m=25;pi_6_reply_m=RESPONSE;temperature_m=25;startu pEnded=
        true;mCurrTimeSecs=2})
run(comp, {adc_reply_m=RESPONSE;fan_working_m=true;pi_6_m=25;pi_6_reply_m=RESPONSE;temperature_m=25;selfTe stPassed=
        true;mCurrTimeSecs=3})
run(comp, {adc_reply_m=RESPONSE;fan_working_m=true;pi_6_m=25;pi_6_reply_m=RESPONSE;temperature_m=25;startV entilation
        =true;mCurrTimeSecs=4})
run(comp, {adc_reply_m=RESPONSE;fan_working_m=true;pi_6_m=25;pi_6_reply_m=RESPONSE;temperature_m=25;mCurrT imeSecs
        =5})
run(comp, {adc_reply_m=RESPONSE;fan_working_m=true;pi_6_m=25;pi_6_reply_m=RESPONSE;temperature_m=25;mCurrT imeSecs
        =6})
```

Code 12. Composition script of the MVM system.

6.1 Architectural Design Principles Comparison

Compositional simulation of I/O ASMs has the main advantage of modeling adhering to well-known design principles that lead to maintainable software, such as “divide and conquer”, high cohesion and low coupling, and information hiding [19]. In general, applying these principles diligently will result in models that have many advantages over monolithic models. In our case, some overall goals we want to achieve with the compositional modeling approach include, but are not limited to:

- Avoiding a monolithic and tightly coupled system model by promoting “separation of concerns” and the “Single Responsibility Principle”.
- Accelerating model development by distributing the work of developing a complex system model among a group of people that would work on individual sub-models in isolation and integrate their work later on to form the final system model thanks to well-defined I/O interfaces.
- Making a system model scalable and potentially distributable across different, separate computation nodes/engines (e.g., as runtime models in a Digital Twin platform).
- Increasing model qualities such as usability, analyzability, maintainability, and reusability.
- Achieve a more evolvable system model by preparing for an extension of the model.

In the following, we provide a comparison of the two approaches by using the general design principles [19, Chapter 9] mentioned before as classification framework. The results are summarized in Table 1.

(1) **Divide and conquer:** The logical breakdown of the system behavior into interrelated self-contained subsystems/components is a well-known engineering technique to tackle design complexity.

Monolithic approach. Within a monolith model, the divide and conquer principle is applicable by modularizing the model in terms of ASM modules (i.e., by introducing separate model units that can be independently developed) and by orchestrating their execution through the main rule of a unique ASM model. Such ASM model composes the ASM modules by containment (i.e., all the ASM modules are imported, compiled and linked into the final executable ASM model).

Compositional approach. Making ASM models compositional, with well-defined I/O bindings, facilitates both the decomposition task (the separate models can be independently developed) and the integration task. This last would be achieved at runtime through the orchestration formula over the ASM assembly.

```
========= Compositional step 1 =========
[step 1 of MVMHardware]
Out locations: {temperature=25, fan_working=true, adc_reply=RESPONSE, pi_6_reply=RESPONSE, pi_6=25}
[step 1 of MVMController]
Out locations: {mCurrSecs=1, all_cont=NONE, watchdog=true}
[step 1 of MVMSupervisor]
Out locations: {status_selftest=NOTSTART, state=INIT, watchdog_st=INACTIVE}
========= Compositional step 2 =========
[step 2 of MVMHardware]
Out locations: {temperature=25, fan_working=true, adc_reply=RESPONSE, pi_6_reply=RESPONSE, pi_6=25}
[step 2 of MVMController]
Out locations: {mCurrSecs=2, all_cont=NONE, watchdog=true, enter_self=true}
[step 2 of MVMSupervisor]
Out locations: {al_bit=NONE, insp_valve=OPEN, status_selftest=PERFORMING, state=SELFTEST, watchdog_st=INA CTIVE, exp_valve
        =CLOSED}
========= Compositional step 3 =========
[step 3 of MVMHardware]
Out locations: {temperature=25, fan_working=true, adc_reply=RESPONSE, pi_6_reply=RESPONSE, pi_6=25}
[step 3 of MVMController]
Out locations: {mCurrSecs=3, all_cont=NONE, watchdog=true, exit_self=true, enter_self=true}
[step 3 of MVMSupervisor]
Out locations: {al_bit=NONE, insp_valve=CLOSED, status_selftest=ENDED, state=VENTILATIONOFF, watchdog_st= INACTIVE,
        exp_valve=OPEN}
========= Compositional step 4 =========
[step 4 of MVMHardware]
Out locations: {temperature=25, fan_working=true, adc_reply=RESPONSE, pi_6_reply=RESPONSE, pi_6=25}
[step 4 of MVMController]
Out locations: {mCurrSecs=4, oValve=CLOSED, run_command=true, breath_sync=INSP, stop_command=false, all_c ont=NONE,
        watchdog=true, exit_self=true, enter_self=true, iValve=OPEN}
[step 4 of MVMSupervisor]
Out locations: {al_bit=NONE, insp_valve=OPEN, status_selftest=ENDED, state=VENTILATIONON, watchdog_st=BRE ATHON,
        exp_valve=CLOSED}
========= Compositional step 5 =========
[step 5 of MVMHardware]
Out locations: {temperature=25, fan_working=true, adc_reply=RESPONSE, pi_6_reply=RESPONSE, pi_6=25}
[step 5 of MVMController]
Out locations: {mCurrSecs=5, oValve=OPEN, run_command=true, breath_sync=EXP, stop_command=false, all_cont =NONE, watchdog
        =true, exit_self=true, enter_self=true, iValve=OPEN}
[step 5 of MVMSupervisor]
Out locations: {al_bit=NONE, insp_valve=OPEN, status_selftest=ENDED, state=FAILSAFE, watchdog_st=BREATHON , exp_valve=
        OPEN}
========= Compositional step 6 =========
[step 6 of MVMHardware]
Out locations: {temperature=25, fan_working=true, adc_reply=RESPONSE, pi_6_reply=RESPONSE, pi_6=25}
[step 6 of MVMController]
Out locations: {mCurrSecs=6, oValve=OPEN, run_command=true, breath_sync=EXP, stop_command=false, all_cont =NONE, watchdog
        =true, exit_self=true, enter_self=true, iValve=OPEN}
[step 6 of MVMSupervisor]
Out locations: {al_bit=NONE, insp_valve=CLOSED, status_selftest=ENDED, state=FAILSAFE, watchdog_st=ALARM, exp_valve=OPEN}
```

Code 13. Composition output of the MVM when error in input valve status.

(2) **Increase cohesion:** A subsystem component has high cohesion if it keeps together things that are related to each other (e.g., functional cohesion), and keeps out other things.

    Monolithic approach. A set of related functions and rules for reading input and producing outputs are to be kept together in one ASM module, and everything else is to be kept out. So a certain effort is required to properly design ASM modules.

    Compositional approach. One module keeps together a set of related functions and rules for reading in values and producing out values, and everything else is kept out. This is done by establishing a well-defined set of interrelated in/out interfaces.

(3) **Reduce coupling:** Coupling occurs when there are interdependencies between one module and another.

Monolithic approach. There is normally data and control coupling since ASM modules are imported into one ASM model.

Compositional approach. It reduces data coupling, since the data being passed over a formula is usually the values of only input/output locations of a model. Of course, there is normally control coupling (output values sent as input by a model to the next one(s)).

(4) **Increase abstraction:** To reduce complexity, your designs should allow you to hide or defer consideration of details. Abstractions work by allowing you to understand the essence of something and make important decisions without knowing unnecessary details.

Monolithic approach. Each individual ASM module can be small, and therefore easier to understand. Separate ASM modules are good abstractions, but it is necessary to understand the details of how they operate to determine possible interactions and avoid inconsistencies and interferences for functions updates.

Compositional approach. Each individual ASM model can be small as well, and therefore easier to understand. Separate I/O ASM models are good abstractions. You can use them as black boxes with no need to understand the details of how they operate.

(5) **Increase reuse and reusability:** Designing for reusability means designing various aspects of your system so that they can be used again in other contexts, both in your system and in other systems.

Monolithic approach. ASM modules are visible within the ASM model that imports them. This benefit encourages reuse, though the absence of explicit I/O module bindings make it difficult to recognize modules with high potential reusability.

Compositional approach. The architectural model assembly makes each model component visible. This benefit encourages reuse. By analyzing the architecture, you can discover those components that can be obtained from past projects or from third parties. You can also identify model components that have high potential reusability.

(6) **Design for flexibility:** It is about anticipating changes that a system model may have to undergo in the future and preparing for them. With a compositional and highly modular model, it is easier to plan the evolution of the system model itself.

Monolithic approach. Low coupled and high cohesive ASM modules facilitate changes and extensions. However, changing the modules orchestration protocol requires changing the main rule of the ASM model, so it requires changing the ASM model.

Compositional approach. Independent models interrelated by well-defined I/O interfaces allows us to more readily replace a model and add new extensions. Also, changing the orchestration protocol of the system sub-models is facilitated by simply changing the orchestration formula. So, different orchestration schemes can be easily proved by changing the formula over the ASM assembly and not the models.

(7) **Design for testability/analizability:** It is about to make testing and analysis of the system model easier. For instance, the scenario-based validation should be easy to apply.

Monolithic approach. An ASM module can be tested/analyzed independently, but it requires a main ASM that imports the module and runs it by invoking its rule(s) properly. From the simulation point of view, the execution of ASM modules as orchestrated by the main rule of a unique ASM model would result in a final global state without any visibility of the intermediate states upon execution of a single module. For instance, writing a scenario for the composed system [12], requires setting and checking the global ASM state.

Table 1. Comparison between Monolithic and Compositional Approaches, w.r.t. Architectural Design Principles in [19]

	Divide & conquer	Increase cohesion	Reduce coupling	Increase abstraction	Increase reuse	Design for flexibility	Design for testability	Design defensively
Monolithic	medium	medium	low	high	low	medium	medium	medium
Compositional	high	high	high	high	high	high	high	medium

```
main rule r_Main =
 seq
  par
   r_Tram[] r_Pedestrian[]
  endpar
  par
   r_CrossManager[]
   seq
    r_Controller[]
    par
     r_TrafficLightA1[] r_TrafficLightA2[] r_TrafficLightB1[] r_TrafficLightB2[]
    endpar
   endseq
  endpar
 endseq
```

Code 14. Scheduling of the traffic light cross manager components.

Compositional approach. ASM models can be tested/analyzed independently before being composed with a simulation formula, also by applying scenario-based validation to each of them. During the compositional simulation, intermediate state changes and update sets of sub-models are externally visible.

(8) **Design defensively:** It is about performing validity checks to avoid attempts to use a system model inappropriately.

Monolithic approach. No rigorous checking can be applied without well-defined in/out interfaces of ASM modules. The designer can use invariants to specify safety constraints over the global ASM state.

Compositional approach. It favors defensive design at the level of components. In/out values of each component model can be rigorously checked/sanitized [9]. Moreover, a design-by-contract approach, writing careful pre-conditions and post-conditions for each component model, could be also adopted.

6.2 Simulation Comparison

We compare here the monolithic and compositional approaches in terms of simulation by reporting the results for the traffic light cross manager.

We have written a unique monolithic ASM that includes all eight component models (as modules) of the traffic light cross-manager system. The behavior of each component is captured by an ASM rule and the monolithic ASM schedules, by the main rule shown in Code 14, their behavior as requested in the formula in Code 6.

A certain modification of the component models, in terms of rules and functions, was necessary to reproduce the execution as given by the compositional simulation. E.g., the operator full-duplex bidirectional pipe and synchronous parallel split, are both translated using the parallel ASM rule, with the addition of shared functions for the full-duplex bidirectional pipe to allow the output exchange. Similar considerations hold for managing, in terms of the sequential rule, the pipe and the half-duplex bidirectional pipe We have also modified the modules accordingly, to avoid, for instance, redefinitions of domains and functions.

Then, we simulated the monolithic ASM giving the same input specified in Code 6. Concerning the performance (in terms of number of run steps, number of updated locations at each step), we observed that the number of run steps to achieve the same result is the same in both approaches, while in the monolithic approach, we observed a lower number of updates than the compositional approach (8 against 10). However, in the output of the compositional approach, we have the advantage of immediately identifying the values of the interface functions related to each component.

Assuming now that we want to reuse the same components to model an intersection with the same behavior but with one traffic light A and one traffic light B. In the case of the compositional approach, the only change to be made concerns the compositional formula in Code 6, which becomes:

(pedestrian || tram) <|> crossManager <||> controller | (trafficlightA || trafficlightB)

Instead, considering the monolithic ASM, the modeler should do the following manual changes: remove the corresponding ASM rules from the main rule in Code 14, remove the traffic lights functions (trafficLightA2 and trafficLightB2), remove every behavior (rule) related to the deleted traffic lights.

6.3 Strengths and Limitations

Based on our modeling and validation experience with the proposed approach, we can conclude that decomposing the system model into two or more sub-models and loosely coupling their simulation (co-simulation), provided us several advantages w.r.t. creating an entire system model. Since from the beginning, the sub-models can be developed by different groups and for different engineering domains and target platforms (e.g., the MVM controller prototype on the Arduino board [8]), this compositional modeling technique allowed us a high degree of flexibility in managing the separation of the modeling/analysis tasks in different modeling/development groups, each working in parallel at their own speed. As a result, subsystem models can be analyzed and validated/verified in isolation to prove their correctness according to the established I/O interfaces with the other subsystem models, so allowing us to speed up the overall formal development process from requirements to code. Moreover, this compositional modeling helped to clarify and make precise w.r.t. the documented system requirements, the communication protocol among components.

Defining the I/O bindings between the sub-models and capturing the computational causality between sub-models in terms of a compositional simulation formula is, however, not a trivial task and requires a clear understanding of how the involved subsystems react to I/O stimuli and of their communication and computation protocol.

In a wider context, this technique could help us in providing virtual models (e.g., an ASM model that is a virtual copy of a system controller) and leveraging model simulation at runtime. Models@run.time together with formal analysis and data analytics are the main enablers of the Digital Twin technology, which is a growing interest in any field. By model simulation, it is possible to interact with the digital twin of a real system (or a part of it) by simulating different *what-if* analysis scenarios [13] to identify the best actions to be then applied on the physical twin.

6.4 Threats to Validity

Some aspects may threaten the validity of both the presented approach and the evaluation results. In the following, we discuss the most important ones, namely *internal*, and *external* validity.

Internal validity. To mitigate this threat we designed a set of controlled validation experiments based on known use cases related to case studies from the research literature to which the authors contributed to. So thanks to this prior knowledge about the case studies and the availability of their modeling and analysis artifacts, we could compare the outcomes of our validation campaign for the compositional simulation of the decomposed version of these system models w.r.t. the validation results obtained for the preceding monolithic version of the same system models. This direct manipulation of the previous system models has been fundamental to assess cause-effect relations among the system model entities and their functions, and therefore to show the correctness of the proposed approach.

External validity. Two main factors might threaten the external validity of our approach.

First, threats may exist if the characteristics of the systems of our case studies are not indicative of the characteristics of other systems. We limited these threats by adopting, in addition to small model examples, two case studies from the medical domain, which are highly safety-critical and complex enough. The application of our approach to additional case studies whose decomposition lead to a high number of sub-models, and therefore to a more complex composition formula with sub-formulas, is part of our future work.

Second, the quality of the decomposition, especially if inherited by system models defined by third parties, may naturally bring to a bad identification of proper composition formula for orchestrating the sub-systems models, and, as a consequence, to failure in providing the desired behavior of the system in an aggregated form. However, being the subsystems models usually treated as black-boxes, the global system behavior can not be realized by acting inside the composed models, rather, it emerges from the models interactions (the I/O function bindings) and therefore has to be realized by a composition formula that acts on these interactions unveiling the right computational causality between sub-models.

7 RELATED WORK

We drew inspiration from existing frameworks that influenced our approach to compositional model-based simulation. These frameworks include those associated with workflow modeling and service orchestration, such as tools like **Business Process Model and Notation (BPMN)** [22] and the Jolie language [2]. Additionally, we were influenced by frameworks related to multi-state machine modeling, such as Yakindu statecharts [18]. However, our proposed technique focuses on a distributed model-based system simulation. Its primary application is in practical scenarios where model simulation is necessary during runtime, and models need to be co-simulated alongside real systems. This includes runtime models that form part of the knowledge base for a self-adaptive and autonomous system [6] or a digital twin plant [16].

Several approaches to model (co)simulation exist for coupling different simulators and simulations of multi-dimensional models of cyber-physical systems [15]. With co-simulation, emphasis is given to combine parts of the systems, modeled and simulated by different techniques and modeling paradigms, such as physics-related models, control laws, and sequential, concurrent and real-time interoperability patterns. Some of these approaches are formal (as our approach). For example, Circus for cyber physical systems [30] is a process algebra that formalizes a co-simulation model conforming to the FMI **Functional Mockup Interface (FMI)** [21]. Although some operators are similar (as the sequence and parallelism), our operators compose models of the same type (ASMs), and are more inspired by workflow/orchestrations operators from the field of service composition and by conventional forms of **inter-process communication (IPC)** from the field of operating systems.

The paper [23] introduces a choreography automaton for choreographic modeling of communication systems. This automaton consists of a system of communicating finite state machines, where the transitions are labeled by synchronous or asynchronous interactions. Choreographies are effective methods for describing contemporary software architectures like microservices. However, in our initial compositional simulation approach, we chose to utilize centralized synchronous communication semantics, which is commonly found in IT service orchestration and automation platforms. The definition and implementation of choreography constructs for deploying and executing asynchronous I/O ASMs are postponed to future work.

ASMs have found utility in the realm of component- and service-based architectures, specifically in the OASIS/OSOA standard **Service Component Architecture (SCA)**. This standard is employed for the assembly of heterogeneous services, and ASMs are utilized for service modeling and prototyping within this framework. In this context, abstract implementations or prototypes of SCA components known as SCA-ASM components are co-executed alongside other component implementations. The work by Riccobene and Scandurra [24] explores this concept in detail.

In a separate study by Mirandola et al. [20], a method for predicting the reliability of service assemblies is presented. The method takes into account both the system-level and component-level aspects of reliability. It achieves this by combining a reliability model designed for an SCA assembly that involves SCA-ASM components.

8 CONCLUSION AND FUTURE DIRECTIONS

This paper presents a modeling and simulation method for DESs based on the I/O ASM concept and the introduction of the compositional simulation technique for I/O ASMs. The utility of compositional simulation of I/O ASMs is demonstrated through the traffic light cross manager, the medicine reminder and monitoring system, and the Mechanical Ventilator Milano utilizing the tool `AsmetaComp`. `AsmetaComp` enables distributed simulation of ASMs by facilitating the connection and co-simulation of separate ASM system models. This allows the creation of simulations for integrated systems or the division of a large ASM model into smaller sub-models that can co-execute, potentially on different local or remote processes/computers.

As part of our future endeavors, we aim to extend our support for supplementary composition operators and patterns like attempt choice and non-deterministic choice. Then, we aim to improve the verification tool to allow compositional verification, by studying the limitations and advantages of verifying small interacting models against monolithic ones.

Additionally, our plans include incorporating choreography constructs from the standard **Business Process Modeling Notation 2.0 (BPMN2)** *Choreography Diagram* to enable decentralized co-simulation of asynchronous ASMs (as described in [22]). Furthermore, we aim to facilitate the co-simulation of ASM models with other simulated or real subsystems within a federated, interoperable simulation environment. This integration could adhere to established standards such as the IEEE 1516 **High-Level Architecture (HLA)** and the FMI [21] for distributed co-simulation [17].

REFERENCES

[1] 2023. ASMETA (ASM mETAmodeling) Toolset. (2023). https://asmeta.github.io/

[2] 2023. Jolie. (2023). https://jolie-lang.org

[3] A. Abba, C. Accorsi, P. Agnes, E. Alessi, P. Amaudruz, A. Annovi, F. Ardellier Desages, S. Back, C. Badia, J. Bagger, V. Basile, G. Batignani, A, Bayo, B. Bell, M. Beschi, D. Biagini, G. Bianchi, S. Bicelli, D. Bishop, T. Boccali, A. Bombarda, S.

Bonfanti, W. M. Bonivento, M. Bouchard, M. Breviario, S. Brice, R. Brown, J. M. Calvo-Mozota, L. Camozzi, M. Camozzi, A. Capra, M. Caravati, M. Carlini, A. Ceccanti, B. Celano, J. M. Cela Ruiz, C. Charette, G. Cogliati, M. Constable, C. Crippa, G. Croci, S. Cudmore, C. E. Dahl, A. Dal Molin, M. Daley, C. Di Guardo, G. D'Avenio, O. Davignon, M. Del Tutto, J. De Ruiter, A. Devoto, P. Diaz Gomez Maqueo, F. Di Francesco, M. Dossi, E. Druszkiewicz, C. Duma, E. Elliott, D. Farina, C. Fernandes, F. Ferroni, G. Finocchiaro, G. Fiorillo, R. Ford, G. Foti, R. D. Fournier, D. Franco, C. Fricbergs, F. Gabriele, C. Galbiati, P. Garcia Abia, A. Gargantini, L. Giacomelli, F. Giacomini, F. Giacomini, L. S. Giarratana, S. Gillespie, D. Giorgi, T. Girma, R. Gobui, D. Goeldi, F. Golf, P. Gorel, G. Gorini, E. Gramellini, G. Grosso, F. Guescini, E. Guetre, G. Hackman, T. Hadden, W. Hawkins, K. Hayashi, A. Heavey, G. Hersak, N. Hessey, G. Hockin, K. Hudson, A. Ianni, C. Ienzi, V. Ippolito, C. C. James, C. Jillings, C. Kendziora, S. Khan, E. Kim, M. King, S. King, A. Kittmer, I. Kochanek, J. Kowalkowski, R. Krücken, M. Kushoro, S. Kuula, M. Laclaustra, G. Leblond, L. Lee, A. Lennarz, M. Leyton, X. Li, P. Liimatainen, C. Lim, T. Lindner, T. Lomonaco, P. Lu, R. Lubna, G. A. Lukhanin, G. Luzón, M. MacDonald, G. Magni, R. Maharaj, S. Manni, C. Mapelli, P. Margetak, L. Martin, S. Martin, M. Martínez, N. Massacret, P. McClurg, A. B. McDonald, E. Meazzi, R. Migalla, T. Mohayai, L. M. Tosatti, G. Monzani, C. Moretti, B. Morrison, M. Mountaniol, A. Muraro, P. Napoli, F. Nati, C. R. Natzke, A. J. Noble, A. Norrick, K. Olchanski, A. Ortiz de Solorzano, F. Padula, M. Pallavicini, I. Palumbo, E. Panontin, N. Papini, L. Parmeggiano, S. Parmeggiano, K. Patel, A. Patel, M. Paterno, C. Pellegrino, P. Pelliccione, V. Pesudo, A. Pocar, A. Pope, S. Pordes, F. Prelz, O Putignano, J. L. Raaf, C. Ratti, M. Razeti, A. Razeto, D. Reed, J. Refsgaard, T. Reilly, A. Renshaw, F. Retriere, E. Riccobene, D. Rigamonti, A. Rizzi, J. Rode, J. Romualdez, L. Russel, D. Sablone, S. Sala, D. Salomoni, P. Salvo, A. Sandoval, E. Sansoucy, R. Santorelli, C. Savarese, E. Scapparone, T. Schaubel, S. Scorza, M. Settimo, B. Shaw, S. Shawyer, A. Sher, A. Shi, P. Skensved, A. Slutsky, B. Smith, N. J. T. Smith, A. Stenzler, C. Straubel, P. Stringari, M. Suchenek, B. Sur, S. Tacchino, L. Takeuchi, M. Tardocchi, R. Tartaglia, E. Thomas, D. Trask, J. Tseng, L. Tseng, L. VanPagee, V. Vedia, B. Velghe, S. Viel, A Visioli, L. Viviani, D. Vonica, M. Wada, D. Walter, H. Wang, M. H. L. S. Wang, S. Westerdale, D. Wood, D. Yates, S. Yue, and V. Zambrano. 2021. The novel Mechanical Ventilator Milano for the COVID-19 pandemic. *Physics of Fluids* 33, 3 (Mar. 2021), 037122.

[4] Paolo Arcaini, Andrea Bombarda, Silvia Bonfanti, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. 2021. *The ASMETA Approach to Safety Assurance of Software Systems*. Springer International Publishing, Cham, 215–238.

[5] José Ángel Bañares and José Manuel Colom. 2019. Model and simulation engines for distributed simulation of discrete event systems. In *Economics of Grids, Clouds, Systems, and Services*, Massimo Coppola, Emanuele Carlini, Daniele D'Agostino, Jörn Altmann, and José Ángel Bañares (Eds.). Springer International Publishing, Cham, 77–91.

[6] Nelly Bencomo, Sebastian Götz, and Hui Song. 2019. Models@run.time: A guided tour of the state of the art and research challenges. *Software and Systems Modeling* 18, 5 (2019).

[7] Massimo Bombino and Patrizia Scandurra. 2013. A model-driven co-simulation environment for heterogeneous systems. *Int. J. Softw. Tools Technol. Transf.* 15, 4 (2013), 363–374. https://doi.org/10.1007/s10009-012-0230-5

[8] Silvia Bonfanti, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. 2022. Compositional simulation of abstract state machines for safety critical systems. In *Formal Aspects of Component Software - 18th International Conference, FACS 2022, November 10–11, 2022, Proceedings (LNCS)*, Silvia Lizeth Tapia Tarifa and José Proença (Eds.), Vol. 13712. Springer, 3–19. https://doi.org/10.1007/978-3-031-20872-0_1

[9] Silvia Bonfanti, Elvinia Riccobene, and Patrizia Scandurra. 2023. A component framework for the runtime enforcement of safety properties. *J. Syst. Softw.* 198 (2023), 111605. https://doi.org/10.1016/J.JSS.2022.111605

[10] Egon Börger and Alexander Raschke. 2018. *Modeling Companion for Software Practitioners*. Springer, Berlin.

[11] Egon Börger and Robert Stärk. 2003. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag.

[12] Alessandro Carioni, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. 2008. A scenario-based validation language for ASMs. In *Proceedings of the 1st International Conference on Abstract State Machines, B and Z (ABZ '08) (Lecture Notes in Computer Science)*, Vol. 5238. Springer-Verlag.

[13] Aidan Fuller, Zhong Fan, Charles Day, and Chris Barlow. 2020. Digital Twin: Enabling technologies, challenges and open research. *IEEE Access* 8 (2020), 108952–108971.

[14] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. 2008. A metamodel-based language and a simulation engine for abstract state machines. *J. UCS* 14, 12 (2008).

[15] Cláudio Gomes, Casper Thule, David Broman, Peter Gorm Larsen, and Hans Vangheluwe. 2018. Co-simulation: A survey. *ACM Comput. Surv.* 51, 3, Article 49 (May 2018), 33 pages. https://doi.org/10.1145/3179993

[16] Michael Grieves and John Vickers. 2016. Origins of the digital twin concept. *Florida Institute of Technology* 8 (2016), 3–20.

[17] Wim Huiskamp and Tom van den Berg. 2016. *Federated Simulations*. Springer International Publishing, Cham, 109–137. https://doi.org/10.1007/978-3-319-51043-9_6

[18] ITEMIS. 2023. YAKINDU Statechart Tools. (2023). https://itemis.com/en/yakindu/state-machine

[19] Timothy Lethbridge and Robert Laganiere. 2002. *Object-Oriented Software Engineering: Practical Software Development Using UML and Java* (1 ed.). McGraw-Hill, Inc., USA.

[20] Raffaela Mirandola, Pasqualina Potena, Elvinia Riccobene, and Patrizia Scandurra. 2014. A reliability model for service component architectures. *J. Syst. Softw.* 89 (2014), 109–127. https://doi.org/10.1016/j.jss.2013.11.002

[21] Modelica Association Project. 2023. Functional Mock-up Interface. (2023). https://fmi-standard.org/

[22] Object Management Group. 2023. Object Management Group Business Process Model and Notation. (2023). https://bpmn.org/

[23] Simone Orlando, Vairo Di Pasquale, Franco Barbanera, Ivan Lanese, and Emilio Tuosto. 2021. Corinne, a tool for choreography automata. In *Formal Aspects of Component Software - 17th International Conference, FACS 2021, Virtual Event, October 28–29, 2021, Proceedings (LNCS)*, Gwen Salaün and Anton Wijs (Eds.), Vol. 13077. Springer, 82–92. https://doi.org/10.1007/978-3-030-90636-8_5

[24] Elvinia Riccobene and Patrizia Scandurra. 2014. A formal framework for service modeling and prototyping. *Formal Aspects Comput.* 26, 6 (2014), 1077–1113. https://doi.org/10.1007/s00165-013-0289-0

[25] Elvinia Riccobene and Patrizia Scandurra. 2020. Model-based simulation at runtime with abstract state machines. In *Software Architecture*, Henry Muccini, Paris Avgeriou, Barbora Buhnova, Javier Camara, Mauro Caporuscio, Mirco Franzago, Anne Koziolek, Patrizia Scandurra, Catia Trubiani, Danny Weyns, and Uwe Zdun (Eds.). Springer International Publishing, Cham.

[26] Elvinia Riccobene and Patrizia Scandurra. 2020. Model-based simulation at runtime with abstract state machines. In *Software Architecture - 14th European Conference, ECSA 2020 Tracks and Workshops, Proceedings (Communications in Computer and Information Science)*, Vol. 1269. Springer.

[27] Carolyn Talcott, Sofia Ananieva, Kyungmin Bae, Benoit Combemale, Robert Heinrich, Mark Hills, Narges Khakpour, Ralf Reussner, Bernhard Rumpe, Patrizia Scandurra, and Hans Vangheluwe. 2021. *Composition of Languages, Models, and Analyses*. Springer International Publishing, Cham, 45–70.

[28] Yentl Van Tendeloo, Simon Van Mierlo, and Hans Vangheluwe. 2019. A multi-paradigm modelling approach to live modelling. *Software and Systems Modeling* 18, 5 (2019).

[29] Danny Weyns and M. Usman Iftikhar. 2016. Model-based simulation at runtime for self-adaptive systems. In *2016 IEEE International Conference on Autonomic Computing, ICAC 2016*, Samuel Kounev, Holger Giese, and Jie Liu (Eds.). IEEE Computer Society.

[30] Frank Zeyda, Julien Ouy, Simon Foster, and Ana Cavalcanti. 2018. Formalising cosimulation models. In *Software Engineering and Formal Methods*, Antonio Cerone and Marco Roveri (Eds.). Springer International Publishing, Cham, 453–468.

Received 26 May 2023; revised 18 December 2023; accepted 28 February 2024