Compositional Simulation of Abstract State Machines for Safety Critical Systems

Silvia Bonfanti[1][0000−0001−9679−4551], Angelo Gargantini[1][0000−0002−4035−0131], Elvinia Riccobene[2][0000−0002−1400−1026], and Patrizia Scandurra[1][0000−0002−9209−3624]

[1] University of Bergamo, Bergamo, Italy
{silvia.bonfanti,angelo.gargantini,patrizia.scandurra}@unibg.it
[2] Università degli Studi di Milano, Milano, Italy
elvinia.riccobene@unimi.it

Abstract. Model-based simulation is nowadays an accepted practice for reliable prototyping of system behavior. To keep requirements complexity under control, system components are specified by separate models, validated and verified in isolation from the rest, but models have to be subsequently integrated and validated as a whole. For this reason, engines for orchestrated simulation of separate models are extremely useful.

In this paper, we present a compositional simulation technique for managing the co-execution of Abstract State Machines (ASMs) communicating through I/O events. The proposed method allows the co-simulation of ASM models of separate subsystems of a Discrete Event System in a straight-through processing manner according to a predefined orchestration schema.

We also present our experience in applying and validating the proposed technique in the context of the MVM (Mechanical Ventilator Milano) system, a mechanical lung ventilator that has been designed, successfully certified, and deployed during the COVID-19 pandemic.

Keywords: Models composition, Models co-simulation, Abstract State Machines, Mechanical Ventilator Milano

1 Introduction

Model-based simulation is a widely accepted technique for prototyping the behavior of Discrete Event Systems (DESs), and the only practical alternative for understanding the behavior of complex and large systems [9, 27]. However, due to requirements complexity and system size, models are becoming unmanageable in the engineering process and are achieving unprecedented levels of scale and complexity in many fields [9]. Thus, it is becoming increasingly important to efficiently and effectively manage a system model as a composition of different sub-models to analyze separately and then integrate [12, 25].

Towards this direction, this paper presents a *compositional model-based simulation* technique to combine the simulation of formal models representing independent and interacting subsystems of a discrete-event system under prototyping. To illustrate the proposed approach, we adopt Abstract State Machines (ASMs) [8,15] as executable state-based formal modeling language for DESs. The approach allows the co-simulation of ASM models of separate subsystems in a *straight-through processing* manner [5] according to a predefined orchestration schema. Coordination/orchestration operators allow the definition of composition patterns for simulating models. The proposed approach is supported by a simulation composer engine, called `AsmetaComp`, which exploits the simulation tool `AsmetaS@run.time` [23] to execute ASMs as *living models* at runtime [10,26].

The paper also presents our experience in validating the proposed method on the Mechanical Ventilator Milano (MVM), a mechanical lung ventilator developed for COVID-19 patients. The MVM is an adequate case study since it is a safety-critical system and it is made of different interacting hardware/software subsystems with human-in-the-loop, which have been developed by different teams. Modeling the entire system in a single ASM model would be impractical and produce a complex model to manage, analyze and share with all groups. Our compositional modeling process started from a formal specification of the MVM subsystems as separate ASM models, which are validated and verified independently. The compositional simulation technique was then used to compose by orchestration the subsystems models according to established communication flows and I/O events exchanged among the MVM components. Therefore, we validated by scenarios simulation the composed model of the MVM system by checking if it performs in accordance with the expected outcomes from the requirements. The results of such modeling and validation approach helped us to clarify and improve the MVM component interfaces and the communication protocol, while taming the system model complexity.

The contributions of this paper can be summarized as follows: (i) we promote a practical and rigorous compositional modeling and simulation method for DESs through formal state-based models (like ASMs) supported by V&V (validation and verification) tools (like the ASMETA toolset for ASMs [1,8]); (ii) we provide the concept of I/O ASM and define a compositional execution semantics of orchestrated I/O ASMs; (iii) we report our experience in applying the proposed method, through the supporting simulation composer engine `AsmetaComp`, to a safety-critical case study in the health-care domain (the MVM).

This paper is organized as follows. Section 2 presents the MVM case study used to show the proposed approach. Section 3 provides basic concepts about ASMs and the ASMETA toolset. Section 4 presents the proposed compositional model-based simulation technique. Section 5 presents the application of the proposed technique to the MVM case study and describes the features of the composition engine supporting the proposed technique. Section 6 reports our lesson learned from such a modeling and validation experience with the MVM system. Finally, Section 7 shows related work and Section 8 concludes the paper with future directions for its extensions.

2 The Mechanical Ventilator Milano (MVM) case study

During the COVID-19 pandemic, our research team was involved in the design, development, and certification of a mechanical lung ventilator called MVM (Mechanical Ventilator Milano)[3] [7]. This section introduces the MVM case study by briefly outlining its purpose, the problem of the case that we address, and an informal description of the behavior of the main MVM components.

2.1 Problem context

Due to time constraints and lack of skills, no formal method was applied to the MVM project. However, later we had planned to assess the feasibility of developing (part of) the ventilator by using a component-based formal specification development. In this project, instead of creating one single system model, we split the system into several subsystems and loosely coupled their formal specifications and analysis. Right from the very beginning, the sub-models were developed by different groups and for different engineering domains and target platforms, in order to speed up the overall formal development process from requirements to code.

Then for the purpose of integrating all sub-models and simulating the whole system, we have adopted the compositional model simulation technique presented here, and used the supporting tool `AsmetaComp` to put it in practice. This technique provided us a high degree of flexibility, since it allowed us to effectively develop the sub-models separately, then to couple them via well-defined input and output interfaces, and examine the behavior of the overall integrated system according to a specific orchestrated co-simulation schema.

Fig. 1 shows a high-level overview of the main architecture components of the MVM system using a UML-like notation.

The core components of the MVM system are the software *MVMController* that manages the lung ventilation based on user inputs (acquired through the *GUI* component) and patient parameters, and the *Supervisor* software that monitors the overall system behavior and ensures that the machine *HW* operates safely. The wires between components (the solid lines with arrows) in Fig. 1 represent the information flow between components: an arrow incoming is the input for the component, an arrow outgoing is the output for the component. The wires are labeled with the name of the UML interface representing the signals exchanged between components, e.g., *ISC* shows the signals sent from the supervisor to the controller: the watchdog status and the status of self-test mode. In [11], we already reported our practical experience in using ASMs/ASMETA for modeling, analyzing, and encoding only the MVM software controller. For the purpose of this paper, the model of the supervisor and the hardware subsystems have also been developed. In addition, this paper describes the integration of all sub-system models via I/O interfaces and their orchestrated co-simulation.

[3] `https://mvm.care/`

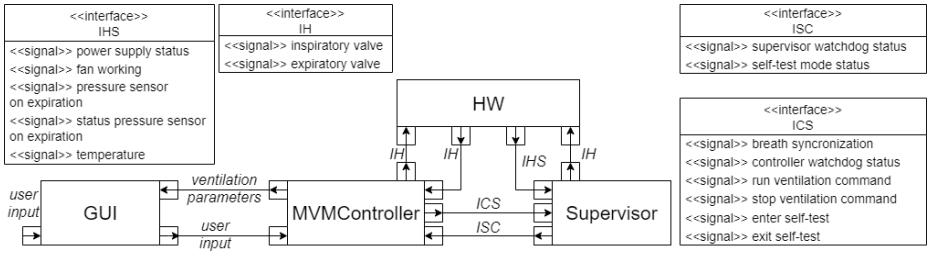


Fig. 1: Main components of the MVM system and signals exchanged

2.2 System behavioral description

MVM provides ventilation support in pressure-mode (i.e. the ventilator uses the pressure as a variable to control the respiratory cycle), for patients that are in intensive therapy and that require mechanical ventilation.

MVM supports two ventilation modes: *Pressure Controlled Ventilation* (PCV) and *Pressure Support Ventilation* (PSV). PCV mode is used for patients that are not able to start breathing on their own. The duration of the respiratory cycle is kept constant and set by the doctor; and the pressure changes between the target inspiratory pressure and the positive end-expiratory pressure. In PSV mode, the respiratory cycle is controlled by the patient. A new inspiration is initiated when a sudden pressure drop occurs, while expiration starts when the patient's inspiratory flow drops below a set fraction of the peak flow. If a new inspiratory phase is not detected within a certain amount of time (apnea lag), MVM will automatically switch to the PCV mode because it is assumed that the patient is not able to breathe alone. The air enters/exits through two valves: an input valve (opened during inspiration) and an output valve (opened during expiration). If the ventilator is not running, the input valve is closed and the output valve is opened to allow the patient to breath thanks to two relief valves; this configuration is called *safe-mode*.

Before starting the ventilation, the MVM controller passes through three phases. The *start-up* in which the controller is initialized with default parameters, *self-test* which ensures that the hardware is fully functional, and *ventilation off* in which the controller is ready for ventilation when requested.

The supervisor monitors the whole system behavior. After *initialization* and *startup* phases, the *self-test* procedure that checks all the hardware components is performed. When the ventilator is running, it is in *ventilation on* phase, in which it monitors all the ventilation parameters to avoid harm to the patient. If the supervisor detects a dangerous situation, the ventilator and the valves are moved to *safe-mode*. When the ventilator is not running, it is in *ventilation off* phase and the valves are in *safe-mode*.

3 Preliminary concepts on ASMs and ASMETA

ASMs [14] are an extension of Finite State Machines. Unstructured control states are replaced by *states* comprising arbitrarily complex data (i.e., domains of objects with functions defined on them); state *transitions* are expressed by transition rules describing how the data (state function values saved into *locations*) change from one state to the next.
An ASM model is structured in terms of:

- The `signature` section where domains and functions are declared. The model interface with its environment is specified by *monitored* functions that are written by the environment and read by the machine, and by *out* functions that are written by the machine and read by the environment; *controlled* functions are the internal functions used by the machine (read in the current state and updated in the next state).
- The `definitions` section where all transition rules and possible invariants are specified. *Transition rules* have different constructors depending on the update structure they express, e.g, guarded updates (`if-then`, `switch-case`), simultaneous parallel updates (`par`), etc. The *update* rule $f(t_1, \ldots, t_n) := v$, being f an n-ary function, t_i terms, and v the new value of $f(t_1, \ldots, t_n)$ in 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.
- The `default init` section where initial values for the *controlled* functions are defined.

An ASM *run* is defined as a finite or infinite sequence $S_0, S_1, \ldots, S_n, \ldots$ of states: starting from an initial state S_0, a *run step* from S_n to S_{n+1} consists in firing, in parallel, all transition rules and leading to simultaneous updates of a number of locations. In case of an inconsistent update (i.e., the same location is updated to two different values 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 ASMETA [1, 8], a set of tools around the ASM formal method. ASMs can be understood as executable pseudo-code or virtual machines working over abstract data structures at any desired level of abstraction.

3.1 Modeling example

As an example of ASM model, Code 1 reports excerpts of the ASM `MVMcontroller` modeling the controller using the `AsmetaL` textual modeling language.

In the section `signature` (see lines 2–11), respirationMode is a monitored function specifying the input command received by the controller from the GUI, as part of the *user inputs*, to change its mode of operation (PCV or PSV), while

```
1  asm MVMcontroller
2  signature:
3  enum domain States = {STARTUP | SELFTEST
4  | VENTILATIONOFF | PCV_STATE | PSV_STATE}
5     enum domain Modes = {PCV | PSV}
6     ...
7  dynamic controlled state: States
8  dynamic monitored respirationMode: Modes
9  dynamic monitored stopRequested: Boolean
10 dynamic out iValve: ValveStatus
11 dynamic out oValve: ValveStatus ...
12 definitions:
13 ...
14 main rule r_Main =
15 par
16 if state = STARTUP then r_startup[] endif
17 if state = SELFTEST then r_selftest[] endif
18 if state = VENTILATIONOFF then
19       r_ventilationoff[] endif
20 if state = PCV_STATE then r_runPCV[] endif
21 if state = PSV_STATE then r_runPSV[] endif
22 endpar
```

```
rule r_runPCV =
 par
  if phase = INSPIRATION then r_runPCVInsp[] endif
  if phase = EXPIRATION then r_runPCVExp[] endif
 endpar
rule r_runPCVInsp =
 par
  if not stopVentilation then
   if stopRequested then stopVentilation := true endif
  endif
  if expired(timerInspirationDurPCV) then
   par
    if respirationMode = PCV then
     r_PCVStartExp[] endif
    if respirationMode = PSV then
     par
      state := PSV_STATE
      r_PSVStartExp[]
     endpar
 endif endpar endif endpar
 default init s0:
 function state = STARTUP
```

Code 1: MVMController ASMETA model

the controlled function state represents the controller status. The out functions iValve and oValve specify the interface IH between MVMController and HW components, and are the input for HW to set the valves status during ventilation.

Initially, the function state is initialized at the value STARTUP. At each step, depending on the current state value, a corresponding rule fires through by the r_Main execution. These call rules are specified in the `definitions` section.

In Code 1, on the right, we show, for example, the rule r_runPCV regulating the PCV mode. It in turn calls rules for the inspiration, r_runPCVInsp (line 6), and the expiration, r_runPCVExp (here missed). In PCV mode, the transition between inspiration and expiration is determined by the duration of each phase decided by the physician (when timers timerInspirationDurPCV, in case of inspiration, and timerExpirationDurPCV, in case of expiration, expire). When the inspiration time is passed (line 11), the controller goes to the PCV expiration phase (line 14). If the physician has required (by setting the value of the monitored function respirationMode) to move to PSV mode (line 15), the machine changes the state from PCV to PSV and executes the rule r_PSVStartExp (line 18). If a stop request (by the monitored function stopRequested, still part of the user input interface between the GUI and the MVMController) is received during the inspiration phase (line 8), it is stored (in stopVentilation) and will be executed in the expiration phase.

3.2 The ASMETA (ASM mETAmodeling) toolset

ASMETA [1] provides the user with modeling notations, different analysis (V&V) techniques and automatic source code and test generators for ASMs to be applied at design-, development-, and operation- time [8]. In particular, a runtime

simulation engine, `AsmetaS@run.time` [24], has been developed within ASMETA as extension of the offline simulator `AsmetaS` [17] to handle an ASM as a living model [10,26] to run in tandem with a real software system. `AsmetaS@run.time` supports simulation *as-a-service* features of `AsmetaS` and additional features such as model execution with timeout and model roll-back to the previous state after a failure step during model execution. All these features are accessible by UI dashboards (both in a graphical and in a command-line way). This runtime model simulation mechanism has been already used within an *enforcer* software tool [13] to sanitize input/output events for a running system or to prevent the execution of unsafe commands by the system.

4 Compositional simulation of ASM models

Consider the partitioning of a software system into distinct subsystems/components interacting for sharing resources in terms of input/output events. Since ASMs are a state-based formalism, i.e., outputs of the machine depend only on its current state and inputs, formal behavioral models of these subsystem/components may be specified in terms of a set of ASMs, each having its own input I (the monitored locations of the ASM), current state (the controlled locations of the ASM), and output O (the out locations of the ASM). We denote by *I/O ASM* each of these component models that are defined as follows:

Definition 1 (I/O ASM). *An* I/O ASM *is an ASM model m with a non-empty set I_m of input (or monitored) functions and a non-empty set O_m of out functions in its signature. 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 out functions with the same symbol name and interpretation. Consider, for example, the typical cascade or sequence of two machines A and B, where the output of the machine A is the input of the 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 instantaneously the effect of x through the cascade of A and B at each 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 out 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 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 $m_i \xrightarrow{B_{i,j}} m_j$ 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 between two models m_i and m_j, then at least one function symbol f must occur in both the two models'

signatures and is used as out function for m_i and as input function for m_j, with the same domain and codomain and same interpretation.

The model `MVMcontroller` shown in Code 1 is an example of I/O ASM, having input functions $I = \{\mathsf{respirationMode}, \mathsf{stopRequest}\}$ and out functions $O = \{\mathsf{iValve}, \mathsf{oValve}\}$. The set O represents also the bindings of this model with the model of the hardware (not reported here), while the set I is the binding with the GUI component. Fig.1 provides a graphical view of the bindings among all the component models as wires labeled with the name of the UML interface representing the binding functions (the exchanged signals values).

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

Definition 3 (I/O ASM assembly). *An I/O ASM assembly is a set of I/O ASMs bound together by I/O ASM bindings.*

Fig.1 illustrates an I/O ASM assembly consisting of the I/O ASM models of the MVM system and the I/O ASM bindings among all the component models.

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 a 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 out 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 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 $c_1 \parallel c_2$, where c_1 and c_2 are composition formulas and $\{|, <|>, <\|>, \|\}$ 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 Fig.1 is executed using a composition formula made by two bidirectional pipes: $Hardware.asm <|> (MVMController.asm <|> Supervisor.asm)$.

Note that in case of a composition formula of the form $c_i \, op \, c_j$, we assume that there is no one model in common between c_i and c_j.

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 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 there 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 $c_1 = Hardware.asm$ and $c_2 = MVMController.asm <|> Supervisor.asm$ for the assembly shown in Fig.1. c_1 and c_2 are bound by the binding $B_{1,2} = B_{HW,MVMController} \cup B_{HW,Supervisor}$ where $B_{HW,MVMController}$ = {IH functions} and $B_{HW,Supervisor}$ = {IHS functions}.

Definition 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 $c_1 \overset{B}{\Longrightarrow} c_2$. 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: $c = c_1 <\|> c_2$*. 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.*

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.

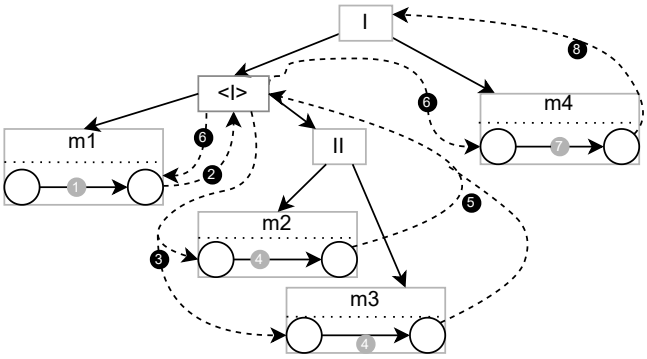


Fig. 2: Visit of a Compositional Simulation Tree for $(m_1 <|> (m_2 \parallel m_3)) \mid m_4$

Definition 7 (Compositional Simulation Tree of I/O ASMs). *Given a composition formula c of I/O ASMs m_i, $i = 1, ..., n$, 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* $\{|, <|>, <\|>, \|\}$.

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 non-empty CST with an empty set I_c[4] 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 8 (Runstep of an I/O ASM composition). *Given a composition formula c for an assembly A of I/O ASMs m_i, $i = 1, ..., n$, 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.*

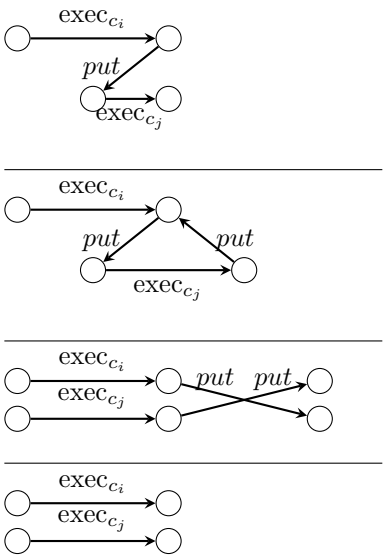
[4] We assume I_c is concretely realized as a map (or dictionary) that associates ASM function symbols (the keys) with their values.

Algorithm 1: *execCST*

```
   Input  : T: a CST
1  Function execCST(T):
2      if T.root ≠ NULL then
3          exec(T.root,∅);
```

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.
1  Function exec(node, I):
2      if node.isLeaf() then
3          run a step of node.model on input I;
4          return node.model.getOutLocations();
5      else
6          c1 = node.left;
7          c2 = node.right;
8          switch node.operator do
9              case | do
10                 O1 = exec(c1,I);
11                 put(c1,c2,B1,2);
12                 O2 = exec(c2,I);
13                 return O1 ∪ O2;
14             case < | > do
15                 O1 = exec(c1,I);
16                 put(c1,c2,B1,2);
17                 O2 = exec(c2,I);
18                 put(c2,c1,B2,1);
19                 return O1 ∪ O2 ;
20             case < || > do
21                 O1 = exec(c1,I);
22                 O2 = exec(c2,I);
23                 put(c1,c2,B1,2);
24                 put(c2,c1,B2,1);
25                 return  O1 ∪ O2;
26             case || do
27                 O1 = exec(c1,I);
28                 O2 = exec(c2,I);
29                 return O1 ∪ O2 ;
```



As an example, Fig. 2 depicts the depth visit of the CST 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 values, while labels 2, 3, 5, 6, and 8 denote copying of the binding function values according to the composition operators.

5 Compositional modeling and simulation at work

In order to support compositional I/O ASM simulation in practice, the tool `AsmetaComp` has been developed as part of the ASMETA tool-set, and it is based on the `AsmetaS@run.time`. We here show the compositional simulation tool on the MVM case study.

`AsmetaComp` can be used through a basic graphical user interface (`AsmetaComp` GUI) or a command line (`AsmetaComp` shell). `AsmetaComp` GUI, after having specified the number of models involved in the composition, allows the user to

```
1  init −n 3
2  setup comp as Hardware.asm <|> (Controller.asm <|> Supervisor.asm)
3  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})
4  run(comp, {adc_reply_m=RESPONSE;fan_working_m=true;pi_6_m=25;pi_6_reply_m=RESPONSE;
       temperature_m=25;startupEnded=true;mCurrTimeSecs=2})
5  run(comp, {adc_reply_m=RESPONSE;fan_working_m=true;pi_6_m=25;pi_6_reply_m=RESPONSE;
       temperature_m=25;selfTestPassed=true;mCurrTimeSecs=3})
6  run(comp, {adc_reply_m=RESPONSE;fan_working_m=true;pi_6_m=25;pi_6_reply_m=RESPONSE;
       temperature_m=25;startVentilation=true;mCurrTimeSecs=4;run_command=false})
7  run(comp, {adc_reply_m=ERROR;fan_working_m=true;pi_6_m=25;pi_6_reply_m=RESPONSE;
       temperature_m=25;startVentilation=false;mCurrTimeSecs=5;run_command=false})...
8  run(comp, {adc_reply_m=ERROR;fan_working_m=true;pi_6_m=25;pi_6_reply_m=RESPONSE;
       temperature_m=25;startVentilation=false;mCurrTimeSecs=9;run_command=true})
```

Code 2: Composition script

define the composition formula and to execute the model composition. Using the `AsmetaComp` shell, instead, the user can set and run a composition of models interactively or via a script of commands. Such a script can be written using a textual composer language for the basic composition operators introduced in Sect. 4.

Applying `AsmetaComp` *to MVM.* Recall from Fig. 1 the four components of the MVM design. We have abstracted the GUI since the user interface is represented by the concept of ASM environment. Each of the other three components has been modeled as an I/O ASM[5], and validated individually by using the ASMETA tools. Then, we have composed the three models and executed their composition.

According to the MVM operation, the hardware sends values to both controller and supervisor and both of them return the configuration of hardware components based on the ventilation status (bidirectional pipe between hardware and controller/supervisor). Moreover, the controller sends information (i.e., its status and alarms raised) to the supervisor, which in turn returns its status to it (bidirectional pipe between controller and supervisor). Therefore, the MVM components interaction is captured by two nested bidirectional pipes: $Hardware.asm <|> (MVMController.asm <|> Supervisor.asm)$.

Through the script reported in Code 2 for the `AsmetaComp` shell, we set this composition formula (by command `setup` at line 2) and express some compositional steps of the interacting MVM models, including inputs from the environment for the unbounded ASM monitored functions of models (by commands `run`). Specifically, we plan to simulate the startup (lines 3–4), selftest (line 5), start ventilation (line 6) and the intervention of the supervisor by setting the valves in safe mode (line 8) because there is an error in the power supply (line 7).

Code 3 reports the script execution trace. The components (hardware, controller and supervisor) perform the first compositional step where all the functions are initialized. In the second step the controller performs self test. In compositional step three the controller is in ventilation off waiting for starting ventilation, and the supervisor has concluded self-test and is waiting for the command

[5] The models are available at `https://github.com/asmeta/asmeta/tree/master/code/experimental/asmeta.simulator%40run.time/examples/MVM/ConfModels`

```
========= Compositional step 1 =========
Running MVMHardware
I={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}
US={iValveMerged=CLOSED; oValveMerged=OPEN}
Running MVMController
US={state=STARTUP; all_cont=NONE; watchdog=true}
Running MVMSupervisor
US={status_selftest=NOTSTART; state=INIT; watchdog_st=INACTIVE; max_attempts_adc=0}

========= Compositional step 2 =========
MVMController performs self test
========= Compositional step 3 =========
MVMController is in VENTILATIONOFF, MVMSupervisor ends SELFTEST
========= Compositional step 4 =========
MVM is in INSPIRATION phase
========= Compositional step 5 =========
Running MVMHardware
ADC reports an error
I={adc_reply_m=ERROR; fan_working_m=true; pi_6_m=25; pi_6_reply_m=RESPONSE; temperature_m=25}
US={iValveMerged=OPEN; oValveMerged=CLOSED}
Running MVMController
I={stopRequested=false; pawGTMaxPinsp=true}
US={oValve=OPEN; phase=EXPIRATION; breath_sync=EXP; iValve=CLOSED}
Running MVMSupervisor
US={insp_valve=CLOSED; max_attempts_adc=1; exp_valve=OPEN; previous_breath=EXP}
========= Compositional step 6 =========
Running MVMHardware
ADC reports an error
I={adc_reply_m=ERROR; fan_working_m=true; pi_6_m=25; pi_6_reply_m=RESPONSE; temperature_m=25}
US={iValveMerged=CLOSED; oValveMerged=OPEN}
Running MVMController
I={stopRequested=false; dropPAW_ITS=false}
Running MVMSupervisor
US={max_attempts_adc=2}
========= Compositional step 7 =========
Running MVMHardware
ADC reports an error
I={adc_reply_m=ERROR; fan_working_m=true; pi_6_m=25; pi_6_reply_m=RESPONSE; temperature_m=25}
US={iValveMerged=CLOSED; oValveMerged=OPEN}
Running MVMController
I={stopRequested=false;dropPAW_ITS: true}
Running MVMSupervisor
The supervisor moves in FAILSAFE
US={max_attempts_adc=3; state=FAILSAFE}
========= Compositional step 8 =========
Running MVMHardware
I={adc_reply_m=ERROR; fan_working_m=true; pi_6_m=25; pi_6_reply_m=RESPONSE; temperature_m=25}
Ventilator is in INSPIRATION
US={iValveMerged=OPEN; oValveMerged=CLOSED}
Running MVMController
US={oValve=CLOSED; state=PCV_STATE; phase=INSPIRATION; run_command=true; breath_sync=INSP; stop_command=false; iValve=OPEN}
Running MVMSupervisor
Supervisor set valves in safe mode
US={max_attempts_adc=4; iValve=CLOSED; oValve=OPEN}
========= Compositional step 9 =========
Running MVMHardware
I={adc_reply_m=ERROR; fan_working_m=true; pi_6_m=25; pi_6_reply_m=RESPONSE; temperature_m=25}
Nevertheless the controller is in INSPIRATION, since the supervisor is in FAILSAFE the valves are in safe mode
US={iValveMerged=CLOSED; oValveMerged=OPEN}
...
```

Code 3: Simulation trace supervisor in FAILSAFE, the valves change state

from the controller when ventilation has begun. In the next compositional step, the ventilator is ventilating and the supervisor detects it. In compositional step five, six and seven the power supply reports an error (adc_reply_m=ERROR) and in compositional step seven the supervisor is in FAILSAFE state. In the next compositional step, since the controller is in inspiration phase, the input valve is open and the output valve is closed. But in the last compositional step reported in Code 3, nevertheless the controller is in inspiration, the valves are in safe mode (input valve is closed and output valve is opened) because the supervisor has detected an error.

6 Discussion and lesson learned

Based on our modeling experience with the MVM system, we can conclude that splitting 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 right from the very beginning, the sub-models were developed by different groups and for different target platforms (e.g., the MVM controller prototype on the Arduino board), this compositional model-

ing 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 could 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 MVM 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, with the proposed technique we intend to contribute 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 [16] to identify the best actions to be then applied on the physical twin.

7 Related work

Existing frameworks that inspired our approach to compositional model-based simulation are those related to workflow modeling and service orchestration (such as tools for the Business Process Model and Notation (BPMN) [4], and the Jolie language [3]), and to multi-state machine modeling (like Yakindu statecharts [6]). However, the proposed technique is oriented to a distributed model-based system simulation, to be used for example in practical contexts where model simulation is required at runtime and models have to be co-simulated along with real systems, such as runtime models that are part of the knowledge base of a self-adaptive and autonomous system [10] or of a *digital twin* plant [18].

In [21], a *choreography automaton* for the choreographic modeling of communicating systems is introduced as a system of communicating finite state machines whose transitions are labelled by synchronous/asynchronous interactions. Choreographies are suitable approaches to describe modern software architectures such as micro-services, but in this first compositional simulation mechanism we preferred to rely on a centralized synchronous communication semantics that is typical of IT service orchestration and automation platforms. We postpone as future work the definition and implementation of choreography constructs to deploy and enact a choreography-based execution of asynchronous I/O ASMs.

Within the context of component- and service- based architectures, ASMs have been used for service modeling and prototyping in the OASIS/OSOA standard Service Component Architecture (SCA) for heterogeneous service assembly.

In such a framework, abstract implementation (or prototype) of SCA components in ASM (SCA-ASM components) are co-executed *in place* with other component implementations [22]. In [20], a method for predicting service assembly reliability both at system-level and component-level is presented by combining a reliability model for an SCA assembly involving SCA-ASM components.

8 Conclusion and future directions

In this paper, we have formulated the concept of I/O ASM and introduced the compositional simulation technique of I/O ASMs. We have also presented, through the MVM case study, the scope and use of the compositional simulation of I/O ASMs as supported by the ASMETA tool `AsmetaComp`. `AsmetaComp` is intended to support distributed simulation of ASMs by allowing separate ASM system models to be connected and co-simulated together to form simulations of integrated systems, or a big ASM model to be divided into smaller sub-models that co-execute, possibly on separate local or remote processes/computers.

As future work, we want to conduct some experiments for evaluating the real benefits of a compositional simulation w.r.t. a conventional non-compositional simulation of one single monolithic model. These include the evaluation of the usability of the technique and, in particular, the understandability of the execution traces. We plan to investigate on the fault-detection capability of the proposed approach to guarantee that using composition does not reduce the ability to discover unacceptable behaviors.

We also want to support additional composition operators and patterns (such as I/O wrapping, conditional execution, iterated execution, alternate execution, attempt choice, non-deterministic choice, etc.), and alternative model roll-back semantics to allow more expressiveness in specifying how models interact in a compositional simulation. We also want to support choreography constructs for decentralized co-simulation of asynchronous ASMs as provided in the standard Business Process modeling Notation 2.0 (BPMN2) *Choreography Diagram* [4]. Moreover, we want to support the co-simulation of ASM models with other DES simulated/real subsystems into a federated, interoperable simulation environment, possibly in accordance with standards such as the IEEE 1516 High Level Architecture (HLA) and the Functional Mockup Interface (FMI) [2] for distributed co-simulation [19].

Acknowledgement: We thank the students Davide Santandrea and Michele Zenoni for their contribution in tool implementation and case study composition.

References

1. ASMETA (ASM mETAmodeling) toolset, `https://asmeta.github.io/`
2. Functional Mock-up Interface, `https://fmi-standard.org/`
3. Jolie, `https://jolie-lang.org`

4. Object Management Group Business Process Model and Notation, https://bpmn.org/
5. Straight Through Processing - STP, Investopedia, October 18, 2020, https://www.investopedia.com/terms/s/straightthroughprocessing.asp
6. YAKINDU Statechart Tools, https://itemis.com/en/yakindu/state-machine
7. Abba, A., et al.: The novel mechanical ventilator milano for the COVID-19 pandemic. Physics of Fluids **33**(3), 037122 (mar 2021)
8. Arcaini, P., Bombarda, A., Bonfanti, S., Gargantini, A., Riccobene, E., Scandurra, P.: The ASMETA Approach to Safety Assurance of Software Systems, pp. 215–238. Springer International Publishing, Cham (2021)
9. Bañares, J.Á., Colom, J.M.: Model and simulation engines for distributed simulation of discrete event systems. In: Coppola, M., Carlini, E., D'Agostino, D., Altmann, J., Bañares, J.Á. (eds.) Economics of Grids, Clouds, Systems, and Services. pp. 77–91. Springer International Publishing, Cham (2019)
10. Bencomo, N., Götz, S., Song, H.: Models@run.time: a guided tour of the state of the art and research challenges. Software and Systems Modeling **18**(5) (2019)
11. Bombarda, A., Bonfanti, S., Gargantini, A., Riccobene, E.: Developing a prototype of a mechanical ventilator controller from requirements to code with ASMETA. Electronic Proceedings in Theoretical Computer Science **349**, 13–29 (Nov 2021)
12. Bombino, M., Scandurra, P.: A model-driven co-simulation environment for heterogeneous systems. Int. J. Softw. Tools Technol. Transf. **15**(4), 363–374 (2013)
13. Bonfanti, S., Riccobene, E., Scandurra, P.: A runtime safety enforcement approach by monitoring and adaptation. In: Biffl, S., Navarro, E., Löwe, W., Sirjani, M., Mirandola, R., Weyns, D. (eds.) Software Architecture - 15th European Conference ECSA 2021. pp. 20–36. No. 12857 in Lecture Notes in Computer Science, Springer International Publishing, Cham (2021)
14. Börger, E., Raschke, A.: Modeling Companion for Software Practitioners. Springer, Berlin, Heidelberg (2018)
15. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer Verlag (2003)
16. Fuller, A., Fan, Z., Day, C., Barlow, C.: Digital twin: Enabling technologies, challenges and open research. IEEE Access **8**, 108952–108971 (2020)
17. Gargantini, A., Riccobene, E., Scandurra, P.: A Metamodel-based Language and a Simulation Engine for Abstract State Machines. J. UCS **14**(12) (2008)
18. Grieves, M.: Origins of the Digital Twin Concept (08 2016)
19. Huiskamp, W., van den Berg, T.: Federated Simulations, pp. 109–137. Springer International Publishing, Cham (2016)
20. Mirandola, R., Potena, P., Riccobene, E., Scandurra, P.: A reliability model for service component architectures. J. Syst. Softw. **89**, 109–127 (2014)
21. Orlando, S., Pasquale, V.D., Barbanera, F., Lanese, I., Tuosto, E.: Corinne, a tool for choreography automata. In: Salaün, G., Wijs, A. (eds.) Formal Aspects of Component Software - 17th International Conference, FACS 2021, Virtual Event, October 28-29, 2021, Proceedings. LNCS, vol. 13077, pp. 82–92. Springer (2021)
22. Riccobene, E., Scandurra, P.: A formal framework for service modeling and prototyping. Formal Aspects Comput. **26**(6) (2014)
23. Riccobene, E., Scandurra, P.: Model-based simulation at runtime with abstract state machines. In: Muccini, H., Avgeriou, P., Buhnova, B., Camara, J., Caporuscio, M., Franzago, M., Koziolek, A., Scandurra, P., Trubiani, C., Weyns, D., Zdun, U. (eds.) Software Architecture. Springer International Publishing, Cham (2020)

24. Riccobene, E., Scandurra, P.: 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 (2020)
25. Talcott, C., Ananieva, S., Bae, K., Combemale, B., Heinrich, R., Hills, M., Khakpour, N., Reussner, R., Rumpe, B., Scandurra, P., Vangheluwe, H.: Composition of Languages, Models, and Analyses, pp. 45–70. Springer International Publishing, Cham (2021)
26. Tendeloo, Y.V., Mierlo, S.V., Vangheluwe, H.: A multi-paradigm modelling approach to live modelling. Software and Systems Modeling **18**(5) (2019)
27. Weyns, D., Iftikhar, M.U.: Model-Based Simulation at Runtime for Self-Adaptive Systems. In: Kounev, S., Giese, H., Liu, J. (eds.) 2016 IEEE International Conference on Autonomic Computing, ICAC 2016. IEEE Computer Society (2016)