Science of Computer Programming 244 (2025) 103299

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Formal specification and validation of the MVM-Adapt system using Compositional I/O Abstract State Machines

Silvia Bonfanti [a,*], Elvinia Riccobene [b], Patrizia Scandurra [a]

[a] University of Bergamo, Bergamo, Italy
[b] Università degli Studi di Milano, Milan, Italy

A R T I C L E   I N F O

Keywords:
Compositional model-based simulation
Models@run.time
Discrete Event Systems modeling
Abstract State Machines

A B S T R A C T

To face complexity and scalability, the design of software-intensive systems requires the *decomposition* of the system into components, each modeled and analyzed separately from the others, and the *composition* of their analysis. Moreover, compositional model simulation is recognized as the only alternative available in practice when systems are large and complex, like in the cyber-physical domain, and intrinsically require combining the specification of ensembles of different parts (subsystems). Therefore, the need for simulation engines for composed model execution is getting a growing interest.

Along this research line, this paper presents the results of the compositional modeling and validation by scenarios of an industrial medical system, called MVM-Adapt, that we designed as an adaptive version of an existing mechanical lung ventilator deployed and certified to treat pneumonia during the COVID-19 pandemic.

We exploit the I/O Abstract State Machine formalism to model the device components as separate and interacting sub-systems that communicate through I/O events and adapt the device ventilation mode at run-time based on the health parameters of the patient. An orchestrated simulation coordinates the overall execution of these communicating I/O ASMs by exploiting suitable workflow patterns.

This compositional simulation technique has proved to be useful in practice to validate the new adaptive MVM's behavior and thus to support architects in better understanding this new mode of operation of the prototyped system.

1. Introduction

Software-intensive systems, including heterogeneous systems, embedded systems for automotive applications, telecommunications and wireless systems, cyber physical systems, business applications with web services, etc., are becoming very complex. Therefore, their design and analysis require the system decomposition into separate components, each modeled, developed, and analyzed separately from the others, and then combined to analyze the overall behavior and quality of the system under development. This necessity has recently pushed the research towards the definition of composable system models and the composition of their analysis [1], as well as the development of simulation engines for composed model execution [2]. Such compositional model simulation is sometimes the only alternative available in practice to evaluate complex system models, and it is particularly used in the context of distributed

* Corresponding author.
E-mail addresses: silvia.bonfanti@unibg.it (S. Bonfanti), elvinia.riccobene@unimi.it (E. Riccobene), patrizia.scandurra@unibg.it (P. Scandurra).

https://doi.org/10.1016/j.scico.2025.103299
Received 31 December 2023; Received in revised form 6 February 2025; Accepted 13 March 2025

Available online 17 March 2025

discrete-event systems (DESs) [3], where distinct subsystems/components interact for sharing resources in terms of input/output events, and of CPSs (considered an evolution of embedded systems) with system constituents of a different nature (hardware, software, and physical parts). Moreover, in several application contexts, the possibility to use models at runtime is getting a growing interest in co-simulating them with the real system. Consider, for example, runtime models being used as part of the knowledge base of a self-adaptive/autonomous system [4] or of a *digital twin* to simulate different *what-if/if-what* analysis scenarios [5] to identify the best actions to be then applied to the physical twin.

The focus of this article is to present the compositional modeling and simulation of *MVM-Adapt*, an industrial medical system that we have been engineering as an adaptive version of the MVM (Mechanical Ventilator Milano) [6], which is a mechanical lung ventilator designed, certified, and deployed during the COVID-19 pandemic. Formal modeling the adaptive behavior of such a medical system requires the involvement of different modelers and domain experts working on specific parts of the system, and specifying the behavior of these parts separately and how these parts merge together by communicating through explicit signal interfaces.

To this end, in [7,8] we defined the concept of I/O ASMs as a formal way to model DESs in Abstract State Machines (ASMs) [9,10], and we introduced a simulation technique and workflow patterns (such as parallel composition and cascading) for the compositional simulation of I/O ASMs. Each DES component can be modeled by an I/O ASM having its input events (monitored locations of the ASM), current state (controlled locations of the ASM), and output events (out locations of the ASM). I/O ASMs can be used as black-boxes to be assembled by suitably binding their inputs/outputs (I/O interfaces) and then their executions can be composed by specific orchestration operators.

We have used the I/O ASM formalism and the compositional simulation engine to model and co-simulate the components of the MVM-Adapt system. The MVM-Adapt project[1] was based on the idea of improving the basic ventilation functionalities (controlled and supported) of MVM with a ventilation mode able to adapt ventilation parameters at run-time on the base of the health parameters of the patient. This adaptive ventilation is available in the most sophisticated lung ventilators existing on the market.

We modeled the behavior of the MVM-Adapt system as a composition of two I/O ASMs, one for the controller and one for the supervisor: they work independently, expose adaptive functionalities, and communicate through I/O events. These models refine the ASM models of the original MVM system, by adding the adaptive ventilation mode and establishing precise I/O signal interfaces. Due their complexity (in terms of number of control variables and signal exchanges, and behavioral features), each model has been defined, validated and verified separately. We then exploited the compositional simulation technique for I/O ASMs [7] to coordinate the overall execution of these components and evaluate in practice, through different adaptation scenarios, the effectiveness of the new adaptive behavior of the MVM.

The models of the MVM-Adapt were concisely presented in [11] as an industrial experience paper in the conference ABZ 2023. This paper extends the work in [11] significantly by describing the models comprehensively, reporting the results of the validation and verification of the component models, and presenting the main results and lesson learned of the compositional modeling and validation of the MVM-Adapt by scenarios. The main contributions of this paper are to:

- apply formal modeling and analysis of the behavior of the MVM-Adapt subsystems separately as I/O ASMs;
- apply compositional simulation of the I/O ASM models of the MVM-Adapt; and
- provide a preliminary behavior validation of the new adaptive mode of MVM through two important scenarios (the *Breath triggered Scenario* and the *Otis Scenario*).

The collected experience on modeling this medical system is used to discuss and further confirm the potential advantages for system engineers of the proposed compositional modeling approach over the traditional, monolithic one, as originally discussed in [8]. In this way, formal analysis is conducted for different system parts in isolation, and then a joint co-simulation allows to understand the cross interactions among the communicating parts. This early compositional modeling approach allows to quickly prototype and highlight unexpected/hidden behaviors (if any), and also facilitates engineers to validate system design alternatives (e.g., different component behaviors) that have to fulfill changing or new requirements (as in the case of the adaptive mode of the MVM system).

This paper is organized as follows. Section 2 presents the MVM-Adapt system and the two execution scenarios of interest for this adaptive version of the MVM. Section 3 provides basic concepts about I/O ASMs and their compositional simulation technique. Section 4 describes the overall component architecture of the MVM-Adapt system and the I/O interfaces of the two main communicating subsystems, the I/O ASM models of the behavior of these two subsystems and the composition formula for their co-simulation. Section 5 describes the validation & verification (V&V) of the two component models in isolation. Section 6 presents the application of the compositional simulation technique to the MVM-Adapt system model through the two validation scenarios outlined in Section 2. Section 7 reports on related works. Section 8 discusses some lessons learned from our modeling and validation experience with the MVM-Adapt system. Finally, Section 9 concludes the paper with future directions for its extensions.

2. MVM-Adapt use case

The Mechanical Ventilator Milano (MVM) system, as outlined in [6], was designed as part of an international research initiative in response to the COVID-19 pandemic. Its primary objective was to address the shortage of mechanical ventilators in hospitals by

[1] MVM-Adapt (Milano Ventilatore Meccanico Adaptive in the presence of uncertainty, FISR (Covid-19) project MUR FISR 2020 IP_05310 MVM-Adapt, funded in 2021.

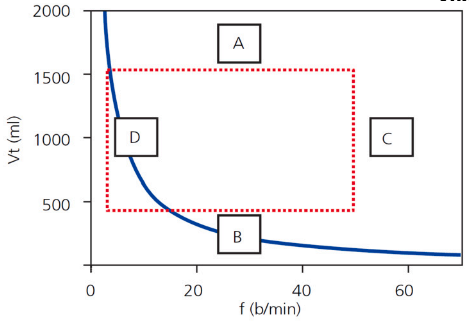


Fig. 1. Lung protection strategy illustration for a 70 kg patient: the blue curve is the *target minute volume curve* (in liters per minute) – taken from [15]. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

rapidly developing a ventilator using cost-effective components. Despite realizing the initial MVM prototype within a month, obtaining certification from the relevant authorities (FDA in the USA, CE in Europe) necessitated an additional three months of dedicated full-time work involving approximately 60 researchers, including computer scientists and engineers. To provide insight into the intricacy of the MVM, its detailed behavior is articulated through approximately 1000 requirement sentences. One document describes the behavior of the overall system, while 15 requirements documents describe the detailed behavior of software components.

Owing to time limitations and a shortage of expertise, the MVM project did not employ any formal methods initially. Subsequently, we evaluated the feasibility of incorporating formal methods [12] and component-based formal specification development [7] to potentially enhance (part of) the ventilator's development process.

MVM was initially conceived to offer two fundamental ventilation modes, for treating individuals with COVID-19 pneumonia: *Pressure Support Ventilation* (PSV) and *Pressure Control Ventilation* (PCV). In PSV mode, MVM assists patients who are partially incapable of breathing independently, while in PCV mode, it governs the respiratory cycle of patients who are entirely unable to breathe on their own. A simplified version of the original (non-disclosable) specification document is available as ABZ Case study in [13].

Subsequently, the MVM project has evolved into the *MVM-Adapt* project, to incorporate the *Adaptive Support Ventilation* (ASV) mode into MVM. This improvement aligns MVM with the requirements of the most advanced mechanical ventilators [14].

In ASV,[2] the patient is guided using a favorable breathing pattern. This target breathing pattern (tidal volume V_t – i.e., the volume of the inspired/expired area – and respiratory rate f) is calculated using the Otis's equation [16], which is based on the assumption that if the optimal breath pattern results in the least work of breathing, it also results in the least amount of inspiratory pressure that must be applied by the ventilator when the patient is passive. Inspiratory pressure and respiratory rate are therefore adjusted to meet the target values according to Otis's curve (see the blue curve in Fig. 1) representing all possible combinations of V_t and f to obtain the target minute volume (in lit/m). Not all combinations of V_t and f are secure for the patient. In contrast to conventional ventilation modes, ASV allows lung protection by avoiding potentially harmful patterns. Specifically, ASV delimits an area (see Fig. 1) that circumscribes the safety boundaries (*Otis boundaries*) derived from appropriate calculations[3] to avoid high tidal volumes and pressures (A), low alveolar ventilation (B), dynamic hyperinflation or breath overlapping (C), and apnea (D).

In the *MVM-Adapt* project, we have implemented ASV both as user-selectable ventilation mode when starting ventilation and as an automatic transition from PCV mode. In PCV mode, the ventilator checks whether the current volume of the breathed area and the respiratory rate are within the Otis boundaries or not. If out of boundaries, the ventilation mode automatically changes from PCV to ASV to protect the patient and ensure a safer ventilation mode.

In Sect. 5, we show the validation and verification activities performed on the controller and supervisor components for the behavior described above. Moreover, for the validation by compositional simulation (Sect. 6), we have selected two scenarios since they are relevant to the adaptive behaviors of the ventilator: the breath trigger scenario and Otis scenario. The breath trigger scenario recreates the situation where MVM is ventilating in PCV mode, and the patient triggers a spontaneous breath three consecutive times. MVM must change ventilation mode from PCV to ASV. The necessity of this ventilation mode change is recognized by the supervisor, which adapts the controller in ventilating the patient. The Otis scenario simulates the situation in which MVM changes ventilation mode from PCV to ASV since the current respiratory values of the patient, i.e., the volume of the breathed area and respiratory rate, are out of the Otis boundaries.

3. Preliminary concepts on formal modeling

This section provides the necessary background on the Abstract State Machine (ASM) formal method and the I/O ASMs, which we exploited to model the components of the MVM-Adapt system, to validate each component in isolation from the others and verify required safety properties, and to validate the scenarios of their composition.

[2] To model the behavior of the MVM-Adapt we considered the ASV ventilation mode as it is implemented in the ventilator HAMILTON MEDICAL GALILEO [15].

[3] These internal calculations are based on some operator inputs for ASV, including the patient's ideal body weight, to determine the absolute limits, and on patient measurements that further narrow these limits to mitigate possible operator's mistakes and follow changes in the mechanics of the respiratory system. For more details, we refer the reader to [15].

3.1. Abstract State Machines

ASMs [9] are an extension of Finite State Machines. ASM *states* replace unstructured FSM control states by algebraic structures, i.e., domains of objects with functions and predicates defined on them. An ASM *location*, defined as the pair (*function-name*, *list-of-parameter-values*), represents the ASM concept of basic object container, and the couple (*location*, *value*) is a memory unit; an ASM state can be thus viewed as a set of abstract memories.

State transitions are performed by firing *transition rules*, which express the modification of functions interpretation from one state to the next one and, therefore, they change location values. Location *updates* are given as assignments of the form $loc := v$, where loc is a location and v its new value. They are the basic units of rules construction. By a limited but powerful set of *rule constructors*, location updates can be combined to express other forms of machine actions as: guarded actions (`if-then`, `switch-case`), simultaneous parallel actions (`par` and `forall`), sequential actions (`seq`), non-deterministic actions (`choose`).

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

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

An ASM *computation* (or *run*) is defined as a finite or infinite sequence $S_0, S_1, \dots, S_n, \dots$ 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) or invariant violations, the ASM run fails.

The left column in Code 1 reports, by using the `AsmetaL` textual modeling language, an example of ASM model: the specification of the adaptive controller component. The model, identified by a *name* after the keyword asm, is structured into four sections:

- The *header*, where the signature (functions and domains) is declared;
- The *body*, where transition rules are defined (details of transition rules are reported on the right column of Code 1), plus concrete domains and derived functions definitions, if any; this section of the model contains invariants and temporal logic formulas (if any);
- A *main rule*, which defines the starting rule of the machine, i.e., the rule that is executed at each run step and that, in turn, may call the other rules;
- The *initialization*, where a *default* initial state (among a set of) is defined.

3.1.1. ASMETA tool-set

The ASM formal method is supported by a tool-set, called ASMETA [17,10], for model editing, validation, verification, automatic test and code generation.

The tutorial paper [18] introduces the modeling notation, and the use of the ASMETA tools to specify and analyze models.

A core part of the tool-set is the `AsmetaS` simulator, enabling model validation by simulating the execution of ASM models and verifying adherence of the model to the requirements. `AsmetaS` offers a rich set of functionalities, including support for a wide range of validation tasks, from invariant checking, consistent update checking, random simulation, to interactive simulation modes. For more powerful scenario-based validation, the `AsmetaV` tool uses the `AsmetaS` simulator along with the *AValLa* language to express scenarios as sequences of actor actions and expected machine reactions, enabling comprehensive validation of execution scenarios on ASM models.

Formal verification of ASM models is handled by the `AsmetaSMV` tool, which translates ASM models into input for the NuSMV [19] model checker. `AsmetaSMV` supports specifying and checking both linear temporal logic (LTL) and computation tree logic (CTL) properties.

3.2. I/O Abstract State Machines and compositional model simulation

In [7,8], we exploited the ASM formal method as a state-based formalism to introduce an approach for the co-simulation of state-based models of independent and interacting components of a discrete event system. We provide the following definition of *I/O ASM* as an ASM exposing its own input/output interfaces and internal current state, plus a set of coordination/orchestration operators (see Sect. 3.2.1) for compositional simulation of ASM models that communicate with each other through I/O events.

Definition 3.1 *(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 output (or 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.[4]

[4] If model m is a Control State ASM [9], we talk of I/O control state ASM.

Table 1
I/O ASM composition operators.

Operator name	Symbol	Semantics description
(Simplex) pipe or sequence	$m_1 \mid m_2$	First execute m_1, then use the output of m_1 as input to execute m_2.
Half-duplex bidirectional pipe	m_1 <|> m_2	First execute m_1, subsequently use the output of m_1 as input to execute m_2; then, return the output of m_2 as input to m_1 for its next future execution (and not the current one).
Full-duplex bidirectional pipe	m_1 <||> m_2	First execute both m_1 and m_2 simultaneously, then return their outputs as inputs to both them for their next execution.
Synchronous parallel split (or fork-join)	$m_1 \parallel m_2$	Execute both m_1 and m_2 separately on their inputs, then return their outputs.

The I/O interface, defined in terms of input and out ASM functions, represents the set of interaction ports (or points) with the environment or other ASMs, while the set of controlled locations represents the internal state of the I/O ASM. Therefore, I/O ASMs can interact in a black-box manner *by binding* [7,8] input and out functions with the same symbol name and interpretation.

I/O ASMs – combined into a precise *I/O ASM Assembly* [7,8] – can be used to model software systems that can be partitioned into distinct subsystems/components interacting by sharing resources in terms of input/output events. A concrete example of I/O ASM assembly is described in Section 4 for modeling the architectural view of the MVM-Adapt system (note that the model at the left column of Code 1 is also an I/O ASM where `monitored` function and `out` functions are the interface of the component model.)

3.2.1. *Model composition mechanism*

Composition operators have been defined in [7,8] to specify the assembly of I/O ASM models by a *composition formula* denoting an orchestration schema for the co-simulation of the composed I/O ASM models. Table 1 reports the four basic binary composition operators over two I/O ASMs m_1 and m_2, and briefly informally describes their execution semantics.

Composition formulas are built by using such operators. For example, $(m_1$ <|> $(m_2 \parallel m_3)) \mid m_4$ denotes the orchestration schema of four ASMs, 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. Further details on the simulation mechanism of an I/O ASM assembly can be found in [7,8].

The I/O ASMs compositional simulation approach is supported by a simulation composer engine, called `AsmetaComp`, which exploits the simulation tool `AsmetaS@run.time` [20], developed within ASMETA as an extension of the offline simulator `AsmetaS` [21]. `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 run failure.

4. MVM-Adapt: architecture and models

This section discusses the architecture description of the MVM-Adapt system, including the I/O interfaces, the signal flows exchanged between the system components for the two main interaction scenarios, an overview of the internal behaviors of the system components and their coordination schema as specified in terms of a composition formula for I/O ASMs.

4.1. *Architecture*

The MVM-Adapt ventilator consists of four components: hardware (HW), Graphical User Interface (GUI), controller, and supervisor. Fig. 2 provides an architecture overview of the MVM-Adapt system model where a set of machines, i.e. I/O ASMs, are assembled via I/O function bindings and have to coordinate to each other. Function bindings are depicted as wires labeled with the name of the I/O interfaces `I_XY` for signals exchanged from component `X` to `Y`.

4.2. *I/O interfaces*

Fig. 3 shows the interfaces modeled using the UML notation. The *HW* is made of all the physical components that are necessary to perform ventilation and measure ventilation parameters. The user interface (*GUI*) is used for two purposes: the former is to display patient data and ventilation parameters, like oxygen concentration, measured by sensors, and the latter is to acquire ventilation parameters and alarm thresholds set by the medical operators. The main component in charge of the ventilation is the *controller*. It sets the input and output valves based on the phase of the respiratory cycle and notifies visual warnings and audible alarms. To assure patient safety, the ventilator was equipped with the *supervisor*, responsible for checking the overall behavior. If errors occur, it brings the system to a *fail-safe* state operating directly on the hardware (bypassing the controller). For example, the supervisor monitors the state of the inspiration and expiration valve based on the phase of the respiratory cycle, the temperature inside the ventilator, and the status of the sensors. Moreover, the supervisor monitors the ventilation when PCV is active, and under certain conditions (i.e. the patient triggers a spontaneous breath three consecutive times or the volume of the breathed area and respiratory rate are out of the

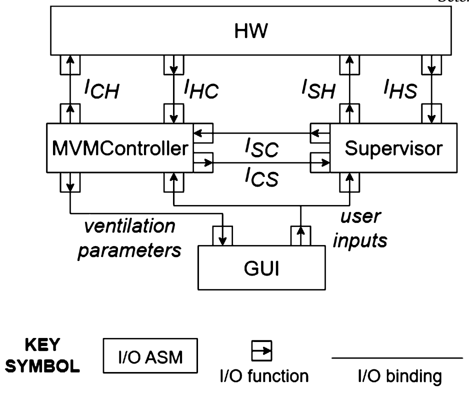


Fig. 2. MVM-Adapt I/O ASM assembly.

Otis boundaries, see Sect. 4.3.2) the ventilation mode changes from PCV to ASV to reduce the patient's effort, and protect his/her lungs. In the sequel, we detail the controller and supervisor components, while we abstract from GUI and HW, since for co-simulation purposes we are not interested in them.

The I/O ASM MVMController has input functions $I = user\_inputs \cup I_{HC} \cup I_{SC}$ and out functions $O = I_{CH} \cup I_{CS} \cup ventilation\_parameters$ (see Fig. 2). The set I is the union of the following interface signals:

User inputs are the parameters received from the GUI given by the doctor. The percentage minute volume (volMinPerc), the Positive End-Expiratory Pressure peep_in, the patient weight, the lung compliance (comp_lung), the expiratory time constant (rcexp_in) are used to compute the target minute volume (in liters per minute) and frequency according to the Otis's curve. Before starting the ventilation, the physician selects the operation mode (PCV, PSV, or ASV) using the respirationMode_doc command. startVentilation and stopVentilation commands, are respectively used to start and stop the ventilation. If expiratory pause (cmdExPause), inspiratory pause (cmdInPause), or recruitment maneuver (cmdRm) are required, the corresponding signal is sent from the GUI to the controller.

I_{HC} are the signals received from the hardware. When in expiration, in case the pressure sensor detects a sudden drop in pressure in the entrance to the patient's airway (the signal dropPAW_ITS is set to true) the ventilator immediately moves to the inspiration phase, the patient triggers a spontaneous breath. When in inspiration and the inspiratory flow drops below a fraction of the peak flow (Expiratory Trigger Setting (ETS)) of a given breath (flowDropASV - flowDropPSV), the ventilator moves to the expiration phase. Moreover, if the airway pressure (PAW) exceeds the maximum inspiratory pressure during inspiration (pawGTMaxPinsp), the cycle shall proceed immediately to expiration.

I_{SC} represents the signals received from the supervisor. The alarm parameters al_bit and the signal watchdoc_st are used by the Supervisor to communicate to the controller an alarm/(in)acting condition. The signal status_selftest communicates the status of the self-test procedure in the supervisor to the controller. Moreover, the respirationMode_out communicates the transition from PCV to ASV, when the supervisor detects excessive strain on the patient.

The set O represents the bindings of the controller component with the graphical user interface, hardware, and supervisor.

Ventilation parameters are all the signals read from sensors and communicated to the GUI. To the aim of our modeling and analysis purposes, we capture such signal values in terms of the unique state parameter, which represents the phase of ventilation the controller is.

I_{CH} are the functions to set the status of the input (iValve) and output (oValve) valves during ventilation.

I_{CS} are the signals sent to the supervisor. The lung compliance (comp_lung) set by the user, respiratoryMode_sup yields the current patient's respiratory mode to the Supervisor; breath_sync, indicates the current patient's inspiratory/expiratory phase; watchdog denotes alive communication between controller and supervisor; dropPAW_ITS_sup notifies that the patient has triggered a spontaneous breath; run_command and stop_ command communicate that the ventilation has started or stopped; enter_self and exit_self to communicate that the controller entered in self-test or exit from self-test; snooze a raised alarm; and all_cont communicates the alarm level (none, low, high, medium) when raised by the controller.

The Supervisor has been modeled as an I/O ASM Supervisor, which input functions and output functions are respectively: $I = I_{HS} \cup I_{CS} \cup user\_inputs$ and $O = I_{SH} \cup I_{SC}$ (see Fig. 2). Here we introduce the signals of I_{HS} and I_{SH}.

I_{HS} are the signals set to the supervisor by the hardware. The hardware communicates the status of the input (iValve) and output (oValve) valve. Moreover, the supervisor monitors the status of and in case of errors the ventilator moves to FAILSAFE mode: adc_reply returns the status of the power supply, fan_working returns if the fan is working or not, temperature returns the tem-

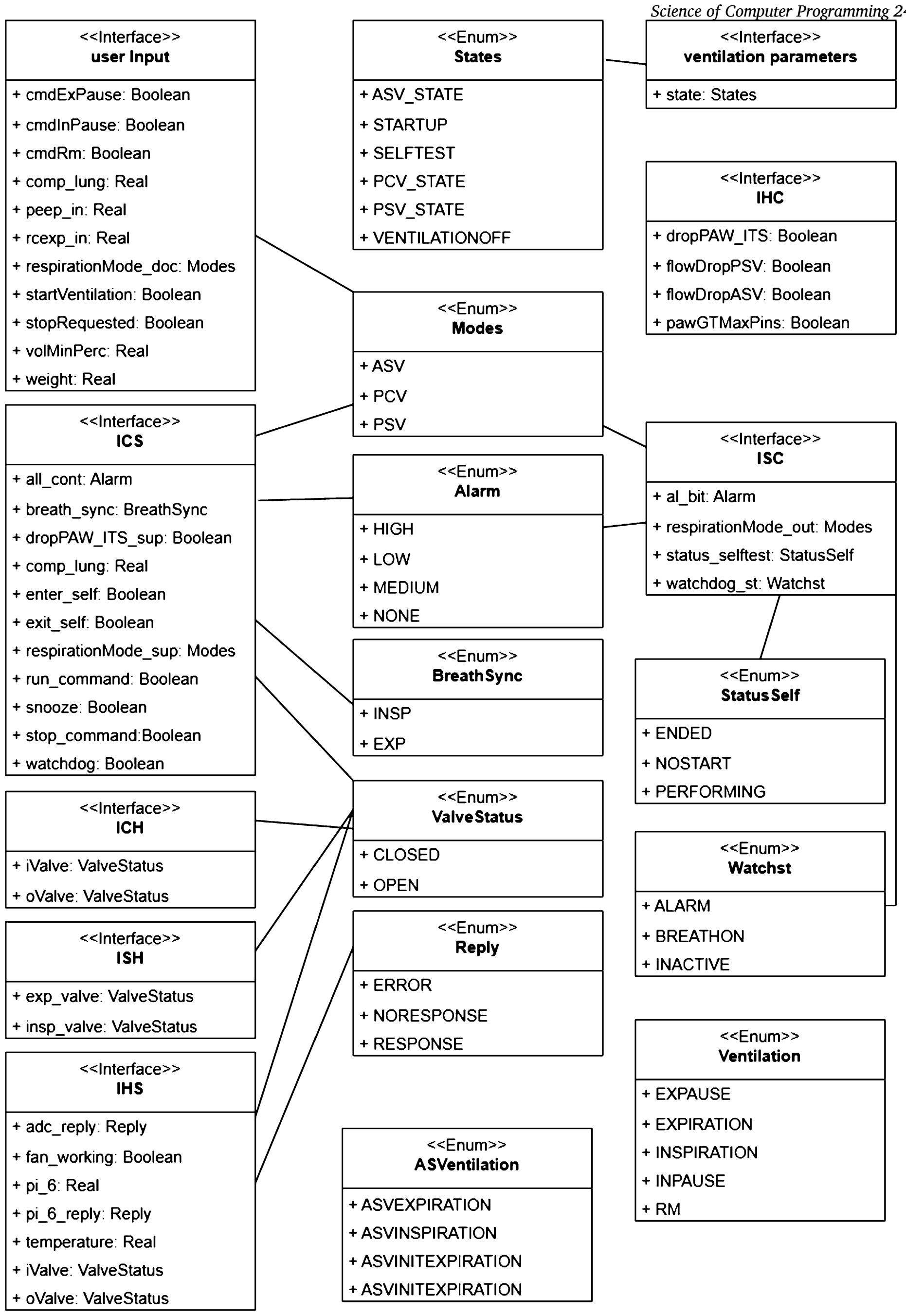


Fig. 3. MVM-Adapt interfaces.

perature measured, pi_6_reply and pi_6 respectively checks the status of the airway pressure sensor and returns the value of the measured airway pressure.

I_{SH} contains the signals used to set the input (inps_valve) and output (exp_valve) valve status. Since both controller and supervisor set the value of valves, in case of normal functioning (no errors in the system) the valves are set equal to the value sent by the controller, otherwise in case of errors the valves are set equal to the value sent by the supervisor.

4.3. Behavior of components

In the following sections, we describe the behavior of the two main components, controller and supervisor, with their adaptive features.

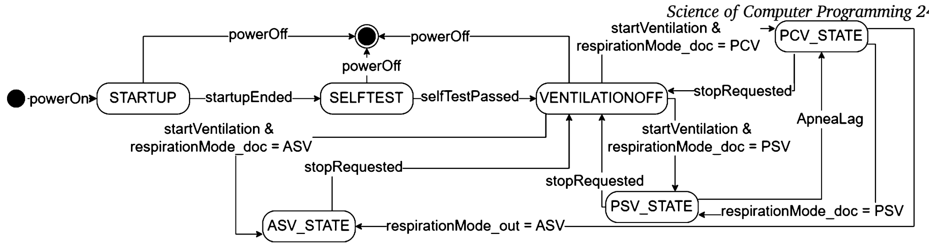


Fig. 4. MVMController state machine.

4.3.1. *Adaptive Controller*

The model of the controller is an extension of the adaptive version presented in [11]. The novelty is that ASV ventilation mode can be activated by a decision taken by the supervisor, as we explain in Sect. 4.3.2.

The introduction of changing the ventilation mode from PCV to ASV by the supervisor under certain conditions does not affect the controller state machine. As shown in the state machine in Fig. 4 (that intuitively outlines the facade behavior of the controller, though it is less precise than the ASM-based specification reported in Code 1 and Code 2), the initial state of the controller is STARTUP, and once the start-up procedures are completed (i.e. the monitored value startupEnded[5] is true), it sends the signal (enter_self) to the supervisor and moves to SELFTEST where it performs a sequence of tests ensuring that the hardware is fully functional (i.e. the system checks that the switchover from external to internal power works, there are no leaks in the breathing circuit, the flow meter is connected in the right direction, the expiratory valve is functional, the oxygen sensor is calibrated, local alarms are functional). In case the self-test procedure fails, the ventilator remains locked in SELFTEST state, and maintenance service is mandatory. Once the self-test procedure is concluded with a positive outcome (selfTestPassed is true), the controller is in state VENTILATIONOFF. The valves are put in a safe position (the input valve is closed and the output valve is opened) and the ventilation is off. Once the physician has updated alarm thresholds and ventilation parameters (if necessary), the ventilation mode can be selected (by respirationMode_doc input function) among PCV, PSV, or ASV mode, and the ventilation starts. The controller moves to the corresponding state (PCV_STATE, PSV_STATE or ASV_STATE), till the physician stops the ventilation (stopRequested); in this case, the controller returns to VENTILATIONOFF and notifies the change to the supervisor by the out function stop command. When the controller is in one of the three ventilation modes, the ventilation mode can be changed automatically or manually. The transition from PCV_STATE to PSV_STATE can only happen after the end of the inspiration if it is set by the physician. While the transition from PSV_STATE to PCV_STATE occurs automatically when the patient is not able to breathe on his/her own and he/she remains in apnea for a certain amount of time. The other automatic transition is from PCV_STATE to ASV_STATE, which is managed by the supervisor (see Sect. 4.3.2).

Code 1 reports, by using the `AsmetaL` textual modeling language, excerpts of I/O ASM `MVMcontroller`. The main rule r_Main shows the rule fired at each step, depending on the current state value.

The boolean conditions that allow the transition of the controller state machine in Fig. 4 from one state to the next, are incorporated as rule guards within the respective state change rules. For instance, consider the rule r_startup (refer to the right column in Code 1), responsible for effecting the transition from the STARTUP state to the SELFTEST state. This rule is guarded by the monitored condition startupEnded, which evaluates to true upon the completion of the starting phase and the initialization of ventilator parameters with default values.

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 (here missed), and the expiration, r_runPCVExp[6] (line 18). 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[7]). When the expiration time is passed (line 20), the controller checks if the supervisor has changed ventilation mode to ASV (line 21), and if so the controller sets its state to ASV_STATE by means of r_ASV rule (line 23). In that case, the controller moves to the ASV inspiration phase, then it is checked if the expiration pause has been requested (line 27). In the affirmative case, expiration pause is executed, otherwise, the controller moves to the PCV inspiration phase (line 30). If the expiration time has not passed yet, but the trigger window has expired and are sudden drop in pressure happened (line 34), the controller checks if the supervisor has changed ventilation mode to ASV (line 35). In that case, the ventilation mode ASV is activated. Otherwise, a new PCV inspiration phase starts (line 42).

The controller transitions to ASV_STATE through two pathways (refer to Fig. 4): either when the physician selects this ventilation mode for the patient via the GUI (using the signal respirationMode_doc), or if the supervisor decides to switch the ventilation mode from PCV to ASV based on ventilation parameters (via the signal respirationMode_out).

The rule r_runASV in Code 2 oversees the management of the ASV ventilation mode through a set of rules (not provided here[8]). These rules are activated to execute based on the value of the phaseASV function.

[5] This monitored function is used to abstract the procedures executed internally by the controller and it holds true when the procedure is completed.

[6] The square brackets in the ASMETA notation are used in a macro call rule to pass parameters if there are any [18].

[7] To model the timers we use the ASMETA library for times [22].

[8] All models and analysis artifacts can be accessed at https://github.com/asmeta/asmeta_based_applications/tree/main/MVM/MVM%20Adapt%20Compositional%20Io%20ASM.

```
 asm MVMcontroller
 signature:
enum domain States = {STARTUP | SELFTEST
 | VENTILATIONOFF | PCV_STATE | PSV_STATE
 | ASV_STATE}
enum domain Modes = {PCV | PSV | ASV}
    ...
dynamic controlled state: States
dynamic controlled phase: Ventilation
dynamic monitored respirationMode_doc: Modes
dynamic monitored respirationMode_out: Modes
dynamic monitored stopRequested: Boolean
dynamic monitored startupEnded: Boolean
dynamic monitored dropPAW_ITS: Boolean
dynamic monitored cmdExPause: Boolean
dynamic out iValve: ValveStatus
dynamic out oValve: ValveStatus
dynamic out respirationMode_sup: Modes
dynamic out enter_self: Boolean
dynamic out dropPAW_ITS_sup: Boolean
    ...

 definitions:
 ...
 rule r_startup = ...
 rule r_selftest = ...
 rule r_ventilationoff = ...
 rule r_runPCV = ...
 rule r_runPCVInsp = ...
 rule r_runPCVExp = ...
 rule r_exPause = ...
 rule r_PCVStartInsp = ...
 rule r_runPSV = ...
 rule r_runASV = ...

 main rule r_Main =
 par
    if state = STARTUP then  r_startup[]  endif
    if state = SELFTEST then r_selftest[] endif
    if state = VENTILATIONOFF then
        r_ventilationoff[] endif
    if state = PCV_STATE then r_runPCV[] endif
    if state = PSV_STATE then r_runPSV[] endif
    if state = ASV_STATE then r_runASV[] endif
 endpar
```

```
default init s0:
 function state = STARTUP

rule r_startup =
  if startupEnded then
   par
    state := SELFTEST
    enter_self := true
   endpar
  endif

rule r_runPCV =
 par
  if phase = INSPIRATION then r_runPCVInsp[] endif
  if phase = EXPIRATION then r_runPCVExp[] endif
 endpar

rule r_runPCVExp =
 ...
if expired(timerExpirationDurPCV) then
 if respirationMode_out = ASV then
  par
   r_ASV[]
   respirationMode_sup := respirationMode_out
  endpar
 else
  if cmdExPause then
   r_exPause[]
  else
   r_PCVStartInsp[]
  endif
 endif
else if expired(timerTriggerWindowDelay) and
dropPAW_ITS then
 if respirationMode_out = ASV then
  par
   r_ASV[]
   respirationMode_sup := respirationMode_out
  endpar
 else
  par
   r_PCVStartInsp[]
   dropPAW_ITS_sup := true
  endpar
 endif endif
```

Code 1: Adaptive Controller model in the ASMETA textual notation (on the left), and definitions of some transition rules (on the right).

```
rule r_runASV =
  par
   if phaseASV = ASVINITINSPIRATION then r_runASVInitInsp[] endif
   if phaseASV = ASVINITEXPIRATION then r_runASVInitExp[] endif
   if phaseASV = ASVINSPIRATION then r_runASVInsp[] endif
   if phaseASV = ASVEXPIRATION then r_runASVExp[] endif
  endpar
```

Code 2: Adaptive Controller in ASV mode.

If the ventilation is not stopped, this last variable, initialized when ventilation in adaptive mode starts, follows the cycle ASVINITINSPIRATION → ASVINITEXPIRATION → ASVINITINSPIRATION three times to deliver three initial test breaths to measure the actual breath pattern and compare it to the target values. Then it follows the cycle ASVINSPIRATION → ASVEXPIRATION → ASVINSPIRATION where the tidal volume and respiratory rate are adapted in order to get the target. When the ventilator operates in ASV mode, the controller performs the following actions: it sets the in and out valves to allow the patient's inspiration (resp. expiration), resets the timers to compute the duration of the next respiratory (insp/exp) phases, computes the target ventilation parameters – the target volume of area to be inspired/expired and the target respiratory rate – by suitable equations[9] that guarantee safe ventilation for the patient according to the Otis curve [15], and communicates the current patient's respiration mode to the supervisor (by out function breath_sync).

[9] Note that these complex formulas have been modeled, but they are not shown here to keep simple the presentation of the case study.

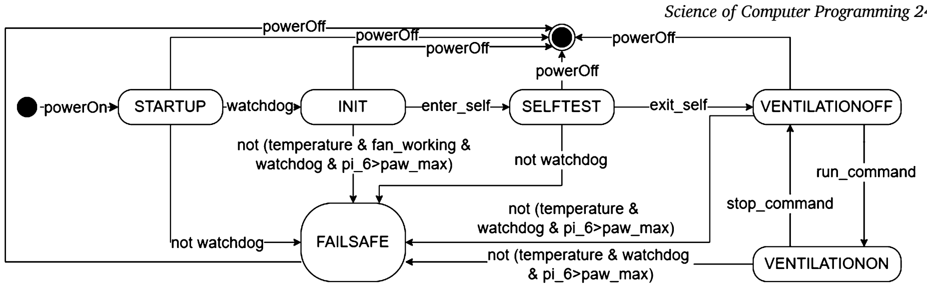


Fig. 5. Supervisor state machine.

```
asm supervisorAdapt

signature:

enum domain StatesSup = {STARTUP | INIT | SELFTEST | VENTILATIONON | VENTILATIONOFF | FAILSAFE}
enum domain Modes = {PCV | PSV | ASV}
enum domain Reply = {RESPONSE | ERROR | NORESPONSE}
enum domain Watchst = {INACTIVE | BREATHON | ALARM}
enum domain ValveStatus = {OPEN | CLOSED}
dynamic monitored adc_reply: Reply
dynamic monitored pi_6_reply: Reply
dynamic monitored fan_working: Boolean
dynamic monitored respirationMode_sup: Modes
dynamic out watchdog_st: Watchst
dynamic out respirationMode_out: Modes
dynamic controlled insp_valve: ValveStatus
dynamic controlled exp_valve: ValveStatus
dynamic controlled state: StatesSup
derived temperatured: Boolean
derived watch: Boolean
derived pi_6_time: Boolean
derived otis_boundaries: Boolean
derived breath_triggered: Boolean
...
```

Code 3: Signature of the adaptive Supervisor in the ASMETA textual notation.

4.3.2. *Adaptive Supervisor*

As shown in the state machine in Fig. 5, the supervisor goes through a sequence of six states. Some states align with those in the controller machine, mirroring the configuration of both components during the operation of the MVM-Adapt, as the STARTUP phase, the SELFTEST, and the VENTILATIONOFF. The transitions between states are guided by the main rule shown in Code 4, while the function's signature is reported in Code 3.

When the ventilator is turned on, the supervisor initializes parameters with default values while in STARTUP state.[10] Upon receiving the watchdog signal from the controller, it transitions to the INIT state through the r_startup rule (refer to line 11 in Code 4). In the INIT state, the supervisor conducts checks, including device temperature, pressure level, fan operation, and communication with the controller using the r_init rule. If no errors or inconsistencies are detected, and the controller signals the successful completion of the startup phase and the commencement of the self-test phase (via the enter_self signal), the supervisor enters the SELFTEST state. Subsequently, it initiates a sequence of hardware tests using the r_selftest rule (see line 3 in Code 4). Upon receiving confirmation from the controller that the self-test phase has concluded (via the exit_self signal), the supervisor transitions to the VENTILATIONOFF state, awaiting the start of ventilation (see line 13). In this state, the supervisor verifies that temperature and pressure levels are within permissible ranges and ensures active communication with the controller. Upon receiving the run_command by the controller, the supervisor shifts to the VENTILATIONON state. While overseeing patient ventilation, executing the r_ventilation_on rule (refer to line 14), the supervisor manages alarms and monitors ventilation parameters to ensure they remain within predefined thresholds. Additionally, if stop_ command is received from the controller, the supervisor reverts to the VENTILATIONOFF state.

When the supervisor is in one of its states (INIT, VENTILATIONOFF/ON, STARTUP, SELFTEST), and problems are revealed on the hardware or in the controller, the rule r_failsafe (see lines 16 and 19) is executed and the supervisor moves to the FAILSAFE state. Simultaneously, the ventilator undergoes a safety configuration with the in-valve closed, out-valve open, and alarms activated to prevent potential harm to the patient.

In the adaptive version, the supervisor is also responsible for communicating the controller to change the ventilation mode from PCV to ASV in order to help the patient to use his/her lungs as much as possible. During ventilation in PCV mode, this mode change can happen under two different circumstances:

[10] The state function of the supervisor is an internal function, for this reason, it is not reported in Fig. 3.

```
 1  main rule r_main =
 2   par
 3    if state = SELFTEST then  r_selftest[] endif
 4        if (state != SELFTEST and state != FAILSAFE) then
 5          par
 6           r_check_adc[]
 7           r_check_pi6[]
 8            if (adc_reply = RESPONSE) and (pi_6_reply = RESPONSE) then
 9                 if (fan_working) then
10                   par
11                    if state = STARTUP then r_startup[]  endif
12                    if state = INIT then r_init[] endif
13                    if state = VENTILATIONOFF then r_ventilation_off[] endif
14                    if state = VENTILATIONON then r_ventilation_on[] endif
15                   endpar
16                 else r_failsafe  endif
17      endif endpar endif  endpar
18
19  macro rule r_failsafe =
20      par
21          state := FAILSAFE
22          insp_valve := CLOSED
23          exp_valve := OPEN
24          watchdog_st := ALARM
25          ...
26      endpar
27
28  macro rule r_ventilation_on =
29      if(temperatured) and (watch) and (pi_6_time) then
30      ...
31          par
32              r_check_sync[]
33              r_check_v[]
34              ...
35          endpar
36      else
37          r_failsafe[]
38      endif
39
40  macro rule r_check_sync =
41      if not(expired(timer_breath)) then
42       par
43        r_breath_transition[]
44              ...
45       endpar
46      else
47          r_failsafe[]
48      endif
49
50  macro rule r_breath_transition =
51      ......
52      if (respirationMode_sup = PCV and (not(otis_boundaries) or breath_triggered)) then
53          respirationMode_out := ASV
54      else
55      respirationMode_out := respirationMode_sup
56      endif
57      ...
```

Code 4: Adaptive Supervisor in the ASMETA textual notation.

- If the patient tries to autonomously trigger inspiration for three consecutive times (the condition breath_triggered at line 52 of Code 4 holds, i.e., the signal dropPAW_ITS_sup was received three consecutive times from the controller).
- If the current volume of the breathed area and the respiratory rate are out of the Otis boundaries (the guard otis_boundaries is false).

In these cases, the supervisor requires the controller to change the ventilation mode from PCV to ASV (by updating the out function respirationMode_out to ASV, see line 53). The rule r_breath_transition, responsible for this ventilation mode change, is invoked by the rule r_ventilation_on when MVM-Adapt is ventilating and no stop ventilation is requested (besides other conditions here not described).

4.3.3. Component interaction scenarios

According to the given interfaces and assuming that no alarms are raised, the interaction protocols between the controller and the supervisor for the Breath Triggered scenario and the Otis scenario are shown in Fig. 6 and Fig. 7, respectively, by using a UML-like notation of sequence diagrams. We convey that the send action by a component A to a component B is depicted by an arrow from A to B labeled with information of the form <sng_1=val_1;sng_2=val_1;...>, being sng_i an interface signal between A and B, and val_i the value

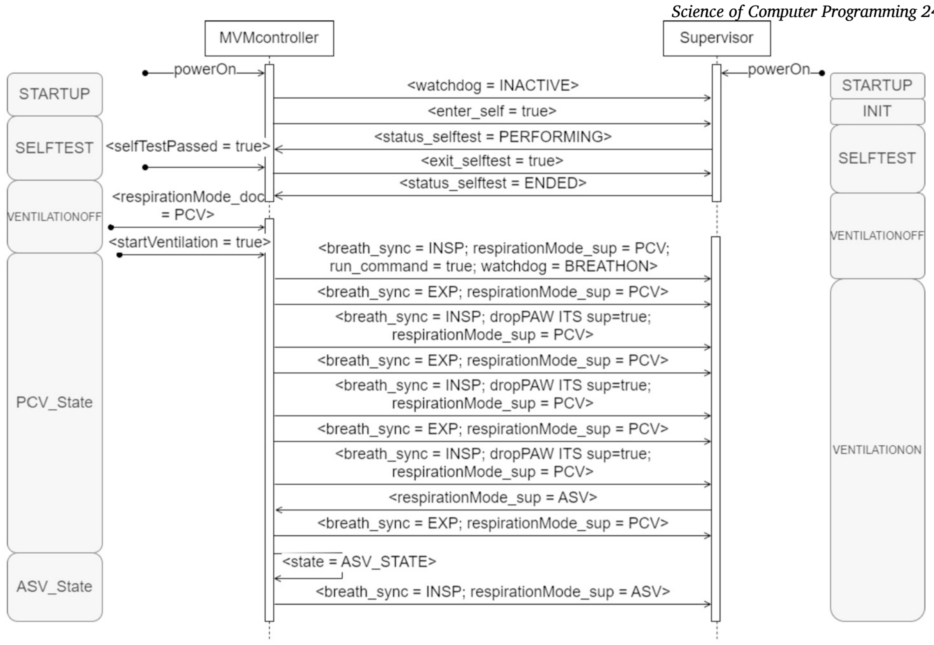

MVMcontroller
Supervisor
powerOn
STARTUP
INIT
SELFTEST
VENTILATIONOFF
PCV_State
ASV_State
VENTILATIONON
<watchdog = INACTIVE>
<enter_self = true>
<status_selftest = PERFORMING>
<selfTestPassed = true>
<exit_selftest = true>
<status_selftest = ENDED>
<respirationMode_doc = PCV>
<startVentilation = true>
<breath_sync = INSP; respirationMode_sup = PCV; run_command = true; watchdog = BREATHON>
<breath_sync = EXP; respirationMode_sup = PCV>
<breath_sync = INSP; dropPAW ITS sup=true; respirationMode_sup = PCV>
<respirationMode_sup = ASV>
<state = ASV_STATE>
<breath_sync = INSP; respirationMode_sup = ASV>

Fig. 6. Interaction diagram - Breath triggered scenario.

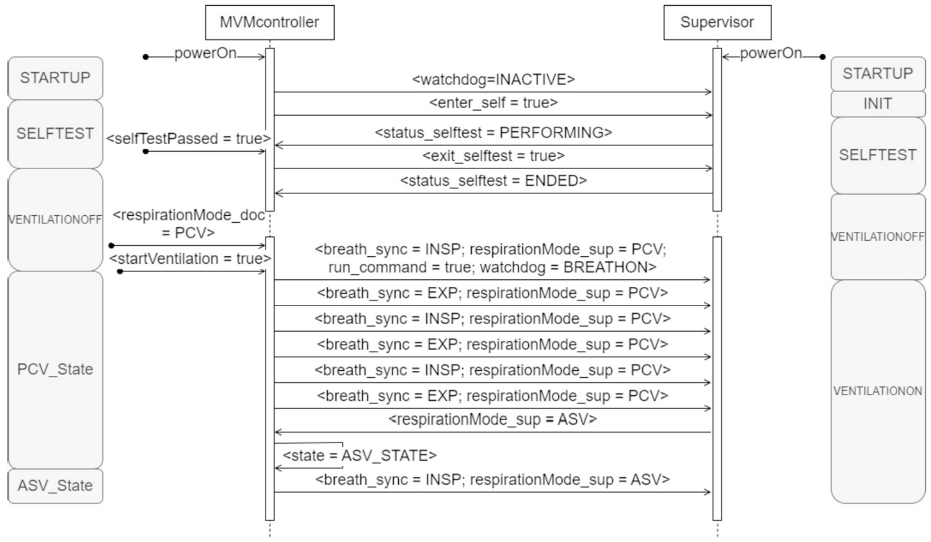

MVMcontroller
Supervisor
powerOn
STARTUP
INIT
SELFTEST
VENTILATIONOFF
PCV_State
ASV_State
VENTILATIONON
<watchdog=INACTIVE>
<enter_self = true>
<status_selftest = PERFORMING>
<selfTestPassed = true>
<exit_selftest = true>
<status_selftest = ENDED>
<respirationMode_doc = PCV>
<startVentilation = true>
<breath_sync = INSP; respirationMode_sup = PCV; run_command = true; watchdog = BREATHON>
<breath_sync = EXP; respirationMode_sup = PCV>
<breath_sync = INSP; respirationMode_sup = PCV>
<respirationMode_sup = ASV>
<state = ASV_STATE>
<breath_sync = INSP; respirationMode_sup = ASV>

Fig. 7. Interaction diagram - Otis scenario.

that A communicates to B for that signal. The state of the controller and supervisor during the interaction is shown respectively on the left and right of the diagram.

After switching on the system, the initial phase of the diagram is the same in both scenarios. The controller sends the watchdog signal to the supervisor and communicates when it enters and exits (after receiving the signal that the self-test has been passed) the self-test mode. Specularly, the supervisor responds when performing and ending the self-test. Then the user selects the respiration mode, PCV in both scenarios, and starts the ventilation. The controller sends the type of ventilation, the start signal, and the breathing phase to the supervisor. From this point, the two scenarios act differently.

In the breath trigger scenario (see Fig. 6) the controller sends the breathing phase and the ventilation mode to the supervisor. Moreover, when in the inspiration phase and the patient triggers a spontaneous breath, it sends the dropPAW_ITS_sup signal. When the supervisor counts three consecutive spontaneous breaths, it reports to the controller to change the ventilation mode from PCV to ASV. The controller changes its ventilation mode (it sets the state to ASV_STATE) at the beginning of the next respiratory cycle. Then it sends the signal to the supervisor with the breathing phase and the respiration mode, which now is ASV.

In the Otis scenario (see Fig. 7), the controller sends the breathing phase and the ventilation mode. The supervisor continually computes the respiratory rate and the current volume of the breathed area. When they are out of the Otis boundaries, it sends a signal to the controller to change ventilation mode from PCV to ASV. Then the controller acts as explained for the breathing scenario.

4.4. Composition formula

The two main execution scenarios of the MVM-Adapt system we consider here, the *Breath triggered Scenario* and the *Otis Scenario*, refer to the two ways in which the supervisor adapts the controller's behavior to change the ventilation mode from PCV to ASV.

The compositional formula to orchestrate the co-simulation of the I/O ASM assembly of the MVM-Adapt in Fig. 2 is:

$$GUI <|> (MVMcontroller <|> Supervisor) <|> HW$$

However, both scenarios can be reproduced by restricting the co-simulation to the controller and supervisor components. Therefore, a full-duplex bidirectional pipe:

$$MVMcontroller <|> Supervisor$$

is enough to reflect the supervisory interaction between the controller and the high-level controller (the supervisor), since this last continually receives information from the MVM controller and closes it.

5. Component models: validation and verification

Once the model has been specified, model validation and verification (V&V) are to be performed to guarantee requirements satisfaction. Each of these analysis activities has different goals and requires different efforts.

Model validation is a lightweight form of model analysis and is usually applied to ensure that the specification reflects the user's needs and statements about the system. It allows for detecting faults in the model as early as possible with limited effort. ASM model validation is possible by model *simulation* using the `AsmetaS` simulator. It can operate *interactively*, i.e., the user inserts the value of monitored functions, or *randomly*, i.e., the tool randomly chooses the value of monitored functions among those available, and reports the trace of the model run as a sequence of states. A tabular presentation of a simulation is possible using the model animator `AsmetaA`, which shows the models' execution through the use of tables. A further technique for model validation is by *construction of scenarios* reflecting expected system behaviors. The tool `AsmetaV` executes scenarios (written in the language `AVALla`) and checks whether the machine runs as expected; it also reports the rule coverage, namely which rules are executed to perform the scenarios.

Model verification. Validation usually precedes the application of more expensive and accurate methods, like verification of properties (e.g., safety, liveness, or reachability properties), that should be applied only when a designer has enough confidence that the specification captures all informal requirements. Formal verification of ASM models is possible by model-checking of temporal properties (expressed both in CTL or LTL).

In the following sections, we report some details of the validation and verification activities performed on the MVM-Adapt components. More precisely, we show how to carry out both activities on the controller component, while for the supervisor we report the validation results and the sequence of properties stated to check the correctness of the component's behavior.

5.1. Validation and verification of the Adaptive Controller Component

Model validation While modeling the controller, we performed validation activities in terms of animation (simulation traces) and scenario construction.

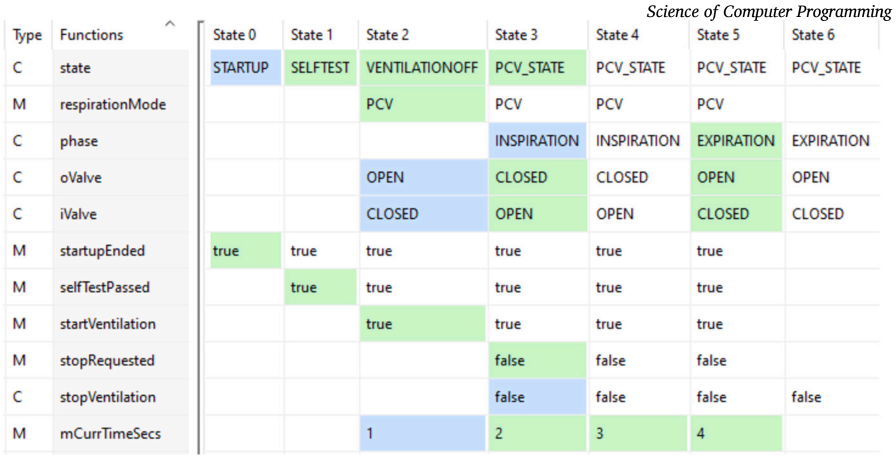
An example of *animation* is reported in Fig. 8. The animation is displayed in a table: the vertical axis lists the locations' functions – marked as controlled (C) and monitored (M)–, while the horizontal axis represents the evolution of the execution by showing the sequence of states, namely the value of each location in the state. The controller, after performing startup and self-test, is in the "ventilation off" state. As expected, the input valve is closed and the output valve is opened. When the "start PCV" command is sent to the controller, the PCV mode starts from the inspiration phase, and the valves are moved to the expected position: the input valve is opened and the output valve is closed. After the inspiration duration, the ventilator is in the expiration phase, the input valve is closed while the output valve is opened.

After that, we wrote *scenarios*, directly derived from the requirements specification document [13,6], to check, whenever it is needed, that the desired behavior is captured by the model. The difference with simple animation is that scenarios can be considered documents of reproducible behavior.

Code 5 reports a scenario where at each **step** of the machine we check the ventilator state (**check** state) and the position of the input (**check** iValve) and output (**check** oValve) valves, given the inputs received (**set**).[11]

By using this technique of scenario construction, we checked and documented critical behaviors of the controller in each state of its state machine. The most critical ones were those checking correct behavior in case of controlled ventilation (e.g., a respiration

[11] **set**, **step** and **check** are Avalla commands to, respectively, provide value of monitored functions, make a step of the model, check for location values, i.e., the state of the ASM.



Type	Functions	State 0	State 1	State 2	State 3	State 4	State 5	State 6
C	state	STARTUP	SELFTEST	VENTILATIONOFF	PCV_STATE	PCV_STATE	PCV_STATE	PCV_STATE
M	respirationMode			PCV	PCV	PCV	PCV	
C	phase				INSPIRATION	INSPIRATION	EXPIRATION	EXPIRATION
C	oValve			OPEN	CLOSED	CLOSED	OPEN	OPEN
C	iValve			CLOSED	OPEN	OPEN	CLOSED	CLOSED
M	startupEnded	true	true	true	true	true	true	
M	selfTestPassed		true	true	true	true	true	
M	startVentilation			true	true	true	true	
M	stopRequested				false	false	false	
C	stopVentilation				false	false	false	false
M	mCurrTimeSecs			1	2	3	4	

Fig. 8. PCV animation example.

```
check state = STARTUP;
set startupEnded := true;
step
check state = SELFTEST;
set selfTestPassed := true;
step
check state = VENT..OFF;
set startVentilation := true;
set respirationMode := PCV;
step
```

```
check state = PCV_STATE;
check oValve = CLOSED;
check phase = INSPIRATION;
check iValve = OPEN;
step
check state = PCV_STATE;
check oValve = CLOSED;
check phase = INSPIRATION;
check iValve = OPEN;
step
```

```
check state = PCV_STATE;
check oValve = OPEN;
check phase = EXPIRATION;
check iValve = CLOSED;
step
check state = PCV_STATE;
check oValve = OPEN;
check phase = EXPIRATION;
check iValve = CLOSED;
check stopVentilation = false;
```

Code 5: PCV scenario example.

cycle starts with inspiration; if a signal of inspiration/expiration pause is received then the phase is updated accordingly) or supported ventilation (e.g., if no spontaneous breath is detected within the apnea time window then the operation mode must switch from PSV to PCV).

Model verification By the model checker, we have verified the following safety properties guaranteeing that the patient cannot choke in case of active ventilation and that when the ventilator is off, the relief valve allows the patient to breathe[12]:

- When ventilation is off, the output valve is open and the input valve is closed:

  LTLSPEC g(state = VENTILATIONOFF implies (iValve = CLOSED and oValve = OPEN))

- When ventilation is on, in the inspiration phase, the output valve is closed and the input valve is open:

  LTLSPEC g((phase = INSPIRATION and (state = PCV_STATE or state = PSV_STATE)) implies (iValve = OPEN and oValve = CLOSED))

- When ventilation is on, in the expiration phase, the output valve is open and the input valve is closed:

  LTLSPEC g((phase = EXPIRATION and (state = PCV_STATE or state = PSV_STATE)) implies (iValve = CLOSED and oValve = OPEN))

- In case of inspiration/expiration pause, both valves are closed:

  LTLSPEC g(((phase = INPAUSE or phase = EXPAUSE) and (state = PCV_STATE or state = PSV_STATE)) implies (iValve = CLOSED and oValve = CLOSED))

- Necessary condition for both valves to be closed is that a breath pause happens:

  LTLSPEC g((iValve = CLOSED and oValve = CLOSED) implies ((phase = INPAUSE or phase = EXPAUSE) and (state = PCV_STATE or state = PSV_STATE)))

Liveness properties have been stated to guarantee the correct system reaction to user input commands. For example, two of the properties we have verified are:

[12] Other properties, not reported here, refer to the recruiting maneuver, i.e. the emergency procedure required after intubation consisting of prolonged lung inflation to reactivate the alveoli.

- When the user starts the ventilation in PSV mode or moves from PCV to PSV mode, in the future the ventilator will support the patient during the ventilation:

```
LTLSPEC g(((state = VENTILATIONOFF or state = PCV_STATE) and
   respirationMode_doc = PSV) implies f (state = PSV_STATE))
```

- When the user stops the ventilation, in the future the ventilator will stop:

```
LTLSPEC g((state = VENTILATIONON and stopRequested = true) implies
   f (state = VENTILATIONOFF))
```

5.2. Validation and verification of the Adaptive Supervisor component

Model validation We have validated the model of the adaptive supervisor by building scenarios (as shown for the controller component) reproducing the critical behaviors of the component in each state of its state machine:

STARTUP**:** five scenarios checking errors or lack of communication with the other components, and if any, guarantee that the component moves to the FAILSAFE configuration (e.g., if the fan is not working (fan_working is false) the supervisor move to a safe configuration).
INIT**:** four scenarios to check correct behavior in case no malfunction is revealed and the change to a safe configuration in case of over-limit temperature and/or communication errors.
SELFTEST**:** two scenarios to check correct behavior in case of no emergency and the change to a safe configuration, otherwise.
VENTILATIONOFF**:** five scenarios to mainly check that the inspiration valve is closed and the expiration valve is open since in this state there is no ventilation.
VENTILATIONON**:** eight scenarios to mainly check that the valves are set correctly since in this state the ventilation is active (both if the phase is inhalation or exhalation).

Model verification The two following main properties have been checked on the model to guarantee safe operation for the patient[13]:

- If the maximum time of one breath has elapsed then the system must be in the failsafe state and at that point the supervisor can no longer change state:

```
LTLSPEC g(expired(timer_breath) implies f(g(state = FAILSAFE)))
```

- If the supervisor is in failsafe mode then the valves must be set in such a way that the inhalation valve is closed and the exhalation valve is open:

```
LTLSPEC g(state = FAILSAFE implies f(g(insp_valve = CLOSED and
            exp_valve = OPEN)))
```

As for the controller component, liveness properties can be verified for the supervisor. For example, the following one guarantees the correct reaction of the supervisor to the *stop* user input commands (received by the supervisor from the controller):

```
LTLSPEC g((state = VENTILATIONON and stop_command = true) implies
   f (state = VENTILATIONOFF))
```

6. System validation by model co-simulation

This section reports the two relevant co-simulation scenarios of the MVM-Adapt system models as described in prose in Section 2 and as components interaction in Section 4.3.3. The composition formula, as already presented in Section 4.4, is specified by the setup command for the machine assembly in the script of each scenario (see lines 1 and 2 in Code 6 and 8, respectively).

Breath triggered scenario The scenario setup and run are shown in the command script fragment in Code 6, and the resulting co-simulation trace is shown in Code 7. The run of the assembly starts in a configuration where the ventilation is in PCV mode (respirationMode_doc=PCV) and the patient is able to breathe on his/her own (dropPAW_ITS=true) (see setting of the highlighted parameters at line 4 in Code 6[14]). Code 7 shows the relevant step 10 of the co-simulation (see the highlighted location updates): the controller communicates to the supervisor that the patient is able to breathe spontaneously (dropPAW_ITS_sup=true) and that the current ventilation mode is PCV (respirationMode_doc=PCV). The supervisor, upon detecting that the patient has breathed spontaneously three consecutive times (count_dropPAW=3), communicates the controller to change the ventilation mode in ASV (respirationMode_out=ASV).

[13] Other properties concern the correct control of the alarms; although the interfaces with the hardware components have been reported, this behavior has been omitted here.

[14] We do not report here the setting of other parameters in the starting configuration of this scenario to keep the description of the case study simple, which is enough complex; abstracting from these parameters does not prevent following the results of the model co-simulation. The same remark holds for the run trace shown in Code 7.

```
1  init −n 2
2  setup MVMComposition as MVMcontrollerAdapt.asm <|> supervisorAdapt.asm
3  ...
4  run(MVMComposition, {adc_reply_m = RESPONSE;cmdExPause = false;cmdInPause = false;cmdRm = false;comp_lung = 0.06;dropPAW_ITS_ASV = false;
       dropPAW_ITS = true; fan_working_m = true;flowDropPSV = false;flowDropASV = false;limPressure = 45.0;mCurrTimeSecs = 13;pawGTMaxPinsp = false;
       peep_in = 5.0;pi_6_m = 25;pi_6_reply_m = RESPONSE;rcexp_in = 2.0; respirationMode_doc = PCV; startupEnded = true;selfTestPassed = true;
       startVentilation = false;stopRequested = false;temperature_m = 15;vol_eff = 0.9;volMinPerc = 100.0;weight = 60.0;})
```

Code 6: Breath triggered Scenario Co-simulation Setting.

```
1  ....
2  [step:10 of MVMcontrollerAdapt]
3  Model execution outcome: SAFE
4  Updated locations: {compliance = 0.06;vol_min_perc = 100.0;run_command = true;duration(timerInspirationDurPCV) = 2;exit_self = true;peep = 5.0;enter_self =
       true;iValve = OPEN;dropPAW_ITS_sup = true;respirationMode_sup = PCV;duration(timerTriggerWindowDelay) = 1;temperature = 15;breath_sync = INSP;
       stopVentilation = false;stop_command = false;duration(timerExpirationDurPCV) = 4;limit_pa = 45.0;mCurrSecs = 13;oValve = CLOSED;phase =
       INSPIRATION;start(timerExpirationDurPCV) = 12;rcexp = 2.0;fan_working = true;all_cont = NONE;watchdog = true;start(timerTriggerWindowDelay)
       = 12;weight_sup = 60.0;state = PCV_STATE;start(timerInspirationDurPCV) = 13;adc_reply = RESPONSE;current_volume = 0.9;pi_6_reply = RESPONSE;pi_6
       = 25}
5  Out locations: {mCurrSecs = 13;oValve = CLOSED;compliance = 0.06;vol_min_perc = 100.0;run_command = true;rcexp = 2.0;fan_working = true;all_cont = NONE;
       watchdog = true;weight_sup = 60.0;exit_self = true;peep = 5.0;enter_self = true;iValve = OPEN;
       dropPAW_ITS_sup = true; respirationMode_sup = PCV; temperature = 15; breath_sync = INSP; stop_command = false;adc_reply = RESPONSE;
       current_volume = 0.9;pi_6_reply = RESPONSE;limit_pa = 45.0;pi_6 = 25}
6  Execution time (in milliseconds): 9 ms
7
8  [step:10 of supervisorAdapt]
9  Model execution outcome: SAFE
10 Updated locations: {status_selftest = ENDED;min_rr = 4.0;time_insp = 2;max_attempts_pi_6 = 0;start(timer_insp) = 13;time_exp = 1;start(timer_exp) = 12;al_bit =
       NONE; count_dropPAW = 3; insp_valve = OPEN;peak_min = 20;respirationMode_out = ASV;count_ie = 0;peak_max = 25;duration(timer_breath) = 62;f_prev
       = 9.126789049141507;start(timer4secondPassed) = 13;max_attempts_adc = 0;previous_breath = INSP;start(timer_breath) = 13;watchdog_st = BREATHON
       ;peep_min = 20;peep_max = 25;ver_rr = 20.0;count_rr_min = 0;count_rr_max = 3;exp_valve = CLOSED;start(timer1secondPassed) = 13;state =
       VENTILATIONON;duration(timer4secondPassed) = 4;rr_supp = 20.0;count_ppmax = 0;max_rr = 10.0;count_pkmin = 0;count_pkmax = 0;duration(
       timer1secondPassed) = 1}
11 Out locations: {al_bit = NONE; respirationMode_out = ASV; status_selftest = ENDED;watchdog_st = BREATHON}
12 Execution time (in milliseconds): 14 ms
```

Code 7: Breath triggered Scenario Co-simulation Run.

```
1  init −n 2
2  setup MVMComposition as MVMcontrollerAdapt.asm <|> supervisorAdapt.asm
3  ...
4   run(MVMComposition, {adc_reply_m = RESPONSE;cmdExPause = false;cmdInPause = false;cmdRm = false;comp_lung = 0.06;dropPAW_ITS_ASV = false;
       dropPAW_ITS = false;fan_working_m = true;flowDropPSV = false;flowDropASV = false;limPressure = 45.0;mCurrTimeSecs = 10;pawGTMaxPinsp = false;
       peep_in = 5.0;pi_6_m = 25;pi_6_reply_m = RESPONSE;rcexp_in = 0.39; respirationMode_doc = PCV; ;startupEnded = true;selfTestPassed = true;
       startVentilation = true;stopRequested = false;temperature_m = 15;vol_eff = 0.9;volMinPerc = 100.0;weight = 60.0;})
```

Code 8: Otis Scenario Co-simulation Setting.

Note that in addition to the locations values and execution time (in ms) of an ASM run step, the echo messages produced by the co-simulation engine in the trace also include the model execution outcome (*SAFE* or *UNSAFE*) to report on whether there has been or not a failure in the ASM run step (e.g., due to an invariant violation, an inconsistent location update, an ill-formed input, etc.). In case an I/O ASM model fails its run step, the composition expression fails, and the faulty ASM model and all models already executed in the composition are rolled back to their previous safe state [8].

Otis scenario The run of the assembly starts in a configuration where the ventilation is in PCV mode (respirationMode_doc=PCV at line 4 in Code 8). The run in Code 9 shows that the controller sends the supervisor (line 5) the signal watchdog (set to true to establish communication) and the current patient's respiratory parameters (compliance, vol_min_perc, etc.). The supervisor, after receiving all the values from the controller and checking (by a complex computation not reported here) if the current respiratory values of the patient are within the Otis boundaries, detects that the respiration rate (f_prev) is too high and therefore requests the controller to change mode to ASV (respirationMode_out=ASV at line 11 in Code 9).

7. Related work

The architectural graphical notation for illustrating I/O ASM assemblies with location ports was inspired by UML-based component and service-oriented architecture modeling languages, such as the UML component notation itself [23] and the UML4SOA profile[15] for modeling service behavior and service protocols in UML.

[15] https://uml4soa.eu/profile/index.html.

```
...
[step:7 of MVMcontrollerAdapt]
Model execution outcome: SAFE
Updated locations: {compliance = 0.06;vol_min_perc = 100.0;run_command = true;duration(timerInspirationDurPCV) = 2;exit_self = true;peep = 5.0;enter_self =
        true;iValve = OPEN;dropPAW_ITS_sup = false;respirationMode_sup = PCV;duration(timerTriggerWindowDelay) = 1;temperature = 15;breath_sync = INSP;
        stopVentilation = false;stop_command = false;duration(timerExpirationDurPCV) = 4;limit_pa = 45.0;mCurrSecs = 10;oValve = CLOSED;phase =
        INSPIRATION;start(timerExpirationDurPCV) = 6;rcexp = 0.39;fan_working = true;all_cont = NONE;watchdog = true;start(timerTriggerWindowDelay) = 6;
        weight_sup = 60.0;state = PCV_STATE;start(timerInspirationDurPCV) = 10;adc_reply = RESPONSE;current_volume = 0.9;pi_6_reply = RESPONSE;pi_6
        = 25}
Out locations: {mCurrSecs = 10;oValve = CLOSED; compliance = 0.06; vol_min_perc = 100.0; run_command = true; rcexp = 0.39; fan_working = true;all_cont =
        NONE; watchdog = true; weight_sup = 60.0; exit_self = true; peep = 5.0; enter_self = true;iValve = OPEN;dropPAW_ITS_sup = false;
        respirationMode_sup = PCV ;temperature = 15; breath_sync = INSP; stop_command = false;adc_reply = RESPONSE; current_volume = 0.9; pi_6_reply =
        RESPONSE; limit_pa = 45.0 ;pi_6 = 25}
Execution time (in milliseconds): 15 ms

[step:7 of supervisorAdapt]
Model execution outcome: SAFE
Updated locations: {status_selftest = ENDED;min_rr = 4.0;time_insp = 2;max_attempts_pi_6 = 0;start(timer_insp) = 10;time_exp = 4;start(timer_exp) = 6;al_bit =
        NONE;count_dropPAW = 0;insp_valve = OPEN;peak_min = 20;respirationMode_out = ASV;count_ie = 0;peak_max = 25;duration(timer_breath) = 62;
        f_prev = 19.942066018199334; start(timer4secondPassed) = 10;max_attempts_adc = 0;previous_breath = INSP;start(timer_breath) = 10;watchdog_st =
        BREATHON;peep_min = 20;peep_max = 25;ver_rr = 10.0;count_rr_min = 0;count_rr_max = 0;exp_valve = CLOSED;start(timer1secondPassed) = 10;state =
        VENTILATIONON;duration(timer4secondPassed) = 4;rr_supp = 10.0;count_ppmax = 0;max_rr = 10.0;count_pkmin = 0;count_pkmax = 0;duration(
        timer1secondPassed) = 1}
Out locations: {al_bit = NONE; respirationMode_out = ASV ;status_selftest = ENDED;watchdog_st = BREATHON}
Execution time (in milliseconds): 17 ms
```

Code 9: Otis Scenario Co-simulation Run.

Our work on compositional model simulation was inspired by model-based approaches to workflow modeling and service orchestration, such as tools for the Business Process Model and Notation (BPMN) [24], and the Jolie language [25], as well as to multi-state machine modeling, as Yakindu statecharts tools [26]). However, our approach is much more oriented to distributed model-based system simulation/prototyping. It could be used, for example, in practical contexts where model simulation is required at runtime and models have to be co-simulated in tandem with real systems, such as runtime models that are part of the knowledge base of self-adaptive and autonomous systems [4] or of a *digital twin* plant [27,28].

In a distributed setting, *choreography automaton* for the choreographic modeling of communicating systems is introduced in [29] as a system of communicating finite state machines whose transitions are labeled by synchronous/asynchronous interactions. The projection of a choreography automaton yields a system of communicating finite-state machines, which are proven to be live as well as lock- and deadlock-free. Usually, choreographies are suitable approaches to describe modern software systems such as micro-services, but in our compositional simulation mechanism we preferred to rely on orchestration, namely on a centralized synchronous communication semantics that is typically much more common in IT service orchestration and automation platforms.

On the use of ASMs in component- and service-based architectures, they have been used for service behavior modeling and prototyping, in conjunction with the OASIS/OSOA standard Service Component Architecture (SCA) for heterogeneous service assembly. In such a SCA-ASM framework, abstract implementation (or prototype) of SCA components in ASM are co-executed *in place* with other component implementations [30]. In [31], 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. Discussion and lesson learned

The development of the MVM-Adapt system model allowed us to highlight the challenges underlying the engineering of this type of systems and at the same time to gain some experience on the practical use of the compositional simulation technique for I/O ASMs. As already anticipated in [8], where a comparison between the monolithic and compositional approaches is given in terms of well-known architectural design principles (like divide-and-conquer, cohesion, decoupling, etc.) and simulation examples, there are strengths and limitations in using a compositional modeling approach. Specifically, for the MVM-Adapt system case, we would like to emphasize the following main lessons:

L0 – Modeling Formal Method ASMs offer several advantages w.r.t. other automaton-based formalisms. The main and unique characteristic is the formal representation of a state as a mathematical algebra, whose functions can change interpretation from state to state by applying transition rules. Modeling by using arbitrary structures, defined in terms of n-ary functions (with arbitrary $n \geq 0$) over infinite domains, offers a greater flexibility w.r.t. other state-based transition systems (e.g., labeled transition systems [32], Kripke structures [33], etc.), where states consist of a set of variables (0-ary functions) with values. Although the models of the MVM components we show here do not immediately expose such a feature (since their excerpts mainly contain definitions of 0-ary controlled functions), we exploited this so-called "freedom of abstraction" in several parts of the specification, for example, for modeling the timer operation and more complex components' operations as that of the controller in ASV mode. In the first case, Code 10 reports the ASM module for the specification of a timer and the relevant functions for dealing with timers' operations: *start* a timer, *reset* a timer, set the *duration* of a timer, determine timer *expiration*. More details on the Timer library are in [22]. For the second case, consider, for example, the definition of the rule `r_computeOtisEq`, reported

```
module TimeLibrary
import StandardLibrary
export *
signature:
 abstract domain Timer
 // starting time taken from the local clock
 controlled start: Timer−> Integer
 // duration in ms of the timer starting from its start
 controlled duration: Timer −> Integer
 //get the current time for the timer
 derived currentTime : Timer−> Integer
 // is the timer expired?
 derived expired: Timer −> Boolean
 // use seconds as time unit
 monitored mCurrTimeSecs: Integer
definitions:
 function currentTime($t in Timer) = mCurrTimeSecs
 function expired($t in Timer) = (currentTime($t) >= start($t) + duration($t))
 // restart the timer
 macro rule r_reset_timer($t in Timer) = start($t) := currentTime($t)
 // change or set the duration of a timer
 macro rule r_set_duration($t in Timer, $ms in Integer) =  duration($t) := $ms
```

Code 10: Timer Library.

```
1     asm MVMcontrollerAdapt
2     ...
3     signature:
4         domain Hist subsetof Integer
5         ...
6         dynamic monitored rcexp_in: Real
7         static a_coeff: Real
8         derived vd: Real
9         derived volume_min: Real
10        dynamic controlled freq_hist: Hist−>Real //memory for frequencies
11  dynamic controlled vol_hist: Hist−>Real //memory for volumes
12    ...
13    definitions:
14        domain Hist = {0:7}
15        function a_coeff=2.0*pwr(3.14,2.0)/60.0
16        function vd=2.2*weight/1000.0
17        function volume_min = weight*0.1*volMinPerc/100.0
18    ...
19         rule r_computeOtisEq =
20           let ($x =(pwr(1.0+2.0*a_coeff*rcexp_in*(volume_min−freq_hist(previous)*vd)/vd,(1/2))−1.0)/(a_coeff*rcexp_in)) in
21             par
22             freq_hist(numCycle mod 8) :=$x
23             vol_hist(numCycle mod 8) := volume_min/$x
24           endpar
25          endlet
26            ...
```

Code 11: Calculation of Optimal respiratory rate and volume.

in Code 11, invoked at any respiratory cycle by the rule `r_runASVExp` (see Code 2 - line 6). The rule `r_computeOtisEq` computes the optimal respiratory rate and volume during ASV ventilation mode. As reported in [15], the optimal respiratory rate is calculated using the following formula (see Code 11 - line 20):

$$freq = \frac{\sqrt{1.0{+}2.0{*}a\_coeff{*}rcexp\_in{*}\frac{volume\_min{-}freq\_hist(previous){*}vd}{vd}}-1.0}{a\_coeff{*}rcexp\_in}$$

where
- `a_coeff` (see Code 11 - line 15) is a factor based on the flux waveform: $a\_coeff = \frac{2*\pi^2}{60}$;
- `vd` (see Code 11 - line 16) is the death space, given the body weight in grams: $vd = 2.2 * weight/1000$;
- `volume_min` (see Code 11 - line 17) is the minute volume calculated using the percentage of minute volume (`volMinPerc`) and the body weight: $volume\_min = weight * 0.1 * volMinPerc/100$;
- `rcexp_in`[16] is the constant breathing time calculated as $volume/flowratio$.

Once the optimal respiratory rate is computed, the optimal volume (see Code 11 - line 23) is calculated as $volume = volume\_min/freq$.

[16] Note that the model does not know the flow ratio, for this reason, we have considered this function as a function provided by the environment.

Then the value of optimal volume and respiratory rate are stored in functions `freq_hist` and `vol_hist` (see Code 11 - lines 22 and 23), whose last eight values are used to compute the total respiratory rate and the total volume to monitor trends over time. E.g., the total respiratory rate is given by the function `freq_target` defined in the specification as:

```
dynamic controlled freq_target: Real
```

updated at the end of each expiratory cycle as:

```
freq_target := compute_freq_target((freq_hist(0) + freq_hist(1) + freq_hist(2) + freq_hist(3) + freq_hist(4) + freq_hist(5) + freq_hist(6) + freq_hist(7))/8.0)
```

and computed by exploiting the derived function:

```
derived compute_freq_target: Real − > Real
function compute_freq_target ($freqmean in Real) =
          if ($freqmean > (20.0/rcexp_in)) then
           if (20.0/rcexp_in) > 60.0 then 60.0
           else 20.0/rcexp_in
           endif
          else
           if ($freqmean < 5.0) then 5.0
           else $freqmean
           endif
          endif
```

Besides, the freedom of abstraction feature, other advantages offered by the ASMs that we have exploited in our experience are: (1) the *pseudo-code format* of a model, so that ASMs can be used for high-level programming; (2) *executability of models*, which allows for model validation; (3) the concept of *model refinement*, which is embedded into the ASM formal approach and allows for facing the complexity of system specification by starting with a high-level description of the system and then adding further details till a desired level of specification has been reached; (4) the concept of ASM *modularization*, i.e., an ASM without the main firing rule, facilitates model scalability and separation of concerns, so tackling the complexity of big system specification.

L1 – Compositional Modeling From our modeling experience with the MVM-Adapt system, we have learned that dividing a system model into two or more sub-models and loosely coupling their simulation through a co-simulation mechanism brings several advantages w.r.t. simulating a single (monolithic) system model.

- *Flexibility in team working* The non-adaptive MVM subsystems and their abstract models/code prototypes have been developed by different groups from the very beginning, for different engineering domains and target running platforms (e.g., the MVM controller prototype on the Arduino board). This compositional modeling approach allowed us a high degree of flexibility in managing the separation of the modeling/analysis tasks and allocating such tasks to different modeling/development groups, each working in parallel at their speed.
- *Model reusability and fast prototyping* Moving from MVM to MVM-Adapt, we kept the same compositional modeling approach, which allowed us to speed up the overall formal requirements specification. The model of the controller and that of the supervisor were reused and refined to add the ASV operation mode by two different groups; each component model was analyzed (i.e., validated and verified) separately to prove its correctness according to the settled I/O interfaces with the other component models; finally, the overall MVM-adapt behavior was validated by co-simulating the I/O ASMs of the components.
- *Enhanced requirements system comprehension* In addition, compositional modeling and the proposed simulation technique helped to clarify to ourselves the documented system requirements and the underlying communication protocol among the MVM-Adapt subsystems. However, defining the I/O interfaces in terms of function bindings between the ASM sub-models and expressing the causality relation between such sub-models in terms of a composition simulation formula is not always an easy task. It requires a clear understanding of how the involved subsystems react to I/O events and the interaction protocol.

L2 – Verification vs Validation V&V are the two half-parts of the quality-assurance process of a software system, however, historically, formal methods and supporting analysis tools have focused much more on verification, while validation has received less attention. In general, however, formal verification proves to be very costly compared to validation, which is less time and resource consuming. Moreover, validation may be the only viable solution, as systems grow rapidly in complexity and size. In our case, the inner complexity of the internal behavior of the MVM-Adapt system makes difficult the adoption of the model checker to prove properties on the overall integrated system, and therefore would require a compositional approach too.

Both activities helped us to unveil errors in the model. No particular errors were discovered in the informal requirements, but modeling them helped us clarify some ambiguities (as stated in the last point of lesson L1). More precisely, scenario-based validation was used to guide the development, and so constantly leads to adjustments of the developed models, while verification allowed us to discover more subtle model errors.

For example, scenario-based validation helped us to discover an error in the model when the supervisor had to force the ventilation mode from PCV to ASV; in particular, the values of the optimal frequency were not those expected because the formula to compute the minute volume uses the weight in the g unit, while the body weight received as an external parameter was in Kg.

Instead, during the verification, we discovered a controller operation error that occurred when the clinician requested the PSV ventilation mode while the ventilator was in PCV. The property

```
LTLSPEC g(((state = VENTILATIONOFF or state = PCV_STATE) and
   respirationMode_doc = PSV) implies f(state = PSV_STATE))
```

failed because the required mode transition occurred before the inspiration timer expired, although the specification document required the transition to occur after the inspiration timer expired.

L3 – Model scalability Another aspect to still explore is the scalability of the approach, and the feasibility of expressing a relatively simple composition formula for co-simulating models of large-scale complex systems into a federated, interoperable simulation environment. In such cases, choreographies would be perhaps more appropriate.

L4 – Model evolvability A compositional approach to system modeling also allows to accommodate new or evolving requirements and have a more evolvable system model. In our case, we could easily implement the new adaptive mode over the base MVM system model since we simply reused a model of the MVM system based on compositional I/O ASM models and worked for an extension of this model starting from the I/O ASM assembly (the architectural model) and the fine-tuning of the I/O interfaces for modeling the new behavioral facets and interaction scenarios.

9. Conclusion and future directions

In this paper, we have shown the concept of I/O ASM and the practical application of a compositional simulation technique for I/O ASMs to the MVM-Adapt case study. This validation technique is supported by the ASMETA tool `AsmetaComp`, which is intended for allowing distributed simulation of ASMs: separate ASM system models can be connected and co-simulated together to provide simulations of integrated systems by scenarios, or a big ASM model can be divided into smaller sub-models that co-execute, possibly on separate local or remote processes/computers.

In the future, we want to conduct more experiments to evaluate the benefits of a compositional simulation w.r.t. simulation of one single monolithic model. These include the evaluation of the usability of the technique [34] and the understandability of the execution traces. We plan to investigate the fault-detection capability of the proposed approach to guarantee that using composition does not reduce the ability to discover unacceptable behaviors.

A further future step is that of enriching the set of composition operators and defining choreography constructs to deploy and enact a choreography-based execution of asynchronous I/O ASMs.

In addition to compositional validation by scenarios (through co-simulation of I/O ASMs), we would also like to investigate on the definition of a compositional verification technique to model check properties of interest based on a composition formula of I/O ASMs.

A more challenging future research line concerns the integration of the compositional simulation of I/O ASMs with other different simulation techniques in the context of Digital Twins and of heterogeneous cyber-physical systems. In particular, 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 for distributed co-simulation [35], such as the IEEE 1516 High Level Architecture (HLA) and the Functional Mockup Interface (FMI) [36].

CRediT authorship contribution statement

Silvia Bonfanti: Writing – review & editing, Writing – original draft, Validation, Methodology, Formal analysis. **Elvinia Riccobene:** Writing – review & editing, Writing – original draft, Validation, Methodology, Formal analysis. **Patrizia Scandurra:** Writing – review & editing, Writing – original draft, Validation, Methodology, Formal analysis.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

We acknowledge the financial support of the project PRIN 2022 SAFEST (Trust assurance of Digital Twins for medical cyber-physical systems), funded by the European Union - Next Generation EU, Mission 4, Component 2, Investment 1.1, CUP F53D23004230006, under the National Recovery and Resilience Plan (NRRP) – Grant Assignment Decree No. 959 adopted on 30 June 2023 by the Italian Ministry of University and Research (MUR).

References

[1] C. Talcott, S. Ananieva, K. Bae, B. Combemale, R. Heinrich, M. Hills, N. Khakpour, R. Reussner, B. Rumpe, P. Scandurra, H. Vangheluwe, Composition of Languages, Models, and Analyses, Springer, 2021, pp. 45–70.
[2] J.Á. Bañares, J.M. Colom, Model and simulation engines for distributed simulation of discrete event systems, in: Economics of Grids, Clouds, Systems, and Services, Springer International Publishing, Cham, 2019, pp. 77–91.
[3] D. Weyns, M.U. Iftikhar, Model-based simulation at runtime for self-adaptive systems, in: S. Kounev, H. Giese, J. Liu (Eds.), 2016 IEEE International Conference on Autonomic Computing, ICAC 2016, IEEE Computer Society, 2016.
[4] N. Bencomo, S. Götz, H. Song, Models@run.time: a guided tour of the state of the art and research challenges, Softw. Syst. Model. 18 (5) (2019).
[5] A. Fuller, Z. Fan, C. Day, C. Barlow, Digital twin: enabling technologies, challenges and open research, IEEE Access 8 (2020) 108952–108971.
[6] A. Abba, et al., The novel mechanical ventilator Milano for the COVID-19 pandemic, Phys. Fluids 33 (3) (2021) 037122.

[7] S. Bonfanti, A. Gargantini, E. Riccobene, P. Scandurra, Compositional simulation of Abstract State Machines for safety critical systems, in: S.L.T. Tarifa, J. Proença (Eds.), Formal Aspects of Component Software - 18th International Conference, FACS 2022, Proceedings, in: LNCS, vol. 13712, Springer, 2022, pp. 3–19.

[8] S. Bonfanti, A. Gargantini, E. Riccobene, P. Scandurra, A compositional simulation framework for Abstract State Machine models of Discrete Event Systems, Formal Aspects of Computing, 2024.

[9] E. Börger, A. Raschke, Modeling Companion for Software Practitioners, Springer, Berlin, Heidelberg, 2018.

[10] P. Arcaini, A. Bombarda, S. Bonfanti, A. Gargantini, E. Riccobene, P. Scandurra, The ASMETA Approach to Safety Assurance of Software Systems, Springer International Publishing, Cham, 2021, pp. 215–238.

[11] S. Bonfanti, E. Riccobene, D. Santandrea, P. Scandurra, Modeling the MVM-adapt system by compositional I/O abstract state machines, in: U. Glässer, J. Creissac Campos, D. Méry, P. Palanque (Eds.), Rigorous State-Based Methods, Springer Nature, Cham, 2023, pp. 107–115.

[12] A. Bombarda, S. Bonfanti, A. Gargantini, E. Riccobene, Developing a prototype of a mechanical ventilator controller from requirements to code with ASMETA, Electron. Proc. Theor. Comp. Sci. 349 (2021) 13–29, https://doi.org/10.4204/eptcs.349.2.

[13] S. Bonfanti, A. Gargantini, The mechanical lung ventilator case study, in: S. Bonfanti, A. Gargantini, M. Leuschel, E. Riccobene, P. Scandurra (Eds.), Rigorous State-Based Methods, Springer Nature, Switzerland, Cham, 2024, pp. 281–288.

[14] J. Fernández, D. Miguelena, H. Mulett, J. Godoy, F. Martinón-Torres, Adaptive support ventilation: state of the art review, Indian J. Crit. Care Med. 17 (1) (2013) 16–22, https://doi.org/10.4103/0972-5229.112149.

[15] Hamilton Medical, Operator's manual 610862/05 Software version 3.44d2015-09-24.

[16] A.B. Otis, W.O. Fenn, H. Rahn, Mechanics of breathing in man, J. Appl. Physiol. 2 (11) (1950) 592–607, https://doi.org/10.1152/jappl.1950.2.11.592.

[17] ASMETA (ASM mETAmodeling) toolset, https://asmeta.github.io/.

[18] A. Bombarda, S. Bonfanti, A. Gargantini, E. Riccobene, P. Scandurra, ASMETA tool set for rigorous system design, in: A. Platzer, K.Y. Rozier, M. Pradella, M. Rossi (Eds.), Formal Methods, Springer Nature, Switzerland, Cham, 2025, pp. 492–517.

[19] A. Cimatti, E.M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, NuSMV 2: an OpenSource tool for symbolic model checking, in: Proceedings of the 14th International Conference on Computer Aided Verification, CAV '02, Springer-Verlag, Berlin, Heidelberg, 2002, pp. 359–364.

[20] E. Riccobene, P. Scandurra, Model-based simulation at runtime with abstract state machines, in: Software Architecture - 14th European Conference, ECSA 2020 Tracks and Workshops, Proceedings, in: Communications in Computer and Information Science, vol. 1269, Springer, 2020.

[21] A. Gargantini, E. Riccobene, P. Scandurra, A metamodel-based language and a simulation engine for abstract state machines, J. Univers. Comput. Sci. 14 (12) (2008).

[22] A. Bombarda, S. Bonfanti, A. Gargantini, E. Riccobene, Extending ASMETA with time features, in: A. Raschke, D. Méry (Eds.), Rigorous State-Based Methods - 8th International Conference, ABZ 2021, Ulm, Germany, June 9-11, 2021, Proceedings, in: Lecture Notes in Computer Science, vol. 12709, Springer, 2021, pp. 105–111.

[23] M. Bjerkander, C. Kobryn, Architecting systems with UML 2.0, IEEE Softw. 20 (4) (2003) 57–61, https://doi.org/10.1109/MS.2003.1207456.

[24] OMG Business Process Model and Notation (July 2024), https://bpmn.org/.

[25] Jolie (July 2024), https://jolie-lang.org.

[26] YAKINDU Statechart Tools (July 2024), https://itemis.com/en/yakindu/state-machine.

[27] M. Grieves, J. Vickers, Origins of the Digital Twin Concept, vol. 8, Florida Institute of Technology, 2016.

[28] M.M. Bersani, C. Braghin, A. Gargantini, R. Mirandola, E. Riccobene, P. Scandurra, Engineering of trust analysis-driven digital twins for a medical device, in: T. Batista, T. Bures, C. Raibulet, H. Muccini (Eds.), Software Architecture. ECSA 2022 Tracks and Workshops - Revised Selected Papers, in: Lecture Notes in Computer Science, vol. 13928, Springer, 2022, pp. 467–482.

[29] S. Orlando, V.D. Pasquale, F. Barbanera, I. Lanese, E. Tuosto, Corinne, a tool for choreography automata, in: G. Salaün, A. Wijs (Eds.), Formal Aspects of Component Software - 17th Int. Conference, FACS 2021, Proceedings, in: LNCS, vol. 13077, Springer, 2021, pp. 82–92.

[30] E. Riccobene, P. Scandurra, A formal framework for service modeling and prototyping, Form. Asp. Comput. 26 (6) (2014) 1077–1113, https://doi.org/10.1007/s00165-013-0289-0.

[31] R. Mirandola, P. Potena, E. Riccobene, P. Scandurra, A reliability model for service component architectures, J. Syst. Softw. 89 (2014) 109–127, https://doi.org/10.1016/j.jss.2013.11.002.

[32] D.A. Peled, Software Reliability Methods, Springer Verlag, 2001.

[33] M. Huth, M. Ryan, Logic in Computer Science: Modelling and Reasoning About Systems, Cambridge University Press, 2004.

[34] P. Arcaini, S. Bonfanti, A. Gargantini, E. Riccobene, P. Scandurra, Addressing usability in a formal development environment, in: e.a. Emil Sekerinski (Ed.), Formal Methods. FM 2019 International Workshops - Revised Selected Papers, Part I, in: Lecture Notes in Computer Science, vol. 12232, Springer, 2019, pp. 61–76.

[35] W. Huiskamp, T. van den Berg, Federated Simulations, Springer International Publishing, 2016, pp. 109–137.

[36] Functional Mock-up Interface (July 2024), https://fmi-standard.org/.