Developing a prototype of a mechanical ventilator controller from requirements to code with ASMETA

Andrea Bombarda
University of Bergamo, Bergamo, Italy
andrea.bombarda@unibg.it

Silvia Bonfanti
University of Bergamo, Bergamo, Italy
silvia.bonfanti@unibg.it

Angelo Gargantini
University of Bergamo, Bergamo, Italy
angelo.gargantini@unibg.it

Elvinia Riccobene
University of Milano, Milan, Italy
elvinia.riccobene@unimi.it

Rigorous development processes aim to be effective in developing critical systems, especially if failures can have catastrophic consequences for humans and the environment. Such processes generally rely on formal methods, which can guarantee, thanks to their mathematical foundation, model preciseness, and properties assurance. However, they are rarely adopted in practice.

In this paper, we report our experience in using the Abstract State Machine formal method and the ASMETA framework in developing a prototype of the control software of the MVM (Mechanical Ventilator Milano), a mechanical lung ventilator that has been designed, successfully certified, and deployed during the COVID-19 pandemic. Due to time constraints and lack of skills, no formal method was applied for the MVM project. However, we here want to assess the feasibility of developing (part of) the ventilator by using a formal method-based approach.

Our development process starts from a high-level formal specification of the system to describe the MVM main operation modes. Then, through a sequence of refined models, all the other requirements are captured, up to a level in which a C++ implementation of a prototype of the MVM controller is automatically generated from the model, and tested. Along the process, at each refinement level, different model validation and verification activities are performed, and each refined model is proved to be a correct refinement of the previous level. By means of the MVM case study, we evaluate the effectiveness and usability of our formal approach.

1 Introduction

To prevent catastrophic consequences for humans and the environment due to system failure or unsafe operation, safety-critical software requires development methods and processes that could lead to provably correct system operation [24, 25]. From long time the use of models and formal analysis techniques is highly demanded already at design-time to improve software quality and guarantee safety, reliability, and other desired qualities. However, the usage of formal methods in industrial projects is still limited [3, 20, 21] and practitioners are still skeptical in using formal methods since they are considered time-consuming approaches that do not fit into an agile continuous integration development process.

Besides the well-known lack of training, among the barriers to the adoption of formal methods, we can remark (i) the complexity of formal notations, (ii) the poor scalability, (iii) the lack of easy-to-use tools supporting modeling, validation, and verification activities at the design phase, and (iv) the gap between models and code. Formal approaches allowing model refinement would help the designer in facing the complexity of system requirements, and techniques of automatic code generation from models would allow for producing correct-by-construction code/artifacts of the system in a seamless manner from the requirements to the final implementation. We believe that some characteristics of formal methods would

Submitted to:
αFM 2021

also favor their use in the assurance process: models should possibly be executable for high-level design validation and endowed with properties verification mechanisms; operational approaches are more adequate than denotational ones to support (automatic) code generation from models and model-based testing. In principle, different methods and tools can be used to guarantee software safety and reliability; however, the integrated use of different tools around the same formal method is much more convenient than having different tools working on input models with their own languages.

Among the plethora of existing formal methods, the Abstract State Machines (ASMs) [17, 18] are a system engineering method that can guide the development of software systems seamlessly from requirements capture to their implementation. This is shown by the adoption of ASMs in a series of cases studies, as in [6, 8, 10, 13], to name a few. Although ASMs have a rigorous mathematical foundation – as transition systems that extend the Finite State Machines (FSMs) [17]–, ASMs can be understood as pseudo-code or virtual machines working over abstract data structures. Besides their *pseudo-code format*, (1) ASM models can be specified at any desired *level of abstraction* and are *executable models*; (2) *model refinement* is an embedded concept in the ASM formal approach; it allows facing the complexity of system specification by starting with a high-level description of the system and then proceeding step-by-step by adding further details till a desired level of specification has been reached; each refined model must be proved to be a correct refinement of the previous one, and checking of such relation can be performed automatically [4]; (3) the concept of ASM *modularization*, i.e., an ASM without the main firing rule, facilitates model scalability and separation of concerns, so tackling the complexity of big systems specification; (4) ASM-based modeling and analysis are supported by a set of tools that can be used in an integrated manner within the ASMETA (ASM mETAmodeling) [5, 11, 12] framework. ASMETA provides tools for specifying the executable behavior of a system, for checking properties of interest, specifying and executing validation scenarios, generating prototype code, etc.

During the COVID-19 pandemic, our research team was involved in the design, development, and certification of a mechanical lung ventilator called MVM (Mechanical Ventilator Milano)[1] [2]. The project started from an idea of the physicist Cristiano Galbiati, who was soon joined by dozens of physicists, engineers, physicians, and computer scientists from 12 countries around the world, including the authors of this paper. The team was able to realize a ventilator that is reliable, easily reproducible on a large scale, available in a short amount of time, and at a limited cost [22]. The MVM has obtained the FDA (Food and Drug Administration) Emergency Use Authorization (EUA) followed by similar authorizations issued by Health Canada and the CE marking as well. During the development of the software, no formal method has been applied mainly because of time constraints and a lack of developers' skills with any formal method. However, we wanted to assess the feasibility (and possibly also the limits) of developing (part of) the ventilator by using a formal method-based approach. Because we were involved in all the software development process and its certification (we wrote the requirements and were involved also in the coding and testing for the MVM), we have all the knowledge and expertise necessary to perform this experiment. In this paper, we report the practical experience of using ASM/ASMETA in modeling, analyzing, and encoding the control software of the MVM.

The paper is structured as follows. Sect. 2 presents the background regarding the ASMETA framework and Sect. 3 introduces the MVM case study. In Sect. 4 we present the modeling, verification, and validation activities performed for the case study. The automatic generation of C++ code and unit tests from the ASMETA specification is described in Sect. 5, while the deployment process on Arduino is described in Sect. 6. Sect. 7 discusses the pros and cons of the ASMETA approach and the aspects that could prevent or favor its use in practice and concludes the paper.

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

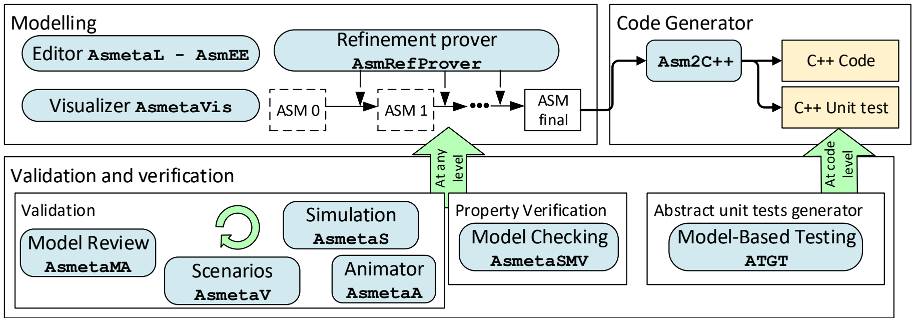


Figure 1: The ASM development process powered by the ASMETA framework

2 Abstract State Machines and ASMETA development process

The formal design of the MVM we propose here is based on the ASMs formal method [17, 18], which is an extension of FSMs where unstructured control states are replaced by states with arbitrarily complex data. The development process from formal requirement specification to code generation has been supported by the ASMETA framework [5], a set of tools around the ASMs, which we used for modeling the ventilator and performing validation and verification activities, together with automatic source code and tests generation.

ASM *states* are mathematical structures, i.e., domains of objects with functions and predicates (i.e., boolean functions) defined on them. The *transition* from one state s_i to the next state s_{i+1} is obtained by firing the set of all ASM *transition rules* invoked by a unique *main* rule, which is the starting point of a computation step. Transition rules express the modification of dynamic controlled functions interpretation from one state to the next one. Indeed, functions are classified as *static* (never change during any run of the machine) or *dynamic* (may change as a consequence of agent actions or *updates*). Dynamic functions are distinguished between *monitored* (only read by the machine and modified by the environment) and *controlled* (read in the current state and updated by the machine in the next state). *Derived* functions are not part of the state since they are defined in terms of other (dynamic) functions.

The *update* rule, as assignment of the form $f(t_1,\ldots,t_n) := v$, is the basic unit of rules construction, being f an n-ary function, t_i terms, and v the new value of $f(t_1,\ldots,t_n)$ in the next state. By a limited but powerful set of *rule constructors*, function updates can be combined to express other forms of machine actions as, for example, guarded actions (`if-then`) and simultaneous parallel actions (`par`).

The ASM method can facilitate the entire life cycle of software development, i.e., from modeling to code generation. Fig. 1 shows the development process based on ASMs and supported by the ASMETA framework[2] [11] which provides a set of tools to help the developer in the following activities: modeling, validation, verification, and, when required, code generation.

In the modeling phase, the user specifies the system models by using the `AsmetaL` language, and the editor `AsmetaXt` provides some useful editing support. ASM models can be read as pseudo-code over abstract data types. Moreover, the ASM visualizer `AsmetaVis` can be used to transform the textual model into graphs using the ASM graphical notation proposed in [4]. The refinement correctness can be automatically proved using the tool `ASMRefProver` [9].

The validation process is supported by the model simulator `AsmetaS`, the animator `AsmetaA`, the scenarios executor `AsmetaV` – all supporting models exposing time features [14] –, and the model re-

[2] `https://asmeta.github.io/`

viewer AsmetaMA. The simulator AsmetaS allows performing two types of simulation: interactive simulation (the user inserts the value of monitored functions) and random simulation (the tool randomly chooses the value of monitored functions among those available). A tabular presentation of a simulation is possible by means of the animator AsmetaA that shows the models' execution through the use of tables. AsmetaV executes scenarios written in Avalla language. Each scenario contains the expected system behavior and the tool checks whether the machine runs correctly. The model reviewer AsmetaMA performs the static analysis. It determines whether a model has sufficient quality attributes, e.g., minimality (the specification does not contain elements defined or declared in the model but never used), completeness (it requires that every behavior of the system is explicitly modeled), and consistency (it guarantees that locations are never simultaneously updated to different values).

Property verification is performed with the AsmetaSMV tool. It verifies if the properties derived from the requirements are satisfied by the models. When a property is verified, it guarantees that the model complies with the intended behavior. AsmetaSMV exploits the NuSMV and NuXmv model checkers. Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) properties can be proved by using the classical NuSMV model checker in the case of limited domains. NuXmv (an extension of NuSMV) is able to deal also with unlimited domains in the case of LTL properties and allows for proving properties based on *time* values.

In case the code is available, the ASMETA framework provides the ATGT tool that generates abstract unit tests starting from the ASM specification by exploiting the counterexample generation of a model checker (NuXMV or NuSMV). If not, a tool that automatically generates C++ code from ASMs is available (Asm2C++) [16].

3 MVM Case study

MVM [2] is an electro-mechanical ventilator equivalent to the old and reliable Manley Ventilator [26]. It is intended to provide ventilation support for patients that are in intensive therapy and that require mechanical ventilation. MVM works in pressure-mode, i.e., the respiratory time cycle of the patient is controlled by the pressure, and, therefore, this ventilator requires a source of compressed oxygen and medical air that are readily available in intensive care units. More precisely, MVM has two operative modes: *Pressure Controlled Ventilation* (PCV) and *Pressure Support Ventilation* (PSV). In the PCV mode, the respiratory cycle is kept constant and the pressure level changes between the target inspiratory pressure and the positive end-expiratory pressure. New inspiration is initiated either after a breathing cycle is over, or when the patient spontaneously initiates a breath. In the former case, the breathing cycle is controlled by two parameters: the respiratory rate (RR) and the ratio between the inspiratory and expiratory times (I/R). In the latter case, a spontaneous breath is triggered when the MVM detects a sudden pressure drop within the trigger window during expiration. The PSV mode is not suitable for patients that are not able to start breathing on their own because the respiratory cycle is controlled by the patient, and MVM partially takes over the work of breathing. A new respiratory cycle is initiated with the inspiratory phase, detected by the ventilator when a sudden pressure drop occurs. When the patient's inspiratory flow drops below a set fraction of the peak flow, MVM stops the pressure support, thus allowing exhalation. If a new inspiratory phase is not detected within a certain amount of time (apnea lag), MVM will automatically switch to the PCV mode because it is assumed that the patient is not able to breathe alone.

The ventilator allows the air to enter/exit through two valves, i.e., an input valve and an output valve. When the ventilator is not running, the valves are set to safe mode: input valve closed and output

declarations	# Functions				# Rules	
	mon.	con.	der.	static	decl.	rules
TimeLibrary	1	2	2	0	2	2
MVMController00	5	1	0	0	8	27
MVMController01	6	5	0	5	19	98
MVMController02	9	6	0	9	27	160
MVMController03	11	6	0	10	27	170

Table 1: Models dimension (including the TimeLibrary)
mon. = monitored, con. = controlled, der. = derived, decl. = macro rule declarations, rules = number of rules (inluding nested)

```
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
endpar
```

Code 1: MVMController00 main rule

valve opened. When the inspiration starts, the input valve is opened and the output valve is closed, while during the expiration the input valve is closed and the output valve is opened. Both in PCV and PSV mode inspiratory pause, expiratory pause, and recruitment manoeuvrer are allowed by user request. Inspiratory/Expiratory pause consists in closing the input and output valves of the ventilator respectively after the inspiration and expiration phase. The inspiratory pause allows measuring the pressure reached inside the alveoli at the end of the inspiratory cycle, while the expiratory pause allows measuring the residual pressure to check possible obstruction in the exhalation channel. Recruitment manoeuvrer is an emergency procedure required after intubation and it consists in prolonged lung inflation as necessary to reactivate the alveoli immediately; during this manoeuvrer, the input valve is opened and the output valve is closed.

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

To give an idea of the complexity of the entire MVM, its detailed behavior is described in the requirements documents which count altogether about 1000 requirements, each being a brief sentence. One document describes the behavior of the overall system, while 15 requirements documents describe the detailed behavior of software components. The controller itself has its own requirement document which consists of 31 pages and 157 requirements.

4 Modeling and V&V

In this section, we present the modeling of the MVM controller and the validation and verification activities we have performed. We proceeded through three refinement steps. (1) The first model (MVMController00) describes the transition between the main operation phases: startup, self-test, ventilation off, PCV, and PSV modes. (2) The second model (MVMController01) introduces the modeling of inspiration and expiration in both PCV and PSV, (3) while the third model (MVMController02) adds the expiratory/inspiratory pauses, the recruitment manoeuvrer, and the apnea. (4) The last refinement step (MVMController03) introduces (in both PCV and PSV) the transition between expiration and inspiration in case of pressure drop, and the transition between inspiration and expiration in case the pressure exceeds a threshold. The time features have been modeled using the TimeLibrary [14].

We have proved the refinement correctness by using the SMT-based tool `AsmRefProver` which proves the stuttering refinement as defined in [9].

In order to have an idea of the complexity of the ventilator models, Table 1 shows the models' dimensions in terms of the number of functions and rules.

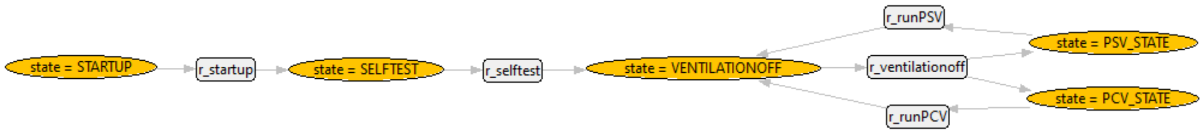


Figure 2: MVM state diagram

4.1 First model: MVMController00

The first model introduces the operation phases of the MVM controller. At the end of startup and self-test, the ventilator goes in the ventilation off state. Afterward, on the basis of the user request, it can go to one of the two operation modes: PCV or PSV. Code 1 shows the main rule of the controller in the first model. It specifies the transitions among the MVM states by setting the value of the state variable (initialized at the STARTUP value). Depending on the state value, the corresponding rule is executed.

The semantic visualization of the model, and in particular of the main rule, is shown in Fig. 2. It represents the MVM operation in terms of a control state machine: the value of the variable state is used as state mode to determine machine states.

4.2 Second model: MVMController01

The second model refines the inspiration and expiration phases in PCV and PSV mode. Code 2 shows the refinement of the rule r_runPCV, which here calls rules for the inspiration r_runPCVInsp (line 6) and the expiration r_runPCVExp (line 24) rules.

In PCV mode, the transition between inspiration and expiration is determined by the duration of each phase decided by the physician (when timers timerInspirationDurPCV, in case of inspiration, and timerExpirationDurPCV, in case of expiration, expire). When the inspiration time is passed (line 11), the controller goes to the PCV expiration phase (line 14). If the physician has required (by setting the value of the monitored function respirationMode) to move to PSV mode the machine changes the state from PCV to PSV and executes the rule r_PSVStartExp (line 15). If a stop request (by the monitored function stopRequested) is received during the inspiration phase, it is stored (in stopVentilation) and will be executed in the expiration phase (line 8). When in expiration, if no stop request is received, the ventilator moves to PCV inspiration when expiration duration expires (line 27).

In PSV mode (see Code 3), the transition from inspiration to expiration happens when the airflow drops a defined threshold (flowDropPSV holds, line 12) after a minimum inspiration time (timerMinInspTimePSV expires), or when the maximum inspiration time set by the doctor is expired. The opposite transition occurs after a minimum expiration time (timerMinExpTimePSV expires, line 19). The transition to ventilation off is allowed only from the expiration phase regardless of the stop command is received. Moreover, the physician can change from PSV to PCV and without interrupting the ventilation when in expiration phase (line 21).

Depending on the ventilator state, the input (iValve) and output (oValve) valves are in the following position: input valve is closed and output valve is open when the ventilator is not running or in the expiration phase, input valve is open and output valve is closed when it is in inspiration phase.

Model validation. While modeling the ventilator, we have performed validation activities: animation (simulation traces), model review, and validation. An example of animation is reported in Fig. 3, where the ventilator, after performing startup and self-test, is in the ventilation off state. As expected, the input

```
rule r_runPCV =
  par
    if phase = INSPIRATION then r_runPCVInsp[] endif
    if phase = EXPIRATION then r_runPCVExp[] endif
  endpar
rule r_runPCVInsp =
  par
    if not stopVentilation then
      if stopRequested then stopVentilation := true endif
    endif
    if expired(timerInspirationDurPCV) then
      par
        if respirationMode = PCV then
          r_PCVStartExp[] endif
        if respirationMode = PSV then
          par
            state := PSV_STATE
            r_PSVStartExp[]
          endpar
        endif
      endpar
    endif
  endpar
rule r_runPCVExp =
  if stopVentilation then r_stopVent[]
  else if stopRequested then r_stopVent[]
  else if expired(timerExpirationDurPCV) then
    r_PCVStartInsp[]
  endif endif  endif
rule r_PCVStartInsp =
  par
    phase := EXPIRATION
    iValve := CLOSED
    oValve := OPEN
    r_reset_timer[timerInspirationDurPCV]
  endpar
```

Code 2: MVMController01 PCV

```
rule r_runPSV =
  par
    if phase = INSPIRATION then r_runPSVInsp[] endif
    if phase = EXPIRATION then r_runPSVExp[] endif
  endpar
rule r_runPSVInsp =
  par
    if not stopVentilation then
      if stopRequested then stopVentilation := true endif
    endif
    if (expired(timerMinInspTimePSV) and flowDropPSV)
      or expired(timerMaxInspTimePSV) then
      r_PSVStartExp[]
    endif
  endpar
rule r_runPSVExp =
  if stopVentilation then r_stopVent[]
  else if stopRequested then r_stopVent[]
  else if expired(timerMinExpTimePSV) then
    par
      if respirationMode = PCV then
        par
          state := PCV_STATE
          r_PCVStartInsp[]
        endpar
      endif
      if respirationMode = PSV then r_PSVStartInsp[] endif
    endpar endif endif endif
rule r_PSVStartInsp =
  par
    phase := EXPIRATION
    iValve := CLOSED
    oValve := OPEN
    r_reset_timer[timerMinExpTimePSV]
    r_reset_timer[timerMaxInspTimePSV]
  endpar
```

Code 3: MVMController01 PSV

valve is closed and the output valve is opened. When the start PCV command is sent to the ventilator, 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.

For the validation phase, we have written scenarios to check, whenever it is needed, that the desired behavior is captured by the model. Code 4 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**).

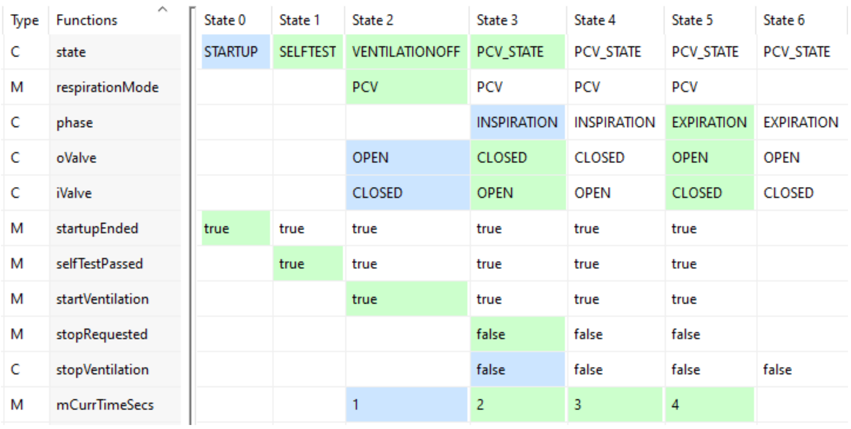
Model verification. Using the model checker, we have verified the following safety properties:

- Valves are never both open or closed at the same time:

  LTLSPEC not f(iValve=oValve)

- 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))



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	

Figure 3: PCV animation example

```
check state = STARTUP;
set startupEnded := true;
step
check state = SELFTEST;
set selfTestPassed := true;
step
check state = VENTILATIONOFF;
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 4: PCV scenario example

These two properties are crucial for the safety of the ventilator. The former guarantees that the patient can not choke, while the latter assures that the system is fail-safe since, when the ventilator is off, the relief valve allows the patient to breathe.

4.3 Third model: MVMController02

At the end of the inspiration phase, the physician can perform an inspiratory pause or recruitment maneuver, and the expiratory pause is allowed after the expiration phase. This has been modeled as shown in Code 5 for PCV mode and Code 6 for PSV mode, which respectively extend the behavior of the MVMController01 as shown in Code 2 and Code 3.

After inspiration, if an inspiratory pause is required (monitored function cmdInPause holds, Code 5 line 11 and Code 6 line 10) the valves are both closed for the entire pause duration. If inspiratory pause is not required, the doctor can select the recruitment maneuver (by setting cmdRm, Code 5 line 14 and Code 6 line 11) and the lungs are filled with oxygen and medical air.

An expiratory pause can be performed upon the doctor's request after the expiration phase (monitored function cmdExPause holds, Code 5 line 29, and Code 6 line 25).

In the expiratory and inspiratory pause states, the input and output valves are both closed, while in the recruitment maneuver the output valve is closed and the input valve is opened to allow the air flowing in the alveoli. Moreover, as shown in Code 6, when PSV is running and the ventilator does not detect a new breath within apnea lag (timer timerApneaLag expires, line 15), the ventilator automatically changes to PCV mode starting from the inspiration phase (line 32).

```
rule r_runPCV =
  par ...
    if phase = INPAUSE then r_runInPause[] endif
    if phase = RM then r_runRm[] endif
    if phase = EXPAUSE then r_runExPause[] endif
  endpar
rule r_runPCVInsp = ...
  if expired(timerInspirationDurPCV) then
  par
      if respirationMode = PCV then
        if cmdInPause then
          r_InPause[]
        else
          if cmdRm then  r_rm[]
          else r_PCVStartExp[]
        endif
      endif endif
      if respirationMode = PSV then
        par
          state := PSV_STATE
          r_PSVStartExp[]
          r_resetApneaBackup[]
        endpar
      endif
    endpar
  endif ...
rule r_runPCVExp =  ...
  if expired(timerExpirationDurPCV) then
    if cmdExPause then
      r_exPause[]
    else
      r_PCVStartInsp[]
    endif
  endif  ...
```

Code 5: MVMController02 PCV

```
rule r_runPSV =
  par ...
    if phase = INPAUSE then r_runInPause[] endif
    if phase = RM then r_runRm[] endif
    if phase = EXPAUSE then r_runExPause[] endif
  endpar
rule r_runPSVInsp =  ...
  if (expired(timerMinInspTimePSV) and flowDropPSV)
    or expired(timerMaxInspTimePSV) then
  if cmdInPause then r_InPause[]
  else if cmdRm then r_rm[]
  else r_PSVStartExp[] endif endif
  endif ...
rule r_runPSVExp =...
  if expired(timerApneaLag) then r_runApnea[]
  else  if expired(timerMinExpTimePSV) then
    par
      if respirationMode = PCV then
        par
          state := PCV_STATE
          r_PCVStartInsp[]
        endpar
      endif
      if respirationMode = PSV then
        if cmdExPause then r_ExPause[]  endif
      endif
    endpar
  endif endif ...
rule r_runApnea =
  par
    state := PCV_STATE
    r_PCVStartInsp[]
    apneaBackupMode := true
  endpar
```

Code 6: MVMController02 PSV

Model validation and verification. Model validation activities have been performed also at this level. Considering the properties verified in the previous refinement step, the property that states "valves are never closed at the same time" does not hold anymore. Indeed, when the ventilator is in pause the valves are both closed as guaranteed by this property:

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

To ensure that the valves are never both closed outside inspiratory and expiratory pause, we have verified the following property, in which valves are both closed if the ventilator is not in inspiration, expiration, recruitment maneuver, ventilation off, startup, or selftest.

LTLSPEC g((iValve=CLOSED and oValve=CLOSED) implies ((not ((phase=INSPIRATION or
phase=EXPIRATION or phase=RM) and (state = PCV_STATE or state = PSV_STATE)))) or
(not (state = VENTILATIONOFF or state = STARTUP or state = SELFTEST)))

4.4 Fourth model: MVMController03

In the last model, we have introduced the transition from inspiration to expiration and vice versa depending on the pressure changes due to spontaneous breathing. The new behavior has been modeled by

```
1  rule r_runPCVInsp =
2  ...
3    if expired(timerInspirationDurPCV) then
4      ...
5    else if pawGTMaxPinsp then
6      r_PCVStartExp[]
7    endif endif
8  rule r_runPCVExp =
9  ...
10   if expired(timerExpirationDurPCV) then
11     ...
12   else if expired(timerTriggerWindowDelay) and dropPAW_ITS then
13     r_PCVStartInsp[]
14   endif endif ...
```

Code 7: MVMController03 PCV

```
1  rule r_runPSVInsp =
2  ...
3    if (expired(timerMinInspTimePSV) and flowDropPSV) or
4     expired(timerMaxInspTimePSV) then
5      ...
6    else if pawGTMaxPinsp then
7      r_PCVStartExp[]
8    endif endif
9  rule r_runPSVExp =
10 ...
11   if expired(timerTriggerWindowDelay) and dropPAW_ITS then
12     r_PSVStartInsp[]
13   else if expired(timerApneaLag) then
14     ...
```

Code 8: MVMController03 PSV

extending the rules r_runPCVInsp and r_runPCVExp as shown in Code 7, and rules r_runPSVInsp and r_runPSVExp as shown in Code 8.

When the ventilator is in expiration (Code 7 line 8 and Code 8 line 9) and it detects after an instant of time (a *trigger window* here modeled as the expiration of the timer timerTriggerWindowDelay) a sudden drop in pressure below the inhale trigger sensitivity threshold (monitored function dropPAW_ITS holds, Code 7 line 13 and Code 8 line 11), the ventilator directly moves to the inspiration phase.

The transition from inspiration to expiration is automatically performed when the pressure goes beyond the maximum threshold set by the doctor (monitored function pawGTMaxPinsp holds, Code 7 line 5 and Code 8 line 6). At this third refinement level, we have performed the validation and verification activities as done for the previous levels.

5 C++ automatic code generation and unit testing

After having modeled the mechanical ventilator controller with AsmetaL and verified the specification, we have automatically generated the C++ code using the Asm2C++ tool starting from the last model, MVMController03.asm. The tool generates two different files, MVMController03.h and MVMController03.cpp, which contain the translation of the ASM model as a C++ class. During the C++ code generation, each ASM rule is translated into a class method. An example of the content of the MVMController03.cpp file is reported in Code 9, which contains the C++ translation of the two rules previously shown in Code 8.

Besides the source code implementing the ASM in C++, we have automatically generated the unit tests in C++ for that code. The automatic test generation is performed by the ATGT tool, which can exploit both the counterexamples given by the model checker and random traces generated by the ASMETA simulator. ATGT produces abstract tests as sequences of ASM states. These tests are then concretized using the Catch2 framework[3] and used for performing unit testing on the C++ code (see Code 10). Each test contains the declaration of the object under test (the mvmcontroller03 in our case). Then, the following steps are repeated: (I) the monitored functions are set (such as startupEnded in Code 10), (II) the possible controlled functions, which need to be initialized to the same value of the monitored ones, are set using the initControlledWithMonitored() method, (III) the main rule r_Main() is executed, (IV) the update set is fired using the fireUpdateSet() method, (V) the values of controlled functions are

[3] https://github.com/catchorg/Catch2

```
[...]
void MVMController03::r_runPCVInsp(){
  if (!stopVentilation[0]){ ... }
  if (expired(timerInspirationDurPCV)){
    if ((respirationMode == PCV)){
      if (cmdInPause){
        r_InPause();
      } else if (cmdRm){
        r_rm();
      } else {
        r_PCVStartExp();
      }
    }
  } else if (pawGTMaxPinsp)
    r_PCVStartExp();
}
void MVMController03::r_runPCVExp(){
  if (stopVentilation[0]){
    r_stopVent();
  } else if (stopRequested){
    r_stopVent();
  } else if (expired(timerExpirationDurPCV)){
    if (cmdExPause){
      r_exPause();
    }else{
      r_PCVStartInsp();
    }
  } else if (expired(timerTriggerWindowDelay) & dropPAW_ITS){
    r_PCVStartInsp();
  }
}
[...]
```

Code 9: Example of the MVMController03.cpp file

```
[...]
TEST_CASE( ”my_test_0”, ”my_test_0”){
  // instance of the SUT
    MVMController03 mvmcontroller03;
  // init controlled with monitored term
    mvmcontroller03.initControlledWithMonitored();
  // check controlled variables
    REQUIRE(mvmcontroller03.state[0] == STARTUP);
  // set monitored variables
    mvmcontroller03.startupEnded = true;
    mvmcontroller03.mCurrTimeSecs=1;
// call main rule
  mvmcontroller03.r_Main();
  mvmcontroller03.fireUpdateSet();
// check controlled variables
  REQUIRE(mvmcontroller03.state[0] == SELFTEST);
// set monitored variables
  mvmcontroller03.mCurrTimeSecs=2;
// call main rule
  mvmcontroller03.r_Main();
  mvmcontroller03.fireUpdateSet();
[...] }
```

Code 10: Example of a Catch2 test case

checked against the expected ones (for instance, state in Code 10). For the MVM controller, the testing process requires the simulation of the time as well: the time is represented as a monitored function whose value is incremented by one second at each step. As shown in Code 10, *mCurrTimeSecs* takes value initially 1 and is then incremented by 1 at each step.

In this case study, we have used the random test generation based on the use of the simulator. The user can configure the test generation process by choosing the number of tests to be generated and the number of steps each. We have generated 50 tests of 50 steps each and we have verified the compliance of the C++ code with the MVMController03.asm model. We have measured the code coverage and found that with the automatically generated tests we have covered 100% of code in terms of instructions of the MVMController03.cpp and MVMController03.h classes generated from the model. We have investigated why we have been able to completely cover the generated code, and found that the tests, although they are randomly generated, cover every possible ASM transition and the tool Asm2C++ does not generate dead code.

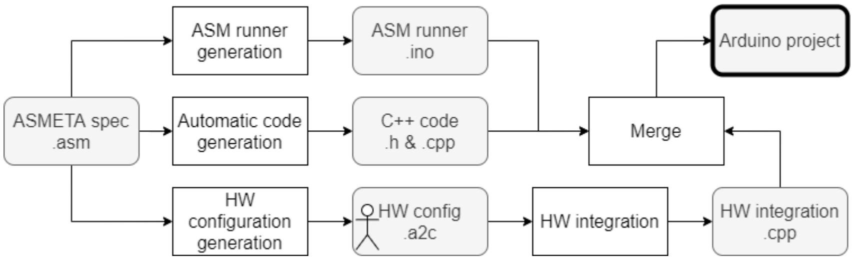


Figure 4: Arduino code generation process

```
{
”arduinoVersion”: ”UNO”,
”stepTime”: 0,
”bindings”: [
  {
    ”mode”: ”DIGITALOUT”,
    ”function”: ”iValve”,
    ”pin”: ”D8”
  },
  {
    ”mode”: ”DIGITALOUT”,
    ”function”: ”oValve”,
    ”pin”: ”D7”
  },{
    ”mode”: ”DIGITALIN”,
    ”function”: ”startupEnded”,
    ”pin”: ”A5”
  },
  {
    ”mode”: ”DIGITALIN”,
    ”function”: ”selfTestPassed”,
    ”pin”: ”A4”
  },

  [...]
}
```

Code 11: Example of the a2c configuration file

6 Code deployment on Arduino

Exploiting the `Asm2C++` tool, ASMETA allows the deployment of the C++ code as an Arduino sketch too [15]. The entire process is depicted in Fig. 4. After the generation of the `.h` and `.cpp` files as explained in Sect. 5, `Asm2C++` automatically generates an `.a2c` file. This configuration file is used for binding each ASM function to an Arduino physical pin. It must be completed by the user who has to insert the correspondence between Arduino physical pins and functions defined in the ASM model. For example, in Code 11, input and output valves (iValve and oValve) are mapped on digital output pins, while the monitored functions are used to set if the current phase is finished or not (e.g. startupEnded and selfTestPassed) are read using digital input pins.

Having completed the `.a2c` file with the correct mappings, `Asm2C++` generates two additional files: the `hw.cpp` and the `.ino`. The former (see Code 12) implements the hardware-specific methods, i.e. those related to the reading of inputs (getInputs() method in Code 12) and writing of outputs (setOutputs() method in Code 12) through physical pins. The latter (see Code 13), contains the execution policy to run the ASM on Arduino and performs cyclically the following operations: (i) getInputs() reads the inputs through digital and analog pins, (ii) r_Main(), represents the main rule of the ASM and executes all the rules, (iii) setOutputs(), sends the output values through the physical Arduino pins to the output components, (iv) fireUpdateSet(), updates the values of controlled functions to be used in the next state.

When modeling time we have exploited the TimeLibrary for time management. However, Arduino has very limited support for time and timers. We decided to use the `millis` instruction to implement temporized operations. `millis` returns the number of milliseconds passed since the Arduino board began running the current program and we have translated all the operations over the timers in terms of this function. We have chosen the hardware to be used in our Arduino-based implementation of the simplified MVM as follows (see Fig. 5):

- Arduino Uno, that executes the state machine;
- 3 LEDs used to communicate the status of input and output valves and the apnea alarm; one 1602 LCD display, which shows the current state;

```
#include ”MVMController03.h”

void MVMController03::getInputs(){
  startupEnded = (digitalRead(A5) == HIGH);
  selfTestPassed = (digitalRead(A4) == HIGH);
  [...]
}
void MVMController03::setOutputs(){
  if (iValve[0] != iValve[1]){
    if(iValve == OPEN)
      digitalWrite(8, LOW);
    else
      digitalWrite(8, HIGH);
  }
  if (oValve[0] != oValve[1]){
    if(oValve == OPEN)
      digitalWrite(7, LOW);
    else
      digitalWrite(7, HIGH);
  }
  [...]
}
```

Code 12: Extract of the `hw.cpp` file containing hardware-specific functions

```
#include”MVMController03.h”

void setup(){
  pinMode(8, OUTPUT);
  // ... set all the other outputs
  pinMode(A2, INPUT);
  // ... set all the other inputs
}
MVMController03 mvmcontroller03;

void loop(){
  mvmcontroller03.getInputs();
  mvmcontroller03.r_Main();
  mvmcontroller03.setOutputs();
  mvmcontroller03.fireUpdateSet();
}
```

Code 13: Example of the `.ino` file containing the implementation of the ASM execution

- 9 buttons, which simulates all the monitored functions contained in the ASM, namely dropPAW_ITS, pawGTMaxPinsp, cmdRm, cmdInPause, cmdExPause, flowDropPSV, respirationMode, stopRequested, startupEnded, selfTestPassed, and startVentilation. They represent both user inputs and external breathing events.

One can play with the MVM on Arduino by pressing the buttons and observing the status of the MVM. However, as it is, it provides a very limited user interface. To enable a more meaningful user experience, we have developed a custom-made Java *breathing simulator*. It is based on a simple lung model [19] whose electrical equivalent circuit is shown in Fig. 7. The breathing simulator allows the simulation of different patients, by setting the right values for resistance and compliance of their lungs, it can provide the measure of the pressure and flow of air in the patient lungs, and moreover, it can simulate events related to the patient’s breathing like a drop in the pressure when the patient starts. We connected the software simulator to the MVM on the Arduino using a serial port - so the MVM can communicate the status of the valves and therefore set the pressure of the ventilator and can read breathing events. An example of the breathing simulation is shown in Fig. 6.

We have tested the Arduino version of the MVM and a video, showing the functioning in all the ventilation modes, can be found at the following link `https://youtu.be/a3fhqLpYVMI`.

7 Discussion and Conclusions

In this section, we try to address the concerns and answer the questions proposed by the organizers. We believe that applying ASMETA to a real case study, that has been successfully completed with our direct involvement but without the use of formal methods, can put us in the privileged position to better understand the strength and weaknesses points of ASMETA and what is still needed to make it applicable in the future. Regarding *applicability*, ASMETA has never been applied in industrial projects, but we have used it in industrial case studies provided in several contexts like in the medical sector [6, 13], automotive [8], and avionics [10] to name a few. Barriers to the wide adoption of a formal method do

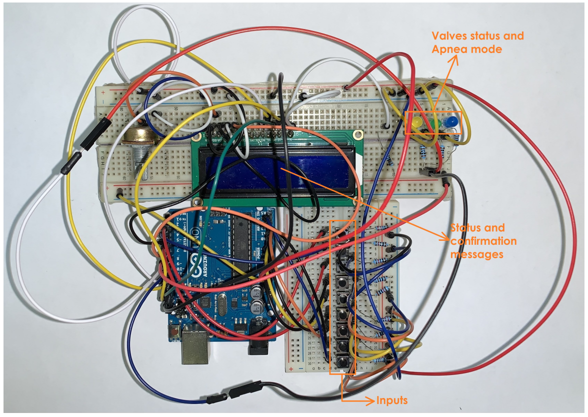

Valves status and Apnea mode
Status and confirmation messages
Inputs

Figure 5: The Arduino version of the MVM

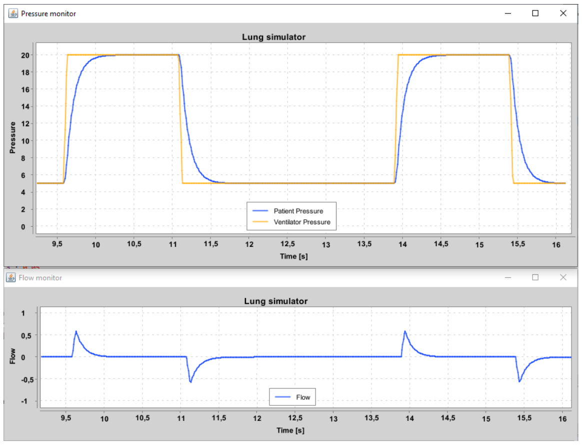


Figure 6: Example of the breathing simulation

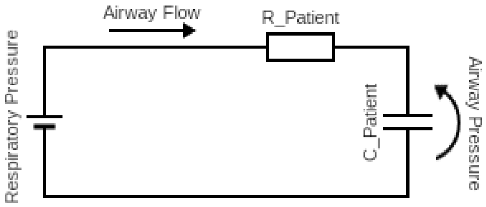


Figure 7: RC equivalent circuit of a human lung

exist [20, 21] and thanks to this work we have a better vision of what we need to do in order to favor the use of ASMETA in an industrial setting.

We have always given great importance to *automation* and our method is well supported by a set of tools (as explained in Sect. 2). In this case study, we did not suffer from the lack of a specific tool supporting a task. However, as explained below, our tools have different levels of maturity and sometimes the tool support has not been exhaustive.

For this work, *integration* among tools and techniques has played an important role. We think that having a model on which several techniques can be applied, including V&V, testing, and code generation is crucial for the use in the context of critical systems. Some integration among ASMETA tools is still work-in-progress. For instance, the TimeLibrary is not directly supported by the test generator tool and the support for the model checker NuXmv is rather new and it has some problems.

Another aspect to be considered is the integration of a formal approach into the engineering process. For MVM, as for some critical software lately [23], the software development process has been agile, with frequent changes in requirements that are then implemented in the code in a continuous integration way. ASMETA, as any Model-Driven Engineering approach, struggles in the integration with agile processes. For instance, direct modifications of the generated source code risk being lost if a change in the specification is performed and a new version of the code is generated.

For complex systems, modeling their entire behavior in one shot may be difficult. Thus, with the ASMETA *methodology*, users can incrementally add details in models using the refinement technique (see Sect. 4) or including already existing modules. Refinement and modularity are exploitable also when defining scenarios for model validation.

We envision the application of a formal process only to core components of the entire system. In the case of the MVM ventilator, for example, only the controller has been modeled by using ASMETA and the code obtained automatically from the specifications. Although for other MVM components, like the alarm system, a formal methodology could be a good fit, for other parts, such as graphical widgets or sensor drivers, formal specifications are possible but sometimes useless or even not suitable, especially if the source code is already provided or it can be reused (for instance for sensor drivers). In any case, the methodology must foresee a phase in which the whole code (the one obtained through formal specification and the one handwritten) is integrated (and tested) for building the complete system. In terms of the hardware platform, Arduino is a good choice when it comes to developing prototypes. However, for the real medical device, it should be replaced with a more robust one (MVM actually embeds a microcontroller that is Arduino code compatible).

One undoubted advantage regarding the *usefulness* of using a formal approach is that the certification process, if required, is easier to complete. The main standard for medical software certification to be considered is the IEC 62304 [1]. It does not prescribe a specific life cycle model but defines the process, activities, and tasks that the life cycle model has to follow. In [6], we have identified how ASMs can be used to satisfy the process and how the activities prescribed by the standard can be mapped to activities performed by using our formal approach. After requirements are captured by models, the verification and validation activities are straightforward, and the desired system behavior is automatically transferred to the generated code. Moreover, the formal process allows for requirements traceability which is required by the certification. Wrt the MVM agile process used for deploying the device and that required a huge effort to provide and guarantee requirements traceability, here, the use of ASMETA formal approach makes traceability easier to demonstrate (e.g. by mapping requirements to rules). When it comes to the usefulness in terms of the rapidness of code development, with the ASMETA framework and following the process described in this paper, we have been able to obtain the final code of the MVM controller in only a couple of days. However, one must consider that when modeling with ASMETA we already knew requirements, having contributed to the device development.

One activity whose *usefulness* is sometimes questioned is the unit test generation. After all, if the code is automatically generated from models, is there any need to validate it with tests derived from the same specification? We believe that there are at least three motivations: (i) the source code could be modified and the tests can check that the behavior as specified by the ASM is preserved; (ii) the translation from ASM to C++ itself must be validated, so unit tests check the conformance between the ASM and the C++ code; (iii) for certification purposes, source code must come with a test suite that provides confidence in its correctness.

Over the years, we have strived to improve ASMETA *usability* [7] by developing it as a set of Eclipse plugins, and feedbacks from our students, as well as the application of the method to competitive case studies, have been the basics for further improvements and extensions. However, it remains an academic tool whose stability and maintenance cannot reach the level often required in an industrial setting (and this was one of the reasons why we decided not to use it during the real MVM development). Moreover, the improvements introduced in the years have made ASMETA similar to higher-level programming languages, as for other formal methods. Nevertheless, the pseudo-code style and freedom of abstraction in ASMs allow for capturing of requirements at a very high-level of abstraction in a form that is understandable by the stakeholders.

In conclusion, we have applied ASMETA to assemble a prototype of a system which we have contributed to develop and deploy in the recent past and we have been able to evaluate the feasibility of our formal process. Although we believe that there is still some work to be done in order to provide the necessary stability and maturity of the tools and of the process, we were able to carry on the development of the MVM controller from the requirement specification to the code.

References

[1] *IEC 62304 Medical device software — Software life cycle processes.*

[2] A. Abba et al. (2021): *The novel Mechanical Ventilator Milano for the COVID-19 pandemic.* Physics of Fluids 33(3), p. 037122.

[3] Jean-Raymond Abrial (2006): *Formal Methods in Industry: Achievements, Problems, Future*, p. 761–768. Association for Computing Machinery, New York, NY, USA.

[4] P. Arcaini, S. Bonfanti, A. Gargantini & E. Riccobene (2016): *Visual Notation and Patterns for Abstract State Machines.* In Paolo Milazzo, Dániel Varró & Manuel Wimmer, editors: Software Technologies: Applications and Foundations: STAF 2016 Collocated Workshops: DataMod, GCM, HOFM, MELO, SEMS, VeryComp, Vienna Austria, July 4-8, 2016, LNCS, Springer International Publishing, pp. 163–178.

[5] Paolo Arcaini, Andrea Bombarda, Silvia Bonfanti, Angelo Gargantini, Elvinia Riccobene & Patrizia Scandurra (2021): *The ASMETA Approach to Safety Assurance of Software Systems.* In Alexander Raschke, Elvinia Riccobene & Klaus-Dieter Schewe, editors: Logic, Computation and Rigorous Methods: Essays Dedicated to Egon Börger on the Occasion of His 75th Birthday, Springer International Publishing, Cham, pp. 215–238.

[6] Paolo Arcaini, Silvia Bonfanti, Angelo Gargantini, Atif Mashkoor & Elvinia Riccobene (2018): *Integrating formal methods into medical software development: The ASM approach.* Science of Computer Programming 158, pp. 148–167.

[7] Paolo Arcaini, Silvia Bonfanti, Angelo Gargantini, Elvinia Riccobene & Patrizia Scandurra (2020): *Addressing Usability in a Formal Development Environment.* In: AFFORD 2019 - Workshop on Practical Formal Verification for Software Dependability - workshop of FM 19, Springer International Publishing, pp. 61–76.

[8] Paolo Arcaini, Silvia Bonfanti, Angelo Gargantini, Elvinia Riccobene & Patrizia Scandurra (2020): *Modelling an Automotive Software-Intensive System with Adaptive Features Using ASMETA.* In Alexander Raschke, Dominique Méry & Frank Houdek, editors: Rigorous State-Based Methods, Springer International Publishing, Cham, pp. 302–317.

[9] Paolo Arcaini, Angelo Gargantini & Elvinia Riccobene (2016): *SMT-based automatic proof of ASM model refinement.* In Rocco De Nicola & Eva Kühn, editors: Software Engineering and Formal Methods: 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings, Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 253–269.

[10] Paolo Arcaini, Angelo Gargantini & Elvinia Riccobene (2017): *Rigorous development process of a safety-critical system: from ASM models to Java code.* International Journal on Software Tools for Technology Transfer 19(2), pp. 247–269.

[11] Paolo Arcaini, Angelo Gargantini, Elvinia Riccobene & Patrizia Scandurra (2011): *A model-driven process for engineering a toolset for a formal method.* Software: Practice and Experience 41, pp. 155–166.

[12] *ASMETA (ASM mETAmodeling) toolset.* `https://asmeta.github.io/`.

[13] Andrea Bombarda, Silvia Bonfanti & Angelo Gargantini (2019): *Developing Medical Devices from Abstract State Machines to Embedded Systems: A Smart Pill Box Case Study.* In: Software Technology: Methods and Tools, Springer International Publishing, Cham, pp. 89–103.

[14] Andrea Bombarda, Silvia Bonfanti, Angelo Gargantini & Elvinia Riccobene (2021): *Extending ASMETA with Time Features.* In: Rigorous State-Based Methods, Springer International Publishing, pp. 105–111.

[15] Silvia Bonfanti, Marco Carissoni, Angelo Gargantini & Atif Mashkoor (2017): *Asm2C++: A Tool for Code Generation from Abstract State Machines to Arduino.* In: Lecture Notes in Computer Science, Springer International Publishing, pp. 295–301.

[16] Silvia Bonfanti, Angelo Gargantini & Atif Mashkoor (2019): *Design and validation of a C++ code generator from Abstract State Machines specifications.* Journal of Software: Evolution and Process 32(2).

[17] Egon Börger (2003): *Abstract State Machines : a Method for High-Level System Design and Analysis.* Springer Berlin Heidelberg, Berlin, Heidelberg.

[18] Egon Börger & Alexander Raschke (2018): *Modeling Companion for Software Practitioners*. Springer Berlin Heidelberg.

[19] D. Campbell & J. Brown (1963): *THE ELECTRICAL ANALOGUE OF LUNG*. *BJA: British Journal of Anaesthesia* 35, pp. 684–692.

[20] Hubert Garavel, Maurice H. ter Beek & Jaco van de Pol (2020): *The 2020 Expert Survey on Formal Methods*. In Maurice H. ter Beek & Dejan Ničković, editors: *Formal Methods for Industrial Critical Systems*, Springer International Publishing, Cham, pp. 3–69.

[21] Mario Gleirscher & Diego Marmsoler (2020): *Formal methods in dependable systems engineering: a survey of professionals from Europe and North America*. *Empirical Software Engineering* 25(6), pp. 4473–4546.

[22] Maria Chiara Di Guardo, Elona Marku, Walter Marcello Bonivento, Manuel Castriotta, Fernando Ferroni, Cristiano Galbiati, Giuseppe Gorini & Michela Loi (2021): *When nothing is certain, anything is possible: open innovation and lean approach at MVM*. *R&D Management*.

[23] Gibrail Islam & Tim Storer (2020): *A case study of agile software development for safety-Critical systems projects*. *Reliability Engineering & System Safety* 200, p. 106954.

[24] Nancy Leveson (2020): *Are You Sure Your Software Will Not Kill Anyone?* *Commun. ACM* 63(2), pp. 25–28.

[25] Robyn R. Lutz (2000): *Software Engineering for Safety: A Roadmap*. In: *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, Association for Computing Machinery, New York, NY, USA, pp. 213–226.

[26] Rodney N Westhorpe & C Ball (2012): *The Manley Ventilator*. *Anaesthesia and intensive care* 40(5), pp. 749–750.