How to assure correctness and safety of medical software: the Hemodialysis Machine Case Study*

Paolo Arcaini[1], Silvia Bonfanti[2], Angelo Gargantini[2], and Elvinia Riccobene[3]

[1] Charles University in Prague, Faculty of Mathematics and Physics, Czech Republic
arcaini@d3s.mff.cuni.cz
[2] Department of Economics and Technology Management, Information Technology and Production, Università degli Studi di Bergamo, Italy
{silvia.bonfanti,angelo.gargantini}@unibg.it
[3] Dipartimento di Informatica, Università degli Studi di Milano, Italy
elvinia.riccobene@unimi.it

Abstract. Medical devices are nowadays more and more software dependent, and software malfunctioning can lead to injuries or death for patients. Several standards have been proposed for the development and the validation of medical devices, but they establish general guidelines on the use of common software engineering activities without any indication regarding methods and techniques to assure safety and reliability.
This paper takes advantage of the Hemodialysis machine case study to present a formal development process supporting most of the engineering activities required by the standards, and provides rigorous approaches for system validation and verification. The process is based on the Abstract State Machine formal method and its model refinement principle.

1 Introduction

In medical treatments depending on the use of a medical device (e.g., a hemodialysis machine), patient safety depends upon the correct operation of the device hardware/software. For this reason, validation and verification of medical devices are mandatory, and methods and techniques to assure medical software safety and reliability are highly demanded. Along this research line, the Hemodialysis machine case study [20] (HMCS) has been proposed, within the ABZ 2016 conference, as an example of medical device that should be formally validated and verified to assure safety and hardware-software correct interaction.

Although several standards for the validation of medical devices have been proposed – as ISO 13485 [1], ISO 14971 [4], IEC 60601-1 [2], EU Directive 2007/47/EC [16] –, they mainly consider physical aspects and electrical components of a device rather than its software. The main references concerning

* The research reported in this paper has been partly supported by the Grant Agency of the Czech Republic project 14-11384S, and by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the Province of Upper Austria in the frame of the COMET center SCCH.

regulation of medical software development are the standard IEC 62304 [3] (International Electrotechnical Commission) and the "General Principles of Software Validation" [21] established by the FDA (Food and Drug Administration). Both documents establish general guidelines on the use of common software engineering activities, but they do not provide any indication regarding life cycle models, or methods and techniques to assure safety and reliability. These qualities could be assured by adopting rigorous approaches of software development, based on the use of formal methods, that allow the designers to specify what the software is intended to do by means of mathematical models, and demonstrate that the use of the software fulfills those intentions by means of precise validation and verification techniques [3,19].

Among the different existing formal methods, Abstract State Machines [13] have been already successfully used in the context of medical software for the rigorous development of an optometric measurement device software [5].

The ASM-based design process [10] has an incremental life cycle model. It is based on model refinement, includes the main software engineering activities, and is supported by techniques for model validation and verification at any desired level of detail. The process can guide the development of software and embedded systems seamlessly from requirements capture to their implementation, and this has been shown by numerous and successful case studies [13]. Although the ASM method has a rigorous mathematical foundation, practitioners need no special training to use the method since ASMs can be correctly understood as pseudocode or virtual machines working over abstract data structures.

In this paper, we apply the ASM-based process for modeling, validating, and verifying the HMCS. At the same time, we present a software development process compliant with two current standards for medical software. It provides a life cycle model for IEC62304 and embeds most of the software engineering activities required by this standard. Moreover, it reflects the principles established by the FDA, especially those regarding the integration of validation and verification activities into the development process.

Sect. 2 briefly introduces the ASM-based development process. Sect. 3 presents the specification of the HMCS given in terms of a chain of refined models. Sect. 4 presents some validation activities we have applied to the developed models, and Sect. 5 reports the verification of the requirements. Sect. 6 recalls the normative for medical software and shows how the ASM-based development process captures the existing regulations. Sect. 7 concludes the paper.

As at the moment of writing this paper no other solutions for the HMCS are available, we do not report any related work. For an example of application of different formal methods to a safety-critical system, we remind the reader to the solutions proposed for the Landing Gear System case study [12].

2 ASM-based development process

ASMs allow an iterative design process, shown in Fig. 1, based on model refinement. Validation and verification (V&V) are fully integrated into the process,

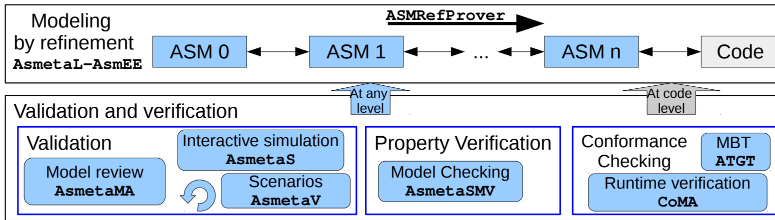

Fig. 1. ASM-based development process

and are possible at any level of abstraction. Tools supporting the process are part of the ASMETA (ASM mETAmodeling) framework[4] [11].

Modeling requirements starts by developing a high-level model called *ground model* (ASM 0 in Fig. 1). It is specified by reasoning on the informal requirements (generally given as a text in natural language) and using terms of the application domain, possibly with the involvement of all stakeholders. The ground model should *correctly* reflect the intended requirements and should be *consistent*, i.e., without possible ambiguities of initial requirements. It does not need to be *complete*, i.e., it may not specify some given requirements. The ground model and the other ASM models can be edited in `AsmEE` by using the concrete syntax `AsmetaL` [18].

Starting from the ground model, through a sequence of *refined* models, further functional requirements can be specified and a complete architecture of the system can be given. The refinement process permits to tackle the complexity of the system, and allows to bridge, in a seamless manner, specification to code.

Each refinement step should be proved to be correct: the refinement correctness proof can be done by hand or, for a particular kind of refinement called *stuttering refinement*, using the tool `ASMRefProver`.

At each level of refinement, already at the level of the ground model, different *validation* and *verification* (V&V) activities can be applied.

Model validation helps to ensure that the specification really reflects the intended requirements, and to detect faults and inconsistencies as early as possible with limited effort. ASM model validation is possible by means of the model simulator `AsmetaS` [18] and by the validator `AsmetaV` [15] that allows to build and execute *scenarios* of expected system behaviors. A further validation technique is *model review* (a form of static analysis) to determine if a model has sufficient *quality* attributes (as minimality, completeness, consistency). Automatic ASM model review is possible by means of the `AsmetaMA` tool [7].

Model verification requires the use of more expensive and accurate methods. Formal verification of ASMs is possible by means of the model checker `AsmetaSMV` [6], and both *Computation Tree Logic* (CTL) and *Linear Temporal Logic* (LTL) formulas are supported.

[4] `http://asmeta.sourceforge.net/`

Ground model	1st refinement	2nd refinement	3rd refinement	4th refinement
Machine phases	Preparation Phase	Initiation Phase	Ending Phase	Checks and errors
size: 0 m, 1 c, 0 d, 5 rd, 11 r, 0 p	*size*: 15 m, 9 c, 2 d, 58 rd, 179 r, 0 p	*size*: 32 m, 22 c, 3 d, 89 rd, 303 r, 2 p	*size*: 39 m, 25 c, 3 d, 103 rd, 363 r, 3 p	*size*: 60 m, 39 c, 12 d, 154 rd, 608 r, 41 p

Fig. 2. Refinement steps - *size*: m=#monitored functions, c= #controlled, d=#derived, rd =#rule declarations, r = #rules, p = #properties

If a system implementation is available, either derived from the model (as last low-level refinement step) or externally provided, also *conformance checking* is possible. Both *model-based testing* (MBT) and *runtime verification* can be applied to check if the implementation conforms to its specification [9]. We support conformance checking w.r.t. Java code. The tool `ATGT` [17] can be used to automatically generate tests from ASM models[5] and, therefore, to check the conformance *offline*; `CoMA` [8], instead, can be used to perform runtime verification, i.e., to check the conformance *online*.

3 Hemodialysis device: modeling

In modeling the hemodialysis device[6] we have proceeded through refinement. The complete model has been developed in five levels, as shown in Fig. 2 where some data of the five models are reported. Each refined model introduces, w.r.t. the previous model, some additional details to the machine behaviour. The ground model abstractly describes the transitions between the three hemodialysis phases: preparation, initiation, and ending. In the first three refinement steps, we refine the three phases singularly. In the last refinement step, to comply with the given requirements, we introduce the modeling of the checks and the error handling performed by the device. For each step of refinement, we prove the refinement correctness using the SMT-based tool `AsmRefProver`.

3.1 Ground model

The ground model simply describes the transition between the phases that constitute a hemodialysis treatment, without any additional detail. Code 1 shows the ground model written using the `AsmetaL` syntax. The machine, at each step, checks the current `phase` of the treatment and executes the corresponding rule. Rule `r_preparation` performs all the activities necessary to prepare the hemodialysis device for the treatment (modeled in Sect. 3.2). Rule `r_initiation` specifies the hemodialysis therapy in which the patient is connected to the device and the blood cleaning is performed (modeled in Sect. 3.3). Rule `r_ending` models the ending of the therapy, in which the patient is disconnected from the device and the device is cleaned for subsequent treatments (modeled in Sect. 3.4).

[5] Note that sequences generated by `ATGT` could be used to test programs written in any programming language.
[6] Models are available at `http://fmse.di.unimi.it/sw/ABZ2016caseStudy.zip`

asm Hemodialysis_GM signature: enum domain Phases = {PREPARATION | INITIATION | ENDING} controlled phase: Phases definitions: rule r_preparation = phase := INITIATION rule r_initiation = phase := ENDING rule r_ending = skip	rule r_run_dialysis = switch(phase) case PREPARATION: r_preparation[] case INITIATION: r_initiation[] case ENDING: r_ending[] endswitch main rule r_Main = r_run_dialysis[] default init s0: function phase = PREPARATION

Code 1. Ground model

3.2 First refinement: preparation phase

The first refinement extends the ground model by refining the preparation phase specified in rule `r_preparation` (see Code 2). In this phase, different activities are performed to prepare the device for the therapy, as the preparation of the heparin or the rinsing of the dialyzer. Function `modePreparation` specifies which activity (i.e., which rule) must be executed in the current state.

For the lack of space, we here only report the setting of the treatment parameters, that is modeled by rule `r_set_treatment_param`. The machine sets one parameter at a time, using function `treatmentParam` to keep track of the parameter that has to be updated. Each parameter is updated by rule `r_insert_param`, shown in Code 2. For each parameter, there exist one monitored function and one controlled function. The model takes the value of the monitored function decided by the environment (in this case, by the user), and, if the specified value is allowed, saves it in the controlled function. When a correct value is acquired, the machine updates function `treatmentParam` with the name of the next parameter that must be set.

3.3 Second refinement: initiation phase

This step refines the initiation phase (see Code 3), which is divided into two steps identified by function `initPhaseMode`: `r_initiate_patient` and `r_run_therapy`. During all the initiation phase, some checks on the device are performed by rule `r_check_initiation_phase`; such rule is left abstract in this refinement step, and it will be modeled in the fourth refinement (see Sect. 3.5).

`r_initiate_patient` is the first activity to perform. It is divided into two parallel actions, `r_check_patient` and `r_connect_patient`. Rule `r_check_patient`, that is responsible for doing some additional checks on the patient, is also left abstract in this refinement step (as rule `r_check_initiation_phase`), and it will be modeled in the fourth refinement (see Sect. 3.5). Rule `r_connect_patient` (shown in Code 4) implements the procedures described in section "Connecting the patient and starting therapy" of the case study document [20]; it executes

```
asm Hemodialysis_ref1
signature:
  enum domain ModePreparation = { ... |
  RINSE_DIALYZER | SET_TREAT_PARAM}
  enum domain TreatmentParam = { ... |
  BLOOD_CONDUCTIVITY | ACTIVATION_H}
  controlled modePreparation: ModePreparation
  controlled treatmentParam: TreatmentParam
  ...
definitions:

  rule r_insert_param($low_lim in Integer,
            $up_lim in Integer,
            $next_param in TreatmentParam,
            $mon_param in Integer,
            $contr_param in Integer) =
    if $mon_param <= $up_lim and
              $mon_param >= $low_lim then
      par
        $contr_param := $mon_param
        treatmentParam := $next_param
      endpar
    endif

  rule r_set_h_activation =
    r_insert_param[SYRINGE_TYPE,
              activation_h, activation_h_contr]

rule r_set_treatment_param =
  switch(treatmentParam)
    case BLOOD_CONDUCTIVITY:
      r_set_blood_conductivity[]
    case ACTIVATION_H:
      r_set_h_activation[]
    ...
  endswitch

rule r_preparation =
  switch(modePreparation)
    case AUTO_TEST:
      r_auto_test[]
    case CONNECT_CONCENTRATE:
      r_connect_concentrate[]
    case SET_RINSING_PARAM:
      r_set_rinsing_param[]
    case TUBING_SYSTEM:
      r_tubing_system[]
    case PREPARE_HEPARIN:
      r_prepare_heparin[]
    case SET_TREAT_PARAM:
      r_set_treatment_param[]
    case RINSE_DIALYZER:
      r_rinse_dialyzer[]
  endswitch
...
```

Code 2. First refinement

a series of activities related to the patient connection, specified by function `patientPhase`. At the beginning, the patient is connected arterially (by rule `r_conn_arterially`). Then, the blood pump is started (by rule `r_start_bp`), the blood flow is set (by rule `r_set_blood_flow`), and the blood tubing system is filled with blood (by rule `r_fill_tubing`). Then, the patient is connected venously (by rule `r_conn_venously`). Finally, the blood pump is started and the blood flow is set again; the connection procedure is terminated (by rule `r_end_connection`).

`r_run_therapy` is executed after `r_initiate_patient` and it is composed of a set of activities, as shown in Code 3. Function `therapyPhase` specifies which activity must be executed in a given moment. At the beginning, rule `r_start_heparin` activates the heparin, if required by the doctor. Then, the therapy is executed (rule `r_therapy_exec`) and terminated (rule `r_theraphy_end`).

Rule `r_therapy_exec` is shown in Code 4. A set of parallel activities are performed as specified in section "During therapy" of the case study document [20]: the monitoring of the blood pressure limits (rule `r_monitor_ap_vp_limits`), the

```
asm Hemodialysis_ref2

signature:
  enum domain ModeInitiation =
  {CONNECT_PATIENT | THERAPY_RUNNING}
  enum domain TherapyPhase =
  {START_HEPARIN | ... | THERAPY_END}
  controlled initPhaseMode: ModeInitiation
  controlled therapyPhase: TherapyPhase
  ...

definitions:

  rule r_initiate_patient =
    par
      r_check_patient[]
      r_connect_patient[]
    endpar
```

```
rule r_run_therapy =
  switch(therapyPhase)
    case START_HEPARIN:
      r_start_heparin[]
    case THERAPY_EXEC:
      r_therapy_exec[]
    case THERAPY_END:
      r_therapy_end[]
  endswitch

rule r_initiation =
  par
    r_check_initiation_phase[]
    switch(initPhaseMode)
      case CONNECT_PATIENT:
        r_initiate_patient[]
      case THERAPY_RUNNING:
        r_run_therapy[]
    endswitch
  endpar
```

Code 3. Second refinement

```
asm Hemodialysis_ref2
signature:
  enum domain PatientPhase =
  {CONN_ARTERIALLY | ... | END_CONN}
  controlled patientPhase: ModeInitiation
  ...
definitions:
rule r_connect_patient =
  switch(patientPhase)
    case CONN_ARTERIALLY: r_conn_arterially[]
    case START_BP: r_start_bp[]
    case BLOOD_FLOW: r_set_blood_flow[]
    case FILL_TUBING: r_fill_tubing[]
    case CONN_VENOUSLY: r_conn_venously[]
    case END_CONN: r_end_connection[]
  endswitch
```

```
rule r_therapy_exec =
  par
    r_pump_heparin[]
    r_monitor_ap_vp_limits[]
    r_therapy_min_UF[]
    r_update_blood_flow[]
    r_check_therapy_run[]
    r_arterial_bolus[]
    r_interrupt_dialysis[]
    r_therapy_completion[]
  endpar
```

Code 4. Second refinement – Rules `r_connect_patient` and `r_therapy_exec`

activation of the treatment at the minimum ultra filtration rate (rule `r_therapy_min_UF`), the update of the blood flow (rule `r_update_blood_flow`), the infusion of sodium chloride (rule `r_arterial_bolus`), the pumping of the heparin in the blood (rule `r_pump_heparin`). Rule `r_check_therapy_run` does some checks during the blood cleaning: it is left abstract at this level of refinement and

```
asm Hemodialysis_ref3                           definitions:
                                                  rule r_ending =
signature:                                          switch(endingPhaseMode)
  enum domain ModeEnding =                            case REINFUSION:
  {REINFUSION | ... | THERAPY_OVERVIEW}                 r_reinfusion[]
  controlled endingPhaseMode: ModeEnding              case DRAIN_DIALYZER:
  ...                                                   r_empty_dialyzer[]
                                                      case EMPTY_CARTRIDGE:
                                                        r_empty_cartridge[]
                                                      case THERAPY_OVERVIEW:
                                                        r_therapy_overview[]
                                                    endswitch
```

Code 5. Third refinement

it will be modeled in the fourth refinement (see Sect. 3.5). Activities `r_interrupt_dialysis` and `r_therapy_completion` can terminate the therapy (by updating `therapyPhase` to `THERAPY_END`): `r_interrupt_dialysis` models the premature interruption of the therapy, whereas `r_therapy_completion` models the normal therapy conclusion.

3.4 Third refinement: ending phase

The third refinement details the behaviour of the ending phase (see Code 5) consisting of a set of activities that are performed at the end of the treatment. Function `endingPhaseMode` specifies the current activity. At the beginning, the reinfusion of the blood takes place by means of rule `r_reinfusion`. Then, the dialyzer is emptied (by rule `r_empty_dialyzer`), and afterwards the cartridge is emptied (by rule `r_empty_cartridge`). At the end, rule `r_therapy_overview` gives an overview of the executed therapy (e.g., how much blood has been treated).

3.5 Fourth refinement: handling of checks and errors

In the last refinement step, we model the checking activities and the error handling performed by the device. During the whole hemodialysis process, the device checks some parameters acquired using sensors. Every parameter is checked and, when an error occurs, the system raises an alarm and an error signal. The alarms are reset just after the device operator presses the button. The errors, instead, are reset when the user fixes them.

For example, we here consider the error due to the high temperature of the dialysing fluid (see Code 6). The machines checks the temperature only if an error related to high temperature has not been raised yet (rule `r_check_temp_high` in Code 6). When the temperature exceeds the threshold value, an error and an alarm signal are generated. If the alarm is running, the device operator

```
asm Hemodialysis_ref4
signature:
  enum domain ErrorAlarmType =
  {TEMP_HIGH | UF_BYPASS |...}
  controlled error: ErrorAlarmType −> Boolean
  controlled alarm: ErrorAlarmType −> Boolean
  ...
definitions:
  rule r_check_temp_high =
    if not(error(TEMP_HIGH)) then
      if current_temp > 41 then
        par
          error(TEMP_HIGH) := true
          alarm(TEMP_HIGH) := true
        endpar
      endif
    endif
```

```
rule r_turnOff_alarm =
  if reset_alarm then
    par
      forall $a in ErrorAlarmType
                  with alarm($a) do
        alarm($a) := false
    ...
    endpar
  endif

rule r_error_temp_high =
  if error(TEMP_HIGH) and
    not(alarm(TEMP_HIGH)) then
      if current_temp <= 41 then
        error(TEMP_HIGH) := false
      endif
    endif
```

Code 6. Fourth refinement – Error of high temperature of the dialyzing fluid

can decide to reset it by setting `reset_alarm` function to `true` in rule `r_turnOff_alarm`. Once the alarm has been reset (but the error is still flagged) and the temperature has returned to an acceptable value, the model resets the error (rule `r_error_temp_high`).

4 Hemodialysis device: Validation

We here describe the validation activities we performed on the produced models.

Interactive simulation and scenario-based validation As first validation activity, we performed *interactive simulation* by means of the simulator `AsmetaS` [18] that allowed us to observe some particular system executions. Interactive simulation consists in providing inputs (i.e., values of monitored functions) to the machine and observing the computed state. The simulator, at each step, performs *consistent updates checking* to check that all the updates are consistent (two updates are inconsistent if they update the same location to two different values at the same time [13]), and *invariant checking*.

As interactive simulation is a tedious and time-consuming activity, after we gained enough confidence that the system roughly captured the intended requirements, we performed a more powerful form of simulation, called *scenario-based validation* [15], that permits to automatize the simulation activity. In scenario-based validation the designer provides a set of scenarios specifying the expected behaviour of the models (using the textual notation `Avalla` [15]). These scenarios are used for validation by instrumenting the simulator `AsmetaS`. During

```
scenario errorPressureTherapy
load Hemodialysis_ref4.asm

execblock completeTherapy.initiationCheck;
step
execblock completeTherapy.preparationPhase;
step
execblock completeTherapy.patientConn;
step
check therapyPhase = THERAPY_EXEC;
check heparin_running = true;

set passedSec(10) := true;
set current_ap := 100;
set vp_limit_low := 50;
set vp_limit_up := 150; ...
step
check therapyPhase = THERAPY_END;

step
execblock completeTherapy.therapyEnd;
```

Code 7. Example of scenario using blocks

```
scenario completeTherapy
load Hemodialysis_ref4.asm

begin initiationCheck
  check phase = PREPARATION and
  prepPhaseMode = AUTO_TEST and
  rinsingParam = FILLING_BP_RATE;
  ...
  set auto_test_end := true;
end

begin preparationPhase
  check prepPhaseMode =
        CONNECT_CONCENTRATE;
  set conn_concentrate := true;
  step
  ...
end
```

Code 8. Example of scenario blocks

simulation, `AsmetaV` captures any check violation and, if none occurs, it finishes with a *PASS* verdict. `Avalla` provides constructs to express execution scenarios in an algorithmic way, as interaction sequences consisting of actions committed by the user to `set` the environment (i.e., the values of monitored/shared functions), to `check` the machine state, to ask for the `exec`ution of certain transition rules, and to enforce the machine itself to make one `step` (or a sequence of steps by command `step until`) as reaction of the actor actions.

While developing our models, we wrote several scenarios for exercising the models under different operating conditions. Soon we discovered that such scenarios had several common parts, since they had to perform the same actions and same checks in different parts of their evolution. Therefore, we extended the validator with the possibility to define some *blocks of actions* that can be reused in different components: a block is a named sequence of `Avalla` commands delimited by keywords `begin` and `end`. A command block can be defined in any `Avalla` scenario and can be called by means of the command `execblock` in other parts of the same scenario or in other scenarios.

Code 7 shows an example of scenario (over the last refined machine) reproducing the situation in which an error is detected and resolved. The scenario calls some blocks (defined in `completeTherapy`) containing actions related to the therapy that are used in several different scenarios (e.g., the initial checks and the operations performed during the preparation phase). Code 8 shows the definition of some blocks.

Static analysis Although interactive simulation and scenario-based validation can discover many faults in the models, some of them may pass undetected. Moreover, even if the models correctly capture the requirements, they may still be improved from the stylistic point of view. We therefore applied *model review*, whose aim is to determine if a model has some particular *qualities* that should help in developing, maintaining, and enhancing it. The `AsmetaMA` tool [7] (based on `AsmetaSMV`) performs *automatic* review of ASMs. Common vulnerabilities and defects that can be introduced during ASM modeling are checked as violations of suitable *meta-properties* (*MPs*, defined in [7] as CTL formulae). The violation of a meta-property means that a quality attribute (*minimality*, *completeness*, *consistency*) is not guaranteed, and it may indicate the presence of an actual fault (i.e., the ASM is indeed faulty), or only of a *stylistic defect* (i.e., the ASM could be written in a better way). For example, the presence of an inconsistent update (meta-property MP1) is the sign of a real fault in the model; the presence of functions that are never read nor updated (meta-property MP7), instead, may simply indicate that the model is not minimal, but not that it is faulty.

In our last step of refinement, we found several violations of meta-property MP1, i.e., inconsistent updates. These were due to a wrong mechanism in the handling of errors. Different kinds of errors can be detected by the device (see Sect. 3.5), and, in all these cases, the state of the device is set to the `BYPASS` mode. In the faulty model, when a given error is resolved, the error handler puts the device back to the normal mode `MAIN_FLOW`; however, if another error is simultaneously detected, another error handler puts the device in the `BYPASS` mode: in this way, an inconsistent update occurs, since two rules try to update the same function to two different values. We have restructured the model so that error handlers only detect errors and mark them as resolved after the error handling (but do not modify the device state); the real mode of the device is now determined by a derived function, whose value is `BYPASS` if at least one not resolved error exists, otherwise its value is determined by the phase of the therapy. Note that we were not able to detect such a problem by means of simulation, since in simulations we considered errors occurring singularly or simultaneously, but we did not consider errors occurring while others are being resolved.

We also found some minimality violations related to some domain values that were not useful. For example, we discovered that value `RED` of domain `SignalLamps` was useless: indeed, in the manual of the device [14], we found that the lamp could be red in particular situations, but this case was not considered in the given requirements [20]. We may decide to remove the value from the domain or keep it for further improvements of the model.

5 Hemodialysis device: Verification of requirements

We verified the 11 general requirements (S-1 – S-11) and the 36 software requirements (R-1 – R-36) specified in the case study description [20]. We specified requirements as LTL formulas and we used the model checker `AsmetaSMV` [6] for their verification. A formula may capture more than one requirement.

As an example of requirement formalization, let us consider the general requirement S-11: "Once *empty dialyzer* has been confirmed, the blood pump cannot be started anymore". Such requirement has been specified as follows:

$\mathsf{g(empty_dialyzer\ implies\ g(bp_status_der = STOP))}$

Along the validation activity, each requirement has been proved as soon as possible in the chain of refinements, i.e., in the first model describing all the elements involved in the requirement. Requirement S-1, regarding the connection of arterial and venous connectors, has been added in the second refinement, that models the initiation phase in which the patient is physically connected to the device. Also requirement S-4, regarding the exchange between saline solution and blood during the connection of the patient, has been specified in the second refinement. Requirement S-11, regarding the stopping of the blood pump at the end of the therapy, has been added in the third refinement that models the ending phase. All the other general requirements and all the software requirements are related to errors and alarms, and, therefore, have been specified in the last refinement step that models error handling.

Although we specified all of the requirements, some of them are trivial since already captured by the semantics of the transition rules. In particular, all the properties (from R-1 to R-36) having form $\mathsf{G}(\varphi\ \mathsf{implies}\ \mathsf{X}(\rho))$, where φ and ρ are predicates over the ASM state, are directly captured by the model structure of nested conditional rules.

We discovered that some requirements specified in [20] are not correct. For example, we specified the following property for S-1:

$\mathsf{g(art_connected_contr\ iff\ ven_connected_contr)}$

checking that "arterial and venous connectors of the EBC are connected to the patient simultaneously" [20]. However, the property is false since the patient is *before* connected to the arterial connector and *then* to the venous connector.

We had problems in verifying some requirements, since these were ambiguous. Requirement S-5, for example, states that "the patient cannot be connected to the machine outside the initiation phase, e. g., during the preparation phase". We were not sure whether to interpret "be connected" as the status of the patient who is attached to the machine, or as the atomic action in which the doctor connects the patient to the machine. Following the former interpretation, the temporal property would be

$\mathsf{g((art_connected_contr\ or\ ven_connected_contr)\ implies\ phase = INITIATION)}$

However, the property is false since the patient can be attached to the machine also outside the `INITIATION`. Following the former interpretation, we wrote the two following properties

$$\mathsf{g((not\ art_connected_contr\ and\ x(art_connected_contr))\ implies}\\ \mathsf{phase = INITIATION)}$$
$$\mathsf{g((not\ ven_connected_contr\ and\ x(ven_connected_contr))\ implies}\\ \mathsf{phase = INITIATION)}$$

5.1 Software development planning	5.2 Software requirements analysis	5.3 Software architectural design	5.4 Software detailed design	5.5 Software unit implementation and verification	5.6 Software integration and integration testing	5.7 Software system testing	5.8 Software release

Fig. 3. IEC 62304 development process

that are both true. It could be the case that this interpretation is indeed the correct one; however, this is a clear example of an ambiguous requirement that would need a clarification from the stakeholders.

For keeping verification time reasonable, we applied some abstractions to our models. For example, in the preparation phase introduced in the first refinement (see Sect. 3.2), the model retrieves values for the parameters from the environment (by means of monitored functions), checks their validity (i.e., whether they are in allowed ranges), and stores them in some controlled functions. For model checking, we abstract from that mechanism: for each parameter, we simply have a monitored function saying whether its new value is allowed or not.

6 ASM process and Normatives for medical software

We want here to relate the ASM-based design process to the current normative for developing medical software. The aim is to evaluate how far we are from having a formal process compliant with the standards. The two main normative references for development and validation of medical software are the standard IEC 62304 [3] and the "General Principles of Software Validation" [21] established by the FDA.

ASM process and IEC 62304 standard The standard IEC 62304 classifies medical software in three classes on the basis of the potential injuries caused by software malfunctions, and defines the life cycle activities (points 5.1-5.8 of Section 5 in [3], also shown in Fig. 3) that have to be performed and appropriately documented when developing medical software. Each activity is split into tasks that are mandatory or not, depending on the class of software. The standard does not prescribe a specific life cycle model, nor it gives indications on methods and techniques to apply. Users are responsible to map their model to the standard.

Step (5.1) essentially consists in defining a life cycle model, planning procedures and deliverables, establishing how to achieve traceability among system requirements, software requirements, software test and risks control. By using the ASMs, we follow an iterative life cycle model based on refinement. Procedures are modeling, validation, verification, and conformance checking. Deliverables are given in terms of a sequence of refined models, each model equipped with validation and verification results. Traceability is given by the conformance relation between abstract and refined models, at each refinement step. We do not consider risk management, although ASM tools can be used to predict possible risks by reasoning on models and checking incorrect behaviors or potential faults. Risk management was also not part of the case study description in [20].

1. A documented software requirements specification should provide a baseline for both V&V.
2. Developers should use a mixture of methods and techniques to prevent and to detect software errors.
3. Software V&V must be planned early and conducted throughout the software life cycle.
4. Software V&V should take place within the environment of an established software life cycle.
5. Software V&V process should be defined and controlled through the use of a plan.
6. Software V&V process should be executed through the use of procedures.
7. Software V&V should be re-established upon any (software) change.
8. Validation coverage should be based on software complexity and safety risk.
9. V&V activities should be conducted using the quality assurance precept of "independence of review".
10. Device manufacturer has flexibility in choosing how to apply these V&V principles, but retains ultimate responsibility for demonstrating that the software has been validated.

Fig. 4. General Principle for Software Validation (FDA)

Step (5.2) consists in defining, documenting and verifying software requirements. This is a continuous activity along the ASM process till the desired level of refinement, possibly to code level. For the HMCS, this is what is reported in Sects. 3, 4, 5 (there is no implementation in this case).

Steps (5.3 - 5.4) regard the definition of *design* from *specification*. In the ASM process, these steps are performed in a seamless manner along the chain of refined models. Already at ground level, system structure is captured, even if not completely, by the model signature. Design decisions and architectural choices are added at a certain level of the refinement. For example, for the hemodialysis machine, patient and hardware checks and error handling are dealt at level four.

Steps (5.5 - 5.7) regard software implementation and testing, at unit and integration levels. Using ASMs, a first code prototype could be obtained as last refinement step and conformance checking is possible. Otherwise, techniques for model-based testing and runtime verification can be used having models available. These techniques have not been shown for the HMCS since not requested. However, we refer to [5] as an example of application.

Step (5.8) includes the demonstration, by a device manufacturer, that software has been validated and verified. It is intrinsic to the ASM process.

ASM process compliant with the FDA principles FDA accepts the standard IEC 62304 for all levels of concern and pushes for an integration of software life cycle management and risk management activities. The organization establishes some general principles [21], reported in Fig. 4, as guidelines for software validation, and promotes the use of formal approaches.

We here discuss how the ASM process realizes these FDA guideline principles.

(1) In our process, requirements are specified and documented through a chain of models providing a *rigorous baseline for both validation and verification.*

(2) A continuous *defect prevention* is supported. At each level, faults and unsafe situations can be checked. Safety properties are proved on models, while software testing for conformance verification of the implementation is possible.

(3)-(6) The ASM process allows *preparation for software validation and verification* as early as possible, since V&V can start at ground level. These activities are *part of* the process, can be *planned* at different abstract levels, are *documented*, and supported by precise *procedures*, i.e., methods and techniques.

(7) In case of an implementation local change not affecting the model, our process requires to re-run conformance checking only; in case of a change affecting the specification at a certain level, it requires to re-prove refinement correctness, and to re-execute V&V from the concerned level down to the implementation.

(8) Regarding *validation coverage*, by simulation and testing, we can collect the coverage in terms of rules or code covered. This can be used by the designer to estimate if the validation activity is commensurate with the risk associated with the use of the software for the specified intended use.

(9) Since V&V are performed by exploiting mathematical-based techniques, they facilitate *independent evaluation* of software quality assurance.

(10) The ASM process allows a *device manufacturer to demonstrate that the software has been validated and verified*, both when an implementation is obtained as last model refinement step, and when it is an external code that has been checked for conformance.

7 Conclusions

We have presented the specification, validation, and property verification of a hemodialysis device by using ASMs and model refinement. By taking advantage of the case study, we have related our ASM-based design process with the current normative for developing medical software. The main advantages offered to the standards are: (i) an iterative software life cycle; (ii) models as a rigorous means for safety properties assurance; (iii) validation and verification performed continuously along the software life cycle, and always aimed at defect prevention; (iv) software quality evaluation performed in an objective and repeatable manner; and (v) demonstration that software has been validated and verified.

References

1. ISO 13485:2003 medical devices – quality management systems – requirements for regulatory purposes, 2003.
2. IEC 60601-1:2005 medical electrical equipment part 1: General requirements for basic safety and essential performance, 2005.
3. IEC 62304 - medical device software - software lifecycle processes, 2006.
4. ISO 14971:2007 medical devices – application of risk management to medical devices, 2007.

5. P. Arcaini, S. Bonfanti, A. Gargantini, A. Mashkoor, and E. Riccobene. Formal validation and verification of a medical software critical component. In *Formal Methods and Models for Codesign (MEMOCODE), 2015 ACM/IEEE International Conference on*, pages 80–89. IEEE, Sept 2015.
6. P. Arcaini, A. Gargantini, and E. Riccobene. AsmetaSMV: a way to link high-level ASM models to low-level NuSMV specifications. In *Proceedings of the 2nd International Conference on Abstract State Machines, Alloy, B and Z (ABZ 2010)*, volume 5977 of *Lecture Notes in Computer Science*, pages 61–74. Springer, 2010.
7. P. Arcaini, A. Gargantini, and E. Riccobene. Automatic Review of Abstract State Machines by Meta Property Verification. In *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010)*, pages 4–13. NASA, 2010.
8. P. Arcaini, A. Gargantini, and E. Riccobene. CoMA: Conformance monitoring of Java programs by Abstract State Machines. In *Runtime Verification*, volume 7186 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 2012.
9. P. Arcaini, A. Gargantini, and E. Riccobene. Offline Model-Based Testing and Runtime Monitoring of the Sensor Voting Module. In *ABZ 2014: The Landing Gear Case Study*, volume 433 of *Communications in Computer and Information Science*, pages 95–109. Springer International Publishing, 2014.
10. P. Arcaini, A. Gargantini, and E. Riccobene. Rigorous development process of a safety-critical system: from ASM models to Java code. *International Journal on Software Tools for Technology Transfer*, pages 1–23, 2015.
11. P. Arcaini, A. Gargantini, E. Riccobene, and P. Scandurra. A model-driven process for engineering a toolset for a formal method. *Software: Practice and Experience*, 41:155–166, 2011.
12. F. Boniol, V. Wiels, Y. A. Ameur, and K.-D. Schewe. *ABZ 2014: The Landing Gear Case Study Case Study Track, Held at the 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z Toulouse, France, June 2-6, 2014, Proceedings.* Springer International Publishing, 2014.
13. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer Verlag, 2003.
14. BRAUN. *Dialog+® Dialysis Machine - Instructions for Use: Software Version 9.1x.*
15. A. Carioni, A. Gargantini, E. Riccobene, and P. Scandurra. A Scenario-Based Validation Language for ASMs. In *Proceedings of the 1st International Conference on Abstract State Machines, B and Z (ABZ 2008)*, volume 5238 of *Lecture Notes in Computer Science*, pages 71–84. Springer-Verlag, 2008.
16. EU. Directive 2007/47/EC of the European Parliament and of the Council. Official Journal of the European Union, September 2007.
17. A. Gargantini, E. Riccobene, and S. Rinzivillo. Using Spin to Generate Tests from ASM Specifications. In *Abstract State Machines 2003*, volume 2589 of *Lecture Notes in Computer Science*, pages 263–277. Springer Berlin Heidelberg, 2003.
18. A. Gargantini, E. Riccobene, and P. Scandurra. A Metamodel-based Language and a Simulation Engine for Abstract State Machines. *J. UCS*, 14(12):1949–1983, 2008.
19. R. Jetley, S. Iyer, Purushothaman, and P. L. Jones. A formal methods approach to medical device review. *Computer*, 39(4):61–67, Apr. 2006.
20. A. Mashkoor. The Hemodialysis Machine Case Study. In *ABZ 2016: Proceedings of the 5th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, Lecture Notes in Computer Science. Springer, 2016.
21. U.S. Food and Drug Administration (FDA). General principles of software validation; final guidance for industry and FDA staff, version 2.0, Jan. 2002.