Integrating Formal Methods into Medical Software Development: the ASM approach

Paolo Arcaini[a], Silvia Bonfanti[b,c], Angelo Gargantini[b], Atif Mashkoor[c], Elvinia Riccobene[d]

[a] *Charles University, Faculty of Mathematics and Physics, Czech Republic*
[b] *Department of Economics and Technology Management, Information Technology and Production, Università degli Studi di Bergamo, Italy*
[c] *Software Competence Center Hagenberg GmbH, Austria*
[d] *Dipartimento di Informatica, Università degli Studi di Milano, Italy*

Abstract

Medical devices are safety-critical systems since their malfunctions can seriously compromise human safety. Correct operation of a medical device depends upon the controlling software, whose development should adhere to certification standards. However, these standards provide general descriptions of common software engineering activities without any indication regarding particular methods and techniques to assure safety and reliability.

This paper discusses how to integrate the use of a formal approach into the current normative for the medical software development. The rigorous process is based on the Abstract State Machine (ASM) formal method, its refinement principle, and model analysis approaches the method supports. The hemodialysis machine case study is used to show how the ASM-based design process covers most of the engineering activities required by the related standards, and provides rigorous approaches for medical software validation and verification.

Keywords: Abstract State Machines, medical device software, certification, modeling, validation, verification, hemodialysis device

1. Introduction

Medical devices are increasingly becoming software intensive. This paradigm shift also impacts patients' safety, as a software malfunctioning can cause injuries or even death to patients [46]. Therefore, assuring medical

software safety and reliability is mandatory, and methods and techniques for medical software validation and verification are highly demanded.

Several standards for the validation of medical devices have been proposed – as ISO 13485 [39], ISO 14971 [40], IEC 60601-1 [37], EU Directive 2007/47/EC [28] –, but they mainly consider hardware aspects of the physical components of a device, and do not mention the software component. The only reference concerning regulation of medical software is the standard IEC (International Electrotechnical Commission) 62304 [38]. This standard provides a very general description of common life cycle activities of the software development, without giving any indication regarding process models, or methods and techniques to assure safety and reliability. The U.S. Food and Drug Administration (FDA), the United States federal executive department that is responsible for protecting and promoting public health through the regulation and supervision of medical devices, although accepts the IEC 62304 standard, also pushes towards the application of rigorous approaches for software validation. In [60], FDA defines several broad concepts that can be used as guidance for software validation and verification, and requires these activities to be conducted throughout the software development life cycle. However, no particular technique or method is recommended.

Both IEC standard and FDA principles aim for more rigorous approaches to certify software of medical devices [41, 44]. Potential methods should allow writing well-defined models that can be used to guide the software development, to prove that safety-critical properties hold, and to guarantee conformance of running code to abstract specification of safe device operation (since, most of the time, software for medical devices is not developed from scratch). To be practical, potential methods should provide the tool support for modeling and analysis.

The formal approach based on Abstract State Machines (ASMs) [21] proposes an incremental life cycle model for software development based on model refinement, includes the main software engineering activities (specification, validation, verification, conformance checking), and is tool-supported [13]. Despite their rigorous mathematical foundation, ASMs can be viewed as pseudo-code (or virtual machines) working over abstract data structures. Therefore, ASMs are relatively easy to understand even by non-experts. The method has been successfully applied to numerous case studies [21], also in the context of medical software, as in [3, 15] for the rigorous development of an optometric measurement device software, and in [4] for the specification and verification of the Hemodialysis Machine Case Study (HMCS) [48].

Although we believe that the rigor of a formal method can improve the current normative and that ASMs have the required potential, evidence must be given about a smooth possible integration of the method into the standards for medical software development. Additionally, it must be studied how much the ASM process is compliant with the normative: which steps and activities of the standard IEC are covered by using ASMs, and which are not; which FDA principles are ensured, and to what extent.

In this paper, we take advantage of the results already presented in [4] for the HMCS to make such a compliance analysis with the aim to understand how far we are from proposing an ASM-based process for medical software certification. The current work improves [4] in several aspects: (a) precise analysis of the advantages and shortcomings of the ASM approach w.r.t. the current normative for medical software development; (b) specification and analysis of the HMCS performed at different levels of refinement to show how validation and verification are continuous activities in the process; (c) model visualization in terms of a graphical notation to provide better evidence of the software operation; (d) encoding of a Java prototype of the software controlling the hemodialysis device, in order to show the applicability of conformance checking techniques, thus showing how we deal with the main software engineering activities of the IEC 62304 standard in a formal way.

The paper is organized as follows. Sect. 2 introduces the current normative for medical software development. Sect. 3 briefly presents the ASM-based development process. Sect. 4 discusses compliance of the ASM process w.r.t. the normative: it shows how the steps and principles of the standards IEC and FDA are fulfilled by the ASM-based process. Sect. 5 takes advantage of the HMCS to show the application of the ASM process to a medical device for which a certification could be required; it first presents the specification of the HMCS by means of four levels of model refinement (at each level, all possible results concerning requirements validation and property verification are reported); then, it describes a Java prototype of the software controlling the machine, and a technique for conformance checking. Sect. 6 compares our approach with other formal approaches applied to the formalization of medical software and, in particular, to the HMCS. Sect. 7 concludes the paper.

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

Figure 1: IEC 62304 development process

2. Normative for medical software

Currently, the main normative for development and analysis of medical software is the standard IEC 62304 [38] and the "General Principles of Software Validation" [60] established by the FDA. We here briefly recall such regulations and their underlying principles since later we want to analyze which activities can be covered by the use of the ASM-based design process.

2.1. 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 Sect. 5 in [38], also shown in Fig. 1) 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 software class. The standard does not prescribe a specific life cycle model, nor it gives indications on methods and techniques to apply. Users are responsible for mapping the adopted life cycle model to the standard.

Step (5.1) essentially consists in defining a life cycle model, planning procedures and deliverables, choosing standards, methods and tools, establishing which activity requires verification and how to achieve traceability among system requirements, software requirements, software tests, and risks control. Step (5.2) consists in defining and documenting functional and non-functional software requirements. It also requires checking for traceability between software requirements and system requirements, including risk control measures in the software requirements, and re-evaluating risk analysis on the established software requirements. Step (5.3) regards the specification of the software architecture from the software requirements. It requires to describe the software structure, identify software elements, specify functional and performance requirements for the software elements, identify software elements related to risk control, and verify the software architecture w.r.t. the software requirements. Step (5.4) regards the refinement of the software

architecture into software units. Steps (5.5 - 5.7) regard software implementation and testing at unit, integration, and system levels. Step (5.8) includes the demonstration, by a device manufacturer, that software has been validated and verified.

2.2. FDA General Principles of Software Validation

FDA accepts the standard IEC 62304 for all levels of concerns and pushes for an integration of software life cycle management and risk management activities. The organization promotes the use of formal approaches for software validation and verification (V&V), and establishes the following general principles [60] as guidelines:

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 should 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 the software complexity and safety risks.
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.

3. ASM-based development process

Abstract State Machines (ASMs) are transition systems that extend Finite State Machines by replacing unstructured control states by algebraic structures, i.e., domains of objects with functions and predicates defined on them. A *state* represents the instantaneous configuration of the system and

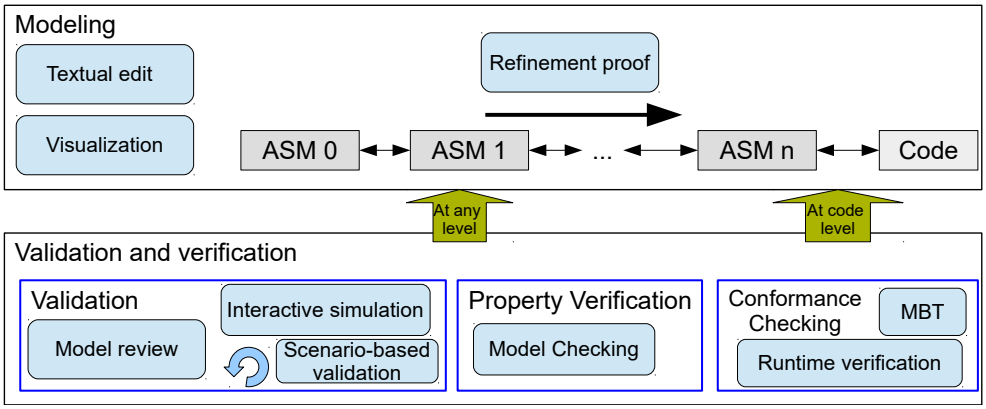


Figure 2: ASM-based development process

transition rules describe the state update. There is a limited but powerful set of *rule constructors*: `if-then` for guarded actions, `par` for simultaneous parallel actions, `choose` for nondeterminism (existential quantification), `forall` for unrestricted synchronous parallelism (universal quantification). A `macro` rule is a "named" rule that can be invoked in any point of the model. A *run* is a (finite or infinite) sequence of states s_0, s_1, ..., s_n, ..., where each s_i is obtained by applying the transition rules at s_{i-1}. Functions can be of different types. In particular, *controlled* functions can be updated by transition rules and represent the internal memory of the ASM; *monitored* functions, instead, cannot be updated by transition rules, but only by the environment, and represent inputs of the machine.

As shown in Fig. 2, the ASM-based development process is carried in an iterative and incremental fashion. The ASMETA (ASM mETAmodeling) framework[1] [13] provides different formal activities supporting the process.

Requirements modeling is based on model refinement; it starts by developing a high-level *ground model* (ASM 0 in Fig. 2) that *correctly* captures stakeholders requirements, and *consistently* (i.e., free from specification inconsistencies) reflects the intended system behavior. However, it does not need to be *complete*, i.e., it may not specify all stakeholder requirements. ASM specifications can be edited by using a *concrete syntax* [33] in a *textual editor*, and graphically *visualized* [5].

Starting from the ground model, through a sequence of *refined* models,

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

further functional requirements can be specified and a complete architecture of the system is defined. At each refinement step, if a particular kind of refinement (called *stuttering refinement*) has been applied, refinement correctness can be automatically checked by the *refinement prover* [11]. Otherwise, a hand-proof must be supplied. The refinement process can stop at any desired level of detail, possibly providing a smooth transition from specification to implementation, which can be seen as the last low-level refinement step.

At each level of refinement, different *validation* and *verification* (V&V) activities can be performed. Model validation is possible by means of an *interactive simulator* [33] and a *validator* [26] which allows to build and execute *scenarios* of expected system behaviors. Automatic *model review* (a form of static analysis) is also possible: it allows to check if a model has sufficient *quality* attributes (i.e., minimality, completeness, and consistency). Property verification of ASMs is possible by means of a *model checker* [6] that verifies both *Computation Tree Logic* (CTL) and *Linear Temporal Logic* (LTL) formulas.

Implementation can be either automatically derived from the model [19] or externally provided. In the former case, the conformance w.r.t. the specification should be assured by the translator; in the latter case, *conformance checking* must be executed. Both *Model-Based Testing* (MBT) and *Runtime Verification* can be applied to check whether the implementation conforms to its specification [10]. We support conformance checking w.r.t. Java code. The MBT feature of ASMETA [32] can be used to automatically generate tests from ASM models and, therefore, to check the conformance *offline*; the support for runtime verification [8], instead, can be used to check the conformance *online*.

4. ASM-based certification

The ASM-based process can be used for developing software of (distributed) medical devices whose (continuous) behaviour can be discretized in a set of states and transitions between them. We are not able to deal with continuous time, although a notion of *reactive timed ASMs* [56] has been proposed.

We here discuss how compliant the ASM-based process is w.r.t. the existing normative in the field of medical software, which activities of the IEC standard can be covered by the use of ASMs, which FDA principles are ensured by the use of ASMs, how the rigor of a formal method such as ASMs

can improve the current normative, and what can not be captured by or is out of the scope of ASMs. Aim of such analysis is to understand how far we are from proposing an ASM-based process for medical software certification.

4.1. Compliance of the ASM process with the IEC standard

Regarding step (5.1) of the IEC 62304 standard, ASMs can supply a precise iterative and incremental life cycle model based on model refinement. Life cycle procedures are modeling, validation, verification, and conformance checking, the last applicable also at the maintenance phase. Deliverables are given in terms of a sequence of refined models, each one equipped with validation and verification results. Traceability is given, at each refinement step, by the conformance relation between abstract and refined models.

ASMs do not support activities peculiar to risk management, although ASM formal verification can be used to check the *absence* of the risks identified by risk analysis. However, risk management sometimes requires to assess the *probability* of risk occurrence: ASMs do not have yet a mature support for such probabilistic analysis.

Once the ASM-based process is established as development model, the subsequent life cycle activities (steps 5.2-5.7) prescribed by the standard can be devised precisely. When a formal method is used, (software) system requirements (step 5.2) and design (steps 5.3-5.4) are expressed in detail by means of a mathematical model, carefully analyzed and checked before the implementation development. When developing such a formal model, one has to translate the *informal requirements*, which are expressed in natural language, diagrams, and tables, into a mathematical language which has a formally defined semantics [58]. Informal requirements are the results of the *requirements gathering activity* (which is also required by step 5.2) which is out of the scope of the ASM method, and thus complementary techniques should be used to this purpose. An example of informal requirements is the HMCS description in [48], that constitutes our input document to cover steps 5.2 to 5.4 for the case study.

There is a tight feedback loop between the informal requirements description and the formal specification. Indeed, one of the main benefits of the formal specification is its ability to uncover problems and ambiguities in the informal requirements. Note that 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. Furthermore, ASMs are particularly suitable for modeling functional requirements,

while non-functional requirements cannot be easily handled.

Thanks to the model refinement mechanism, steps (5.2 - 5.4) are covered by the continuous activity of modeling and verifying software requirements along the ASM process till the desired level of refinement, possibly to code level. For the HMCS, this is what is reported in Sects. 5.2, (till a Java prototypical implementation of the case study – see Sect. 5.3). Already at the ground level, software structure is captured, even if not completely, by the model signature (i.e., domains and functions defined on them), while software behavior is specified by means of transition rules. Model refinement and decomposition can help manage the complexity of systems and move from a global view of the system to a component (or unit) view. Design decisions and architectural choices are added along model refinement. For example, for the HMCS, different operation phases are introduced through different levels of refinement, and patient treatments, error handling and property verification are increasingly dealt with at each level. Risk control is performed in terms of verification of required (functional) safety properties, assurance of quality properties, and design of critical scenarios.

Steps (5.5 - 5.7) concern code and testing. Although in ASMs a code prototype could be obtained through a translator as last model refinement step, usually we expect code to be developed by a vendor and implemented by the use of powerful programming techniques and languages. Thus, the ASM process does not fully cover these development steps. However, having executable models available, ASM techniques for conformance checking (model-based testing and runtime verification) are applicable. This is shown for the HMCS in Sect. 5.3 by using a prototypical Java implementation.

Regarding step (5.8), if a device manufacturer adopts the ASM process, demonstration that software has been validated and verified is straightforward, since validation and verification are continuous activities along the process, and conformance checking is possible on the subsequent released versions of the software.

4.2. ASM process compliant with the FDA principles

By proposing the ASM process for medical software development, we respond to the request of using formal approaches for software validation and verification that the FDA organization promotes. Here, we discuss how ASM V&V activities achieve the FDA principles. We still miss a way to integrate software life cycle management and risk management.

(1) Using ASMs, requirements are specified and documented by means of a chain of models providing a *rigorous baseline for both validation and verification*.

(2) Continuous *defect prevention* is supported. At each modeling 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 changes only regard the software implementation and do not affect the model, our process requires to re-run conformance checking only; in case a software change requires to review the specification at a certain level, then refinement correctness must be re-proved and V&V re-executed 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 unambiguous 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*: if an implementation is obtained as the last model refinement step, it is correct-by-construction due to the proof of refinement correctness; if the code has been developed by a vendor, conformance checking can guarantee correctness w.r.t. a verified model.

5. Hemodialysis device case study

In this section, we exemplify the application of the ASM-based development process by providing a formal specification of the HMCS. In Sect. 5.1 we provide a brief description of the case study[2]. In Sect. 5.2 we show how

[2]Due to space limitation, we are not able to report the long list of requirements presented in [48]. The reader can access the document also from here: http://www.cdcc.

the ASM method can be used to support (in a formal way) the activities required by steps 5.2-5.4 of the IEC 62304 standard, and in Sect. 5.3 to support those required by steps 5.5-5.7. We also show how the FDA principles are concretely fulfilled by the early and continuous V&V activities of the ASM process.

5.1. Case study description

Hemodialysis is a medical treatment that uses a device to clean the blood. The hemodialysis device transports the blood from and to the patient, filters wastes and salts from the blood, and regulates the fluid level of the blood. The connection between the patient and the device is surgically created by means of a venous and an arterial access.

During the therapy, the device extracts the blood through the arterial access. The dialyser separates the metabolic waste products from the blood. At the end, the clean blood is pumped back to the patient. A therapy session is divided into three phases: *preparation*, *initiation*, and *ending*.

The first operation executed in the *preparation phase* is an automatic test to check all the device functionalities. After that, the *concentrate* for the therapy is connected and a nurse sets all rinsing parameters. The tubing system is connected to the machine and filled with saline solution. Afterwards, the nurse prepares the heparin pump and inserts the treatment parameters. At the end of the preparation phase, the dialyser is connected to the machine and rinsed with the saline solution.

During the *initiation phase*, the patient is connected to the device through the arterial access and the tubes are filled with blood until the *venous red detector* (VRD) sensor detects that they are full. Subsequently, the patient is connected venously and the therapy starts. During the therapy, the blood is extracted by the *blood pump* (BP) and is cleaned by the dialyser using the *dialysing fluid* (DF).

When the therapy is finished, the machine starts the *ending phase*. The patient is disconnected arterially and the saline solution is infused venously. When the solution is infused completely, the patient is disconnected also venously. After that, the dialyser and the cartridge are emptied. Finally, an overview of the therapy is shown on the device display.

`faw.jku.at/ABZ2016/HD-CaseStudy.pdf`.

refinement	#monitored functions	#controlled functions	#derived functions	#rule declarations	#rule	#properties
Ground model	0	1	8	5	11	0
1st - preparation phase	52	17	8	68	242	6
2nd - initiation phase	91	36	14	143	578	46
3rd - ending phase	101	39	15	159	648	52

Figure 3: Models data

5.2. Modeling by refinement

In modeling the HMCS, we proceeded through refinement. A peculiarity of the case study [48] is that the device behavior is clearly divided in phases, each characterized by the execution of activities (or sub-phases), as shown in Table 1.

At the highest level of abstraction, the *ground model* gives the overall abstract view of the whole device that goes through three phases: the `PREPARATION` of the device, the execution (or `INITIATION`[3]) of the therapy, and the termination (or `ENDING`) of the process. Then, we proceeded by refining each of these phases. Each refinement step models all the (possible) activities – that lead the device to go through specific sub-phases – performed in the phase and all the controls that are done, with related errors and alarms. The deepest nesting is of four levels, since in phase `INITIATION` the activity `THERAPY_RUNNING` requires that the `THERAPHY_EXEC`ution considers subsequent operations on the `arterialBolus`. Fig. 3 reports some data of the chain of the four models that form the complete specification.

While the ground model is rather simple and it has no monitored function and no requirement property, all refinements add functions of all types, rules, and properties, depending on the phase they refer to. Note that the second refinement allows to prove the majority of requirements (40 more w.r.t. the previous refinement), since it models the main part of the therapy.

5.2.1. Ground model

As said before, the ground model simply describes the transitions between the phases constituting a hemodialysis treatment, without any additional detail. Code 1 shows the ground model written using the ASMETA concrete syntax. The main rule simply executes rule `r_run_dialysis` that, depending

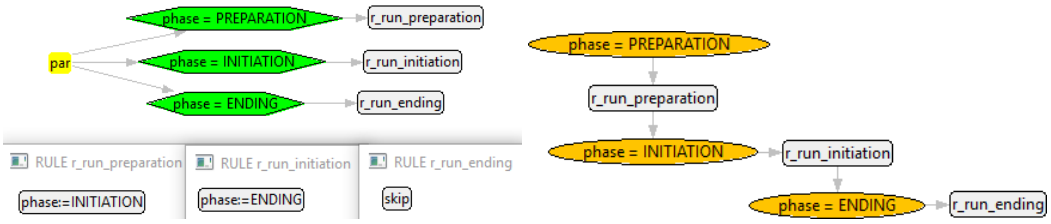
[3]Note that we use `INITIATION` to denote this phase to be consistent with the case study [48], although the term is misleading.

Table 1: Hemodialysis device phases

phase			
PREPARATION	**prepPhase** AUTO_TEST CONNECT_CONCENTRATE SET_RINSING_PARAM		
	TUBING_SYSTEM	**tubingSystemPhase** CONNECT_AV_TUBES CONNECT_ALL_COMP SET_SALINE_LEVELS INSERT_BLOODLINES PRIMING CONNECT_AV_ENDS	
	PREPARE_HEPARIN		
	SET_TREAT_PARAM	**treatmentParam** BLOOD_CONDUCTIVITY BIC_AC BIC_CONDUCTIVITY DF_TEMP DF_FLOW UF_VOLUME THERAPY_TIME MIN_UF_RATE MAX_UF_RATE MAX_AP DELTA_AP PERC_DELTA_TMP LIMITS_TMP MAX_TMP EXTENDED_TMP MAX_BEP STOP_TIME_H BOLUS_VOLUME_H RATE_H ACTIVATION_H SYRINGE_TYPE	
	RINSE_DIALYZER	**rinsePhase** CONNECT_DIALYZER FILL_ART_CHAMBER FILL_VEN_CHAMBER FILL_DIALYZER	
INITIATION	**initPhase**		
	CONNECT_PATIENT	**patientPhase** CONN_ART START_BP BLOOD_FLOW FILL_TUBING CONN_VEN END_CONN	
	THERAPY_RUNNING	**therapyPhase** START_HEPARIN	
		THERAPY_EXEC	**arterialBolusPhase** WAIT_SOLUTION SET_ARTERIAL_ BOLUS_VOLUME CONNECT_SOLUTION RUNNING_SOLUTION
		THERAPY_END	
ENDING	**endingPhase**		
	REINFUSION	**reinfusionPhase** REMOVE_ART CONN_SALINE START_SALINE_INF CHOOSE_NEXT_REINF_STEP RUN_SALINE_INF START_SALINE_REIN RUN_SALINE_REIN REMOVE_VEN	
	DRAIN_DIALYZER EMPTY_CARTRIDGE THERAPY_OVERVIEW		

```
asm HemodialysisGround

signature:
  enum domain Phases = {PREPARATION | INITIATION | ENDING}
  controlled phase: Phases

definitions:
  macro rule r_run_preparation =
    phase := INITIATION

  macro rule r_run_initiation =
    phase := ENDING

  macro rule r_run_ending =
    skip

  macro rule r_run_dialysis =
    par
      if phase = PREPARATION then
        r_run_preparation[]
      endif
      if phase = INITIATION then
        r_run_initiation[]
      endif
      if phase = ENDING then
        r_run_ending[]
      endif
    endpar

  main rule r_Main = r_run_dialysis[]

default init s0:
  function phase = PREPARATION
```

Code 1: Ground model



(a) Basic visualization (b) Semantic visualization

Figure 4: Ground model visualization

on the current `phase`, executes the corresponding rule:

- `r_run_preparation`, refined in the first refinement step (see Sect. 5.2.2);
- `r_run_initiation`, refined in the second refinement step (see Sect. 5.2.3);
- `r_run_ending`, refined in the third refinement step (see Sect. 5.2.4).

Fig. 4 shows a graphical representation [5] of the ground model, in the two ways supported by ASMETA, *basic visualization* and *semantic visualization*. The basic visualization permits to show the syntactical structure of the ASM in terms of a tree (similar to an AST); the notation is inspired by the classical flowchart notation, using green rhombuses for guards and grey rectangles for rules. The leaves of the tree are the update rules and the macro call rules. For each macro rule in the model, there is a tree representing the definition of the rule; double-clicking on a macro call rule shows the tree of the corresponding macro rule. The basic visualization of the model of Code 1 (starting from rule `r_run_dialysis`) is shown in Fig. 4a. The figure shows the visualization provided by the tool when all the macro rules are shown (i.e., the user has

double-clicked on all the call rules).

The semantic visualization provides a more advanced way of representing ASMs, trying to extrapolate part of the behavior from the model. As observed before, some systems naturally evolve through phases (or modes), called *control states* in [21], that are represented by a suitable function of the model (called *phase function*). Phases and transitions between them can sometimes be statically identified directly in the model. This visualization tries to identify a phase function in the model and shows how the system evolves through these phases by the execution of the transition rules. The visualization consists in a graph where control states are shown using orange ellipses. Note that a control state is not an ASM state, but an abstraction of a set of ASM states having the same value for the phase function. The semantic visualization of the ground model is shown in Fig. 4b. The system starts in the `PREPARATION` phase and moves to the `INITIATION` phase by executing rule `r_run_preparation`, from which it moves to the `ENDING` phase by the execution of rule `r_run_initiation`. In the `ENDING` phase, rule `r_run_ending` is executed, that, however, does not modify the phase. The simple visual inspection was sufficient to give us confidence that the model correctly evolves through the three top-level phases.

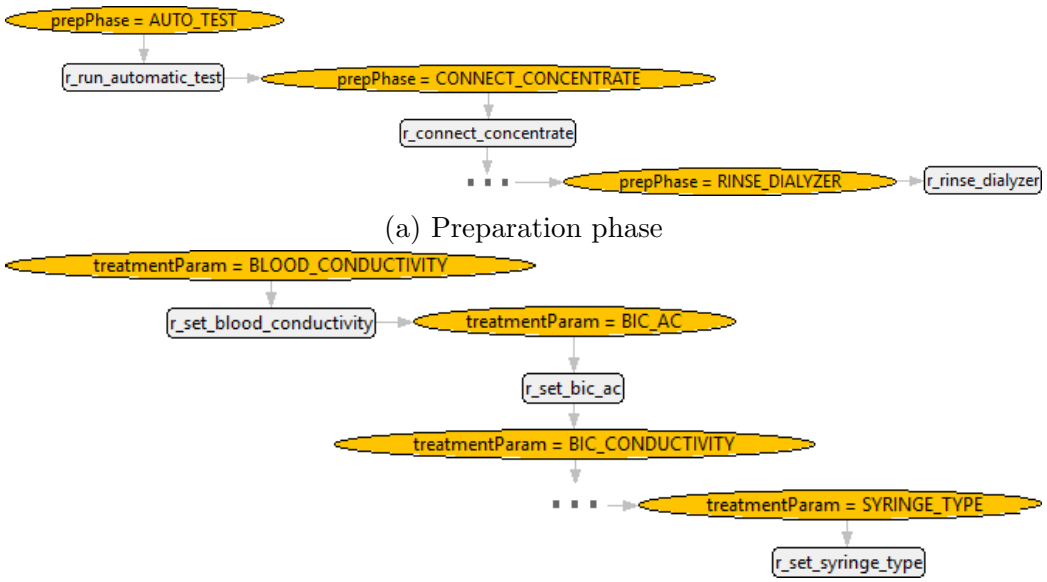
In the following refinement steps, we will only show the semantic visualizations of the models. The complete textual specifications are available online[4].

5.2.2. First refinement: preparation phase

The first refinement extends the ground model by refining the `PREPARATION` phase. As shown in Fig. 5a, the preparation consists in a sequence of activities, specified by function `prepPhase`. For each value of `prepPhase`, a given rule performs some actions related to the device preparation and updates `prepPhase` to the next value. Examples of these activities are the concentrate connection and the dialyzer rinsing. As shown in Table 1, some phases are further divided in sub-phases. For example, Fig. 5b shows how phase `SET_TREAT_PARAM` is specified by the treatment parameter (function `treatmentParam`). Also in this case, the sub-phases are executed in sequence.

The correctness of each refinement step has been proved with the *refinement prover* integrated in ASMETA that checks a particular kind of

[4]Models are available at http://fmse.di.unimi.it/sw/SCP2017.zip



(a) Preparation phase

(b) Treatment parameter

Figure 5: First refinement – Semantic visualization

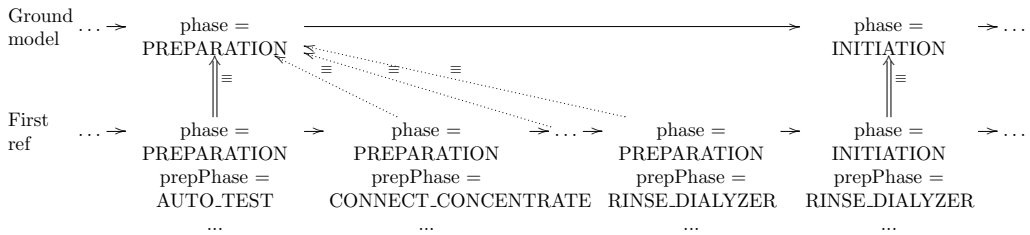


Figure 6: Hemodialysis case study – Relation between a refined run and an abstract run

refinement called *stuttering refinement* [11]. A model R is a correct stuttering refinement of a model A iff for each run ρ^R of R there is a run $\rho^A = S_1, S_2, \ldots, S_n$ of A such that ρ^R can be split in sub-runs $\rho_1^R, \rho_2^R, \ldots, \rho_n^R$ where all the states of ρ_i^R are conformant with S_i $(i = 1, \ldots, n)$, according to a given conformance relation $\equiv$. As conformance relation, we usually use the equality between some selected locations (called *locations of interests*) of the two models.

The first refined model is a correct stuttering refinement of the ground model, using as conformance relation the equality on the `phase` function. Fig. 6 shows the correspondence of a refined run with an abstract run. We

can see that, in the run of the ground model (abstract run), the machine goes from a state in which `phase` is `PREPARATION` to a state in which phase is `INITIATION` in one step. Instead, in the run of the refined model (refined run), there is a sequence of intermediate states in which `phase` remains in `PREPARATION`. These states are all conformant with the first abstract state. The state in the refined run in which `phase` becomes `INITIATION` is conformant with the second abstract state.

Validation and Verification. Starting from the first refinement, we applied *model review*. 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). In this model, we found that controlled function `machine_state` was initialized but never updated (violation of meta-property MP7 that requires that every controlled location is updated and every location is read). Although this is not a real fault of the model, it could make the model less readable, since a reader may expect an update of the function (since it is controlled). Declaring the function static made apparent that, in this refinement step, the function is not updated.

When modeling other case studies [3, 12, 14], we extensively used *interactive simulation* [33] that allowed us to observe some particular system executions. In this case study, we could largely reduce the effort spent in simulation, since by semantic visualization we could get a feedback regarding the control flow similar to that provided by simulation. Fig. 7 shows a simulation trace (two steps) of the current model: the values chosen by the user for the monitored functions are shown in the *monitored* part of the state, while the updates computed by the machine are shown in the *controlled* part. Transitions between phases can be discovered also through simulation, but in a less direct way than with semantic visualization (see Fig. 5a). A further advantage of semantic visualization is that it also shows the rule that changes a given phase. However, simulation shows ASM states, whereas semantic visualization only shows control states given by the value of the phase function. Therefore, if we are interested in observing the exact ASM runs, we still have to use simulation. Instead, if we are only interested in knowing how the machine evolves through its phases, the semantic visualization is

```
Insert a boolean constant for auto_test_end:
true
<State 0 (monitored)>
auto_test_end=true
</State 0 (monitored)>
<State 1 (controlled)>
alarm(DF_PREP)=false
alarm(SAD_ERR)=false
alarm(TEMP_HIGH)=false
dialyzer_connected_contr=false
error(DF_PREP)=false
error(SAD_ERR)=false
error(TEMP_HIGH)=false
phase=PREPARATION
prepPhase=CONNECT_CONCENTRATE
preparing_DF=false
signal_lamp=GREEN
</State 1 (controlled)>
```

```
Insert a boolean constant for conn_concentrate:
true
<State 1 (monitored)>
conn_concentrate=true
</State 1 (monitored)>
<State 2 (controlled)>
alarm(DF_PREP)=false
alarm(SAD_ERR)=false
alarm(TEMP_HIGH)=false
dialyzer_connected_contr=false
error(DF_PREP)=false
error(SAD_ERR)=false
error(TEMP_HIGH)=false
phase=PREPARATION
prepPhase=SET_RINSING_PARAM
preparing_DF=true
signal_lamp=GREEN
</State 2 (controlled)>
```

Figure 7: Simulation trace of first refinement model

enough.

Instead of interactive simulation, we mainly performed *scenario-based validation* [26] that permits to automatize the simulation activity, so scenarios can be re-run after specification modifications. In scenario-based validation the designer writes a scenario specifying the expected behavior of the model; scenarios are similar to test cases. The *validator* reads the scenario and executes it using the simulator. The validator language provides constructs to express scenarios 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 to the user's actions. We wrote several scenarios for the different refinement steps. 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 *blocks of actions* that can be reused in different scenarios: a block is a named sequence of commands delimited by keywords `begin` and `end`. A command block can be defined in any scenario and can be called by means of the command `execblock` in other parts of the same scenario or in other scenarios. A block can also be nested in another block.

Code 2 shows an example of scenario for the first refined model reproduc-

```
scenario completeTherapyRef1

load HemodialysisRef1.asm

begin initStatePrep
  check phase = PREPARATION;
  check prepPhase = AUTO_TEST;
  check rinsingParam = FILLING_BP_RATE;
  ...
end

begin preparationPhase
  begin automaticTest
    set auto_test_end := true;
    step
  end
  begin connectConcentrate
    check prepPhase = CONNECT_CONCENTRATE;
    check signal_lamp = GREEN;
    set conn_concentrate := true;
    step
  end
  ...
end

check phase = INITIATION;
check bp_status = STOP;
check bp_status_der = STOP;
step

check phase = ENDING;
step
```

Code 2: Scenario for the first refinement

ing the whole therapy process. We defined the block `initStatePrep`, since its instructions regarding the initial state will be reused in scenarios written for other refinement steps. We also defined the block `preparationPhase` containing instructions related to the `PREPARATION` phase. Such block is further divided in sub-blocks (e.g., `automaticTest`); indeed, some scenarios will reuse the whole block `preparationPhase`, while others will reuse only some sub-blocks and redefine some others.

Once a modeler is confident enough that the model correctly reflects the intended requirements, heavier techniques can be used for property verification. The case study document [48] reports a list of safety requirements (divided between *general* (S1-S11) and *software* (R1-R36) requirements) that must be guaranteed. We have specified them as LTL properties and verified using the integrated model checker [6] that translates ASM models to models of the model checker `NuSMV`. Whenever a property is violated, the designer can inspect the returned counterexample to understand whether the problem is in the model that is actually faulty, or in the property that wrongly specifies the requirement; since counterexamples are returned as ASM runs, such task should be easy for the developer. Note that, thanks to meta-property MP10 of the model reviewer, we are also able to detect whether a property is *vacuously* satisfied, i.e., it is true regardless the truth value of some of its sub-expressions.

Since NuSMV works on finite state models, we have slightly modified our models by abstracting all the infinite domains with finite ones. As future

work, we plan to support the translation to nuXmv [52] that allows the verification of infinite state systems.

Each requirement has been proved as soon as possible in the chain of refinements, i.e., in the model that describes the elements involved in the requirement. At this refinement step, we were able to express only 13 of the 47 requirements; these are software requirements regarding the flow of bicarbonate concentration into the mixing chamber, the heating of the dialyzing fluid, and the detection of safety air conditions.

For example, requirement R20 states that "if the machine is in the preparation phase and performs priming or rinsing or if the machine is in the initiation phase and if the temperature exceeds the maximum temperature, then the software shall disconnect the dialyzer from the DF and execute an alarm signal." The requirement has been formalized in LTL as follows.

```
//R20
g((phase = PREPARATION and dialyzer_connected_contr and prepPhase =
    RINSE_DIALYZER and not error(TEMP_HIGH) and current_temp = HIGH) implies x(
    error(TEMP_HIGH) and alarm(TEMP_HIGH) and not dialyzer_connected_status))
```

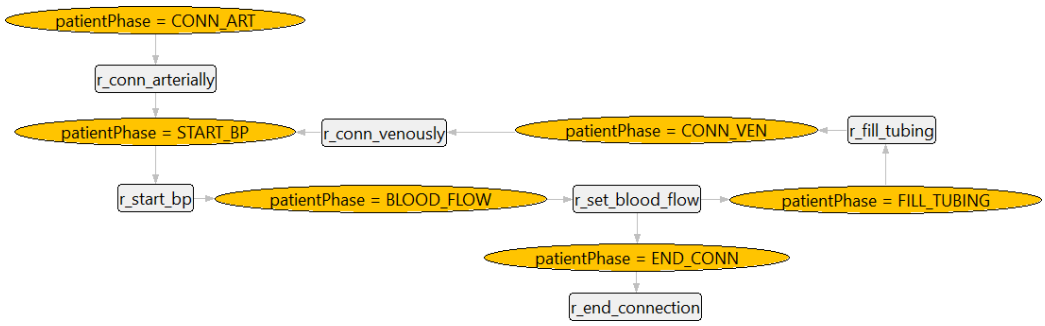
Note that some requirements are strictly related and somehow redundant and, therefore, can be verified together with only one property. This is the case of requirements R18 and R19, and requirements R23-R32.

```
//R18−R19
g((phase = PREPARATION and prepPhase = RINSE_DIALYZER and
    dialyzer_connected_contr and not error(DF_PREP) and preparing_DF and not
    detect_bicarbonate) implies x(error(DF_PREP) and alarm(DF_PREP) and not
    dialyzer_connected_status))
//R23−R32
g((phase = PREPARATION and prepPhase = TUBING_SYSTEM and passed1Msec and
    currentSAD != PERMITTED and current_air_vol != PERMITTED and not error(
    SAD_ERR)) implies x(error(SAD_ERR) and alarm(SAD_ERR)))
```

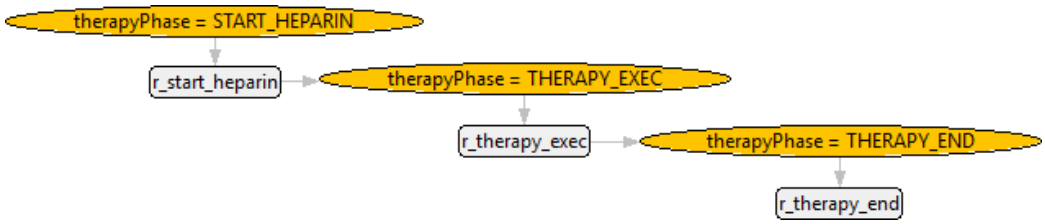
Requirements R20 and R23-R32 are related both to the preparation and initiation phases; R23-R32 also consider the ending phase. At this level of refinement, the corresponding properties can only check the preparation phase; they will be refined in the second refinement to take into consideration the initiation phase (see Sect. 5.2.3), and in the third refinement for considering the ending phase (see Sect. 5.2.4).

5.2.3. Second refinement: initiation phase

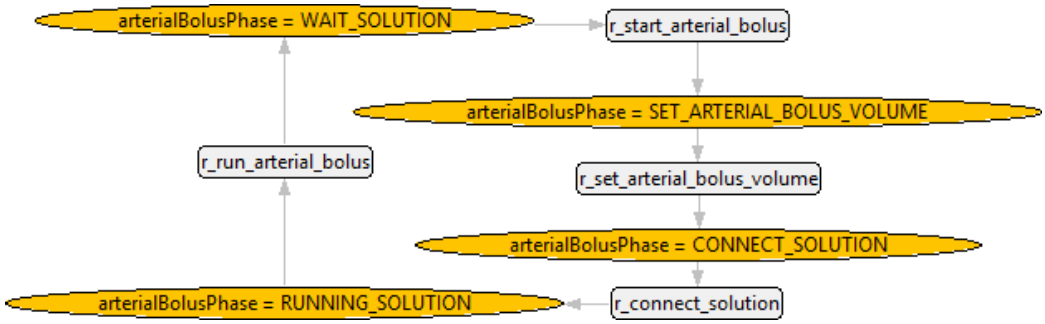
The second refinement extends the first one by refining the `INITIATION` phase. As shown in Table 1, the phase is further divided into two phases



(a) Patient phase



(b) Therapy phase



(c) Arterial bolus phase

Figure 8: Second refinement – Semantic visualization

(recorded by function `initPhase`): the connection of the patient (`CONNECT_PATIENT`) and the running of the therapy (`THERAPY_RUNNING`). As shown in Fig. 8a, function `patientPhase` indicates in which step the patient is during the connection. We can see that the patient is initially connected arterially; then the blood pump is activated to extract the blood from the patient (in state `BLOOD_FLOW`). In this state, rule `r_set_blood_flow` can follow two different paths:

- `patientPhase` is updated to `FILL_TUBING`. Then, the operator sets the blood flow and the blood pump stops when the blood fills the tubes be-

tween the patient and the dialyzer. After this, the patient is connected venously and the blood pump is restarted to fill the tubes between the dialyzer and the patient's vein.

- `patientPhase` is updated to `END_CONN`. Then, the therapy can start (i.e., `initPhase` goes to `THERAPY_RUNNING`).

Note that, in this case, the semantic visualization is not sufficient to completely understand the model behavior, since the paths are taken in two subsequent executions of the rule. This can only be discovered by simulation.

The therapy status is specified by function `therapyPhase` (see Fig. 8b). When the therapy status is `THERAPY_EXEC`, it is further specified by function `arterialBolusPhase`, whose semantic visualization is shown in Fig. 8c. Such sub-phase consists in the infusion of saline solution and it is activated by the operator. `arterialBolusPhase` is initially in state `WAIT_SOLUTION` until the operator presses the start button. After that, the doctor sets the volume of the saline solution, the solution is connected to the machine, and the infusion starts. When the predefined volume is infused, the `arterialBolusPhase` returns to `WAIT_SOLUTION` state until the operator restarts again the saline infusion. Also in this case, semantic visualization does not allow to fully understand the machine behavior: the initial state of the graph in Fig. 8c can only be discovered through simulation.

As required by our modeling process, before any further validation and verification activity of the requirements, it is necessary to guarantee correctness of the refinement step. This has been carried out, similarly to what described at the end of Sect. 5.2.2, by means of the refinement prover.

Validation and Verification. By model review, we found that some locations were *trivially updated* (meta-property MP4), i.e., that the value of the location before the update was always equal to the new value. This means that the update is not necessary. Removing trivial updates is important because the reader may have the feeling that the ASM is modifying its state when it is not. The trivial updates were related to `signal_lamp` when updated to `GREEN`, and `error(UF_DIR)` and `error(UF_RATE)` when updated to true. The update of `signal_lamp` was indeed unnecessary and we removed it; the updates of `error(UF_DIR)` and `error(UF_RATE)`, instead, were not correct since the locations had to be updated to false and so we fixed the fault.

Moreover, we detected some violations of meta-property MP7 requiring that every controlled location is updated and every location is read. We found that functions `bf_err_ap_low`, `reset_err_pres_ap_low` were never up-

```
scenario completeTherapyRef2

load HemodialysisRef2.asm

begin initStateInit
  execblock completeTherapyRef1.initStatePrep;

  check patientPhase = CONN_ART;
  check arterialBolusPhase = WAIT_SOLUTION;
  ...
end

execblock completeTherapyRef1.preparationPhase;
```

```
begin initiationPhase
  begin patientConnection
    check phase = INITIATION;
    check initPhase = CONNECT_PATIENT;
    check patientPhase = CONN_ART;
    set art_connected := true;
    step
    ...
  end
end

check phase = ENDING;
step
```

Code 3: Scenario for the second refinement

dated: this was due to a wrong guard in a conditional rule. This shows that model review is also useful in detecting behavioral faults. We found that also locations `error(ARTERIAL_BOLUS_END)`, `error(UF_BYPASS)`, and `error(UF_VOLUME_ERR)` were never updated. This is due to the fact that functions `error` and `alarm` share the domain `AlarmErrorType` representing the different alarms; for each alarm there is an error, except for `ARTERIAL_BOLUS_END`, `UF_BYPASS`, and `UF_VOLUME_ERR`. Therefore, locations `error(ARTERIAL_BOLUS_END)`, `error(UF_BYPASS)`, and `error(UF_VOLUME_ERR)` are actually unnecessary. We could have declared two different domains for errors and alarms, but we think that the specification would have been less clear and it would have been more difficult to keep the values of errors and alarms consistent. Therefore, we ignored the meta-property violation without changing the model.

We also wrote some scenarios for this refinement step, as the one shown in Code 3. We can see that the scenario reuses blocks `initStatePrep` and `preparationPhase` defined in scenario `completeTherapyRef1` for the first refined model (see Code 2).

At this modeling level, we were able to prove 23 more safety requirements. They concern patient connection, infusion of the saline solution when the patient is connected to the extra-corporeal blood circuit, pressure during the therapy, dialyzing fluid temperature, heparin infusion, air detected in the blood, and ultrafiltration process. Among these, we realized that some were not correctly described in [48]. For example, S1 states that "arterial and venous connectors of the EBC are connected to the patient simultaneously". The corresponding LTL property is as follows

```
g(art_connected_contr iff ven_connected_contr)
```

However, the property is false because the patient is connected *before* to the arterial connector and *then* to the venous connector.

Other requirements are instead ambiguous and so we had problems in formalizing them. For example, S5 states that "the patient cannot be connected to the machine outside the initiation phase, e.g., during the preparation phase." We did not know how to interpret "be connected": as the patient status of being attached to the machine, or as the atomic action performed by the operator of connecting the patient to the machine? The former interpretation would require to prove the following property:

```
g((art_connected_contr or ven_connected_contr) implies phase = INITIATION)
```

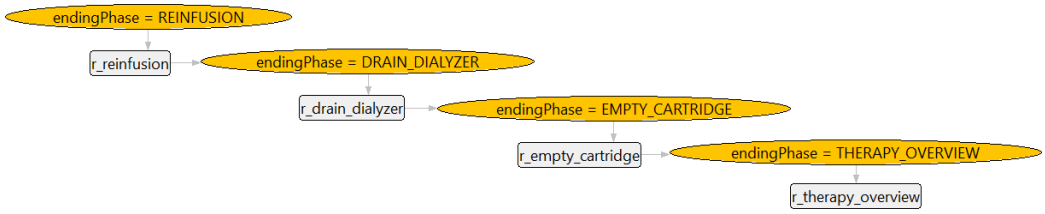
that, however, is false. Indeed, the patient can be attached to the machine also outside the `INITIATION` phase. The former interpretation, instead, would require to prove the two following properties:

```
g((not art_connected_contr and x(art_connected_contr)) implies phase = INITIATION)
g((not ven_connected_contr and x(ven_connected_contr)) implies phase = INITIATION)
```

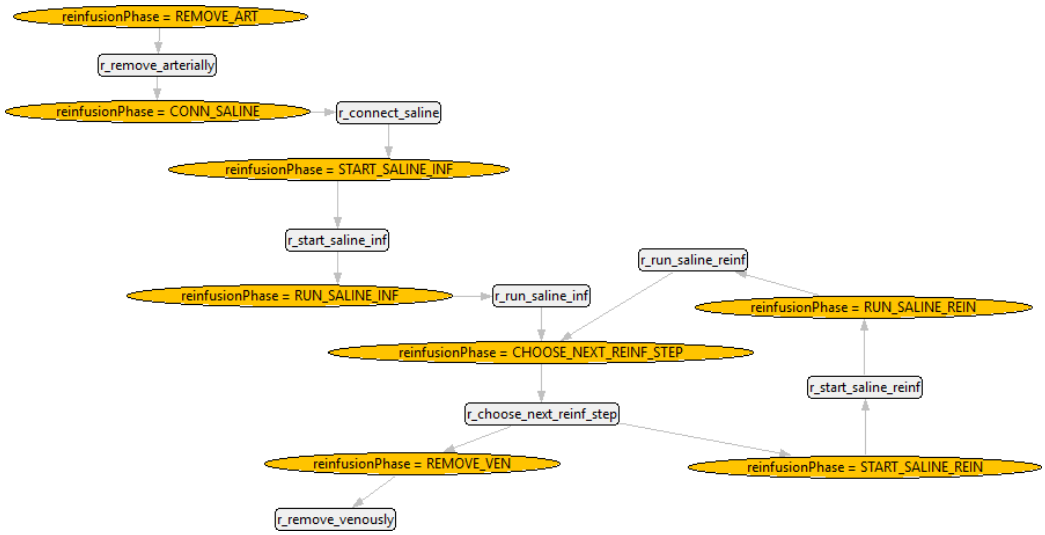
that are actually both true. It may be the case that this interpretation is not correct; this is a clear example of ambiguous requirement that would need a clarification from the stakeholders. Ambiguity also characterizes requirements S2, S6, S7, and R16, for which we were not able to provide a satisfactory formalization. For example, S6 requires that BP cannot be used outside the INITIATION phase; however, Sect. 3.2 of [48] states that BP must also be used in the ENDING phase: therefore, we were not sure how to interpret and formalize such requirement.

In this refinement step, we could refine properties related to requirements R20 and R23-R32 to take into consideration also the initiation phase.

```
//R20 updated
g(((phase = INITIATION and not error(TEMP_HIGH) and current_temp = HIGH) or (phase
    = PREPARATION and dialyzer_connected_contr and prepPhase = RINSE_DIALYZER
    and not error(TEMP_HIGH) and current_temp = HIGH)) implies x(error(TEMP_HIGH)
    and alarm(TEMP_HIGH) and not dialyzer_connected_status))
//R23−R32 updated
g((((phase = PREPARATION and prepPhase = TUBING_SYSTEM) or (phase =
    INITIATION and bp_status_der = START)) and (passed1Msec and currentSAD !=
    PERMITTED and current_air_vol != PERMITTED and not error(SAD_ERR))) implies x
    (error(SAD_ERR) and alarm(SAD_ERR)))
```



(a) Ending phase



(b) Reinfusion phase

Figure 9: Third refinement – Semantic visualization

5.2.4. Third refinement: ending phase

The third refinement extends the second one by refining the `ENDING` phase. As shown in Fig. 9a, the ending consists in a sequence of activities (specified by function `endingPhase`). When in `REINFUSION`, the phase is further refined by `reinfusionPhase`, whose semantic visualization is shown in Fig. 9b. The reinfusion consists in an initial sequence of activities for starting the infusion of the saline solution, followed by a loop in which the doctor performs the solution reinfusion. Rule `r_choose_next_reinf_step` is responsible for deciding the loop termination: either going to `START_SALINE_REIN` (i.e., the operator decides to continue the reinfusion) or to `REMOVE_VEN` (i.e., the operator disconnects the patient).

To complete modeling at this level, the refinement prover was used to prove that this model is a correct stuttering refinement of the second refine-

```
scenario installTubingTempHigh

load HemodialysisRef3.asm

execblock completeTherapyRef3.initStateEnd;

execblock completeTherapyRef1.automaticTest;
execblock completeTherapyRef1.connectConcentrate;
execblock completeTherapyRef1.setRinsingParam;
execblock completeTherapyRef1.installTubingSystem;
execblock completeTherapyRef1.prepareHeparin;
execblock completeTherapyRef1.setTreatmentParam;
```

```
check prepPhase = RINSE_DIALYZER;
check syringe_type_contr = syringe_type;
check rinsePhase = CONNECT_DIALYZER;
check preparing_DF = true;
set stop_DF_preparation := true;
step
check preparing_DF = false;
set dialyzer_connected := true;
...

execblock completeTherapyRef2.initiationPhase;
execblock completeTherapyRef3.endingPhase;
```

Code 4: Scenario for the third refinement – Triggering of error `TEMP_HIGH`

ment.

Validation and Verification. We applied model review also to this model, but we did not find any meta-property violation.

We also wrote some new scenarios. In particular, we wrote scenarios for reproducing the occurrence of some errors. Code 4 shows the scenario that triggers error `TEMP_HIGH` related to high temperature of the dialyzer fluid during the preparation phase. We can see that in this scenario we reused some sub-blocks of block `preparationPhase` defined in scenario `completeTherapyRef1` (see Code 2) as, for example, `automaticTest` and `connectConcentrate`. We did not use the whole block because the instructions related to the dialyzer rinsing had to be changed in order to trigger the error.

In this refinement step, we were able to prove three more requirements. Moreover, we could refine the property related to requirements R23-R32 in order to take into consideration also the ending phase.

```
//R23−R32 updated
g((((phase = PREPARATION and prepPhase = TUBING_SYSTEM) or (phase =
    INITIATION and bp_status_der = START) or (phase = ENDING and endingPhase =
    REINFUSION and not error_rein_press and bp_status_der = START)) and (passed1Msec
     and currentSAD != PERMITTED and current_air_vol != PERMITTED and not error(
    SAD_ERR))) implies x(error(SAD_ERR) and alarm(SAD_ERR)))
```

Note that there are four requirements (S3, S8, S9, and S10) that we do not consider in our work. They are all related to the blood flow rate; for example, S8 requires that the "blood flow rate should be adjusted, taking into consideration the AP" [48]. However, since continuous values have been discretized for doing model checking, we are not able to express such requirements as temporal properties. As said before, as future work we plan to

```
import org.asmeta.monitoring.*;

@Asm(asmFile = "models/HemodialysisRef3.asm")
public class HemodialysisMachine {
    HemodialysisMachinePanel dialog;

    @FieldToFunction(func = "phase")
    Phases phase = Phases.PREPARATION;
    @FieldToFunction(func = "preparing_DF")
    boolean preparing_DF = false;
    @FieldToFunction(func = "initPhase")
    InitPhase initPhase = InitPhase.CONNECT_PATIENT;
    ...
    @Monitored(func = "interrupt_dialysis")
    boolean interrupt_dialysis = false;
    @Monitored(func = "error_heparin_resolve")
    boolean error_heparin_resolve = false;
    @Monitored(func = "blood_conductivity")
    int blood_conductivity = HIGH;
    ...
    public HemodialysisMachine() {
        dialog = new HemodialysisMachinePanel(this);
        dialog.setDefaultCloseOperation(JDialog.DISPOSE_ON_CLOSE);
        dialog.setVisible(true);
    }

    @RunStep
    public void execDialysis() {
        if (phase == Phases.PREPARATION) {
            ...
        ...
        dialog.updateGUI();
    }

    @MethodToFunction(func="error")
    public boolean error(AlarmErrorType aet){
        return error[aet.ordinal()];
    }
}
```

Code 5: Java implementation of the HMCS

provide a mapping from ASM to nuXmv [52] that allows the verification of infinite states systems: in this way, we will also be able to verify such kind of requirements.

5.3. Conformance checking

We do not have access to the implementation of the hemodialysis device. Therefore, we have built a prototypical implementation in Java of the hemodialysis device software, in order to show the last part of the ASM-based process, i.e., the *conformance checking* between the implementation and the specification) and how the process can help perform the activities required by steps 5.5-5.7 of the IEC 62304 standard, as well as step 5.8 on subsequent releases of medical software. Techniques of conformance checking are also useful for validation coverage and re-verification of software upon changes, as required by the FDA principles.

The implementation faithfully reflects (or, at least, it should) the case study requirements; however, some components (e.g., the connection with the hardware) have not been implemented and so they have been substituted with mock objects. Part of the Java implementation is shown in Code 5. Note that the implementation has many details that are not present in the specification, as the graphical user interface (shown in Fig. 10) that allows to visualize the output of the device. The top part of the GUI shows the current state of the device, the bottom left part displays the status of alarms and errors (red means that there is an alarm/error, while green means that the component is working properly), and the bottom right part reports the

Figure 10: Hemodialysis device program GUI

configuration of the parameters.

The implementation has been developed by two authors who were only partially involved in the writing of the specification; in this way, we aimed at reproducing a setting in which the system designers (also responsible for model-based testing) are different from the developers.

Conformance checking can be done *offline* (i.e., *before* the deployment) by MBT or *online* (i.e., *after* the deployment) by runtime verification. We here show the application of the former approach to the case study and we refer the reader to [8, 12] for examples of applications of the latter approach.

In MBT [35, 61], *abstract test sequences* are derived from the specification; such sequences are then concretized in tests for the implementation. In order to generate abstract test sequences, we use the MBT feature of ASMETA [32] that first derives from the specification some test goals (called *test predicates*) according to some *coverage criteria* [31], and then generates sequences for covering these goals. For example, the *update rule coverage* criterion requires that each update rule is executed at least once in a test sequence and the update is not trivial (i.e., the new value is different from the current value of the location). A classical approach based on model checking is used for

```
- State 1 -
phase = PREPARATION
initPhase = CONNECT_PATIENT
therapyPhase = START_HEPARIN
interrupt_dialysis = FALSE
...

...
- State 46 -
phase = INITIATION
initPhase = CONNECT_PATIENT
therapyPhase = START_HEPARIN
interrupt_dialysis = FALSE
...
```

```
...
- State 53 -
phase = INITIATION
initPhase = CONNECT_PATIENT
therapyPhase = START_HEPARIN
interrupt_dialysis = FALSE
...
- State 54 -
phase = INITIATION
initPhase = THERAPY_RUNNING
therapyPhase = START_HEPARIN
interrupt_dialysis = FALSE
...
```

```
- State 55 -
phase = INITIATION
initPhase = THERAPY_RUNNING
therapyPhase = THERAPY_EXEC
interrupt_dialysis = TRUE
...
```

Figure 11: Abstract test sequence

generating tests: the ASM model is translated in the language of a model checker, and each test goal is expressed as a temporal property (called *trap property*); if the trap property is proved false, the returned counterexample is the *abstract test sequence* covering the test goal (and possibly also other test goals). For this work, we extended the ASMETA MBT component in order to work with the model checker NuSMV.

For the case study, we used the structural coverage criteria presented in [32]. Since the test generation approach is based on model checking, we used the same models we obtained to perform formal verification (having only finite domains). 980 test predicates have been built and 183 tests have been generated for covering them (in around 2 hours on a Linux PC with Intel(R) Core(TM) i7 CPU, 8 GB of RAM). Note that each generated test can cover more than one test predicate and we avoid generating tests for already covered test predicates. An example of test predicate for the update rule coverage criterion is:

phase = INITIATION and initPhase = THERAPY_RUNNING and therapyPhase = THERAPY_EXEC and interrupt_dialysis and therapyPhase != THERAPY_END

requiring to observe a state in which the update rule of `therapyPhase` to `THERAPY_END` fires and `therapyPhase` is not already equal to `THERAPY_END`. Note that a sequence covering this test predicate is guaranteed to exist if the specification has been previously checked with model review and no violation of MP4 (requiring that all the update rules are not always trivial) occurred.

The sequence covering the predicate is shown in Fig. 11. We can see that the test predicate holds in the last state of the sequence.

```
@Test
public void test() {
    HemodialysisMachine sut = new HemodialysisMachine();
    // check conformance
    assertEquals(Phases.PREPARATION, sut.phase);
    assertEquals(InitPhase.CONNECT_PATIENT, sut.initPhase);
    assertEquals(TherapyPhase.START_HEPARIN, sut.therapyPhase);
    ...
    // set monitored
    sut.interrupt_dialysis = false;
    ...
    // perform step
    sut.execDialysis();
    ...
    sut.execDialysis();
    // check conformance
    assertEquals(Phases.INITIATION, sut.phase);
    assertEquals(InitPhase.THERAPY_RUNNING, sut.initPhase);
    assertEquals(TherapyPhase.THERAPY_EXEC, sut.therapyPhase);
    ...
    // set monitored
    sut.interrupt_dialysis = true;
    // perform step
    sut.execDialysis();
}
```

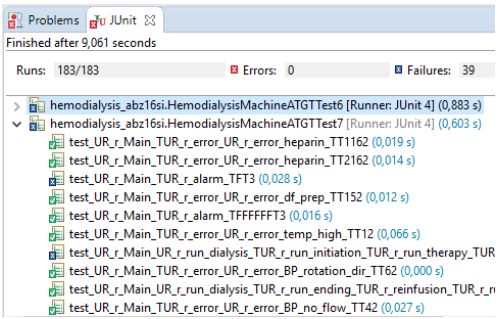
Code 6: JUnit test

In order to concretize the abstract test sequences into tests for the implementation, we need to provide a linking between the specification and the implementation. In [8], we proposed a technique to do the linking using Java annotations. We defined different annotations to:

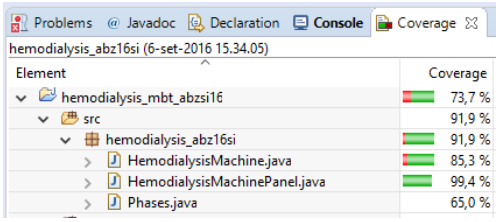
- associate a Java class with the corresponding ASM model (`@Asm`);
- associate the ASM state with the Java state:
  - `@FieldToFunction` connects a Java field with an ASM controlled function;
  - `@MethodToFunction` connects a Java pure (i.e., returning a value but not modifying the object state) method with an ASM controlled function;
  - `@Monitored` connects a Java field with an ASM monitored function; such fields represent the inputs of the Java class that take their value from the environment (as monitored functions in ASMs).
- associate the ASM behavior with the Java object behavior; `@RunStep` is used to annotate methods whose execution corresponds to a step of the ASM model.

Given the mapping provided by the Java annotations, we translated abstract test sequences in JUnit tests following the technique described in [9]. For example, Code 6 shows the JUnit test corresponding to the sequence shown in Fig. 11. Each test is built by creating the initialization of the Java class and then, for each state of the corresponding abstract test sequence,

- updating the fields annotated with `@Monitored` to the values of the corresponding ASM functions. In the example, field `interrupt_dialysis` is updated to the value of the homonymous ASM function.
- invoking the method annotated with `@RunStep`. In the example, method `execDialysis()` (see Code 5) is executed.



(a) JUnit failures



(b) Code coverage

Figure 12: JUnit testing results

- adding JUnit `assert` commands that check that the Java state is conformant with the ASM state; they check that the values of the fields annotated with `@FieldToFunction` and the values returned by the methods annotated with `@MethodToFunction` are equal to the values of the corresponding ASM functions. In the example, fields `phase`, `initPhase`, `therapyPhase`, ... are linked with `@FieldToFunction` and checked during conformance checking.

We run all the 183 tests (divided in multiple JUnit files) and we actually found some conformance violations (i.e., some tests failed. See Fig. 12a). We analyzed the failing tests and we discovered that the authors writing the implementation misunderstood some requirements. Consequently, the errors were fixed. The coverage obtained by the tests is shown in Fig. 12b: our tests were able to cover more than 90% of the Java code. Although the obtained coverage is already high, we found that some parts of the code are not covered, since the structure of the code in some parts is different from the structure of the specification and using structural coverage criteria for test generation may not guarantee the full coverage. We plan to study other criteria (like MCDC (Modified Condition Decision Coverage) or property-based) in order to improve the coverage.

6. Related work and comparison with other approaches

The use of rigorous methods is escalating for the engineering of medical device software in recent times. A systematic review of the use of formal methods for medical software is presented in [20]; we here report those works

that are more related to the approach presented in this paper.

In the past, formal methods have been applied to a variety of medical devices. Osaiweran et al. [54] use the formal Analytical Software Design (ASD) [23] approach for the development of a power control service of an interventional X-ray system. Jiang et al. [42] present a methodology based on timed automata to extract timing properties of heart that can be used for the verification and validation of implantable cardiac devices. Méry et al. [51] and Macedo et al. [47] present a pacemaker model in Event-B [1] and VDM [43] methods, respectively.

One of the medical devices relatively close to hemodialysis machines is the infusion pump. It is primarily responsible for delivering fluids, such as nutrients and medications, into a patient's body in controlled amounts. Arney et al. [16] present a reference model of PCA (Patient Control Analgesia) infusion pumps and test the model for structural and safety properties. Campos et al. [25] present a formal model in MAL (Modal Action Logic) [24] that helps compare different infusion devices and their provided functionalities. Bowen et al. [22] use the ProZ model checker [55] to test various safety properties of infusion pumps.

Considering the HMCS, Hoang et. al. [36] present an Event-B [1] inspired solution. The main difference of the approach is the usage of a multi-formal development paradigm where the requirements are modeled using the UML-like notation iUML-B [57] and then subsequently verified in the formal framework of Event-B using the deductive theorem proving and model checking. The approach also lets the specification be validated using animation and Domain Specific Visualizations (DSVs). A similar Event-B based solution is also presented in [49, 50]. In this work, the requirements are specified using a refinement-based modeling approach, and are then checked for consistency and conformance using the standard theorem proving, model checking, and animation techniques. The resulted formal requirements model is fed to a code generator that transforms the formal model into a sequential programming language code that runs on the given hardware. The translation process is semi-automatic and requires post-processing of the generated code before the final deployment. The main limitation of the approach is that the supported tool only translates a limited subset of the B syntax during the automatic translation process. Moreover, a formal proof that the translation process preserves the safety properties of the model is missing.

The solution presented by Banach [17] is based on Hybrid Event-B [18], an extension of the Event-B framework to explicitly focus on continuously

varying state evolution, along with the usual discrete state transitions. The main difference of this approach comes from its ability to explicitly distinguish between discrete and continuous elements of hemodialysis machines. The resulted specification consists of two types of state transitions: the natural discrete changes of state and continuously varying state changes. The model takes the individual discrete events of the model and interleaves them with continuous events. This allows to specify the complete behavior of hemodialysis machines considering both discrete as well as continuous elements. The drawback of the approach is the limited tool support. Hybrid Event-B relies on the Rodin platform [2] for specification and proving. However, not all features of Hybrid Event-B are supported by the Rodin platform, e.g., how to specify (and consequently prove) events capturing continuously changing behaviors (also known as pliant events), or single- and multi-machine systems in general.

The solution presented by Fayolle et. al. [29] is based on a combination of Algebraic State-Transition Diagrams (ASTD) [30] and Event-B. ASTD use a graphical notation to model problems as a combination of state transition diagrams and classical process algebra operators like sequence, iteration, parallel composition, quantified choice, and quantified synchronization. The main difference of this solution comes from a multi-formal approach and the way how the sequencing order of the machine is described. Due to the use of a graphical notation, the model is easy to follow and validate. For verification purposes, the model relies on the strength of the Event-B platform that stems from theorem proving and model checking. However, this means that the approach also suffers from the limitations associated with the Event-B method and its toolset. Some of these limitations are: lack of sophisticated tools and elaborated guidelines for managing the complexity of growing models by decomposition, lack of an implicit notion of time (this will be necessary for an elegant expression of timing properties which play a critical role in medical devices), and failure of ProB [45] (at the detailed level of refinements) to prove temporal properties of the system due to state space enumeration and explosion problems. In our opinion, a standard and more natural way is required to specify and prove that temporal properties of the system are preserved by Event-B refinements. Finally, a tool that is able to automatically generate ready-to-deploy machine code from Event-B formal models is also missing; currently available tools require manual post-processing of the generated code.

The solution presented by Gomes et. al. [34] is based on Circus [53], a

fusion of the formal notation Z [59] and Communicating Sequential Processes (CSP). The main difference of this solution model is the use of a well-defined theory of process algebra to specify the concurrent and parallel aspects of the system and explicit focus on timing properties. The main limitation of the approach is that no tool currently exists that directly supports the consistency and conformance checking of a Circus specification. In the current development, the Circus model is translated into a machine-readable CSP which is then model checked for verification purposes. The lack of automatic code generation from Circus models is another limitation.

Explicitly regarding the problem of software certification, there exist few attempts addressing this problem, like the CHI+MED project [27] which presents a formal methodology for certification and assurance of medical devices. More in general, Jetley et. al. [41] advocate the use of formal methods for medical software quality assessment. However, there is no standardized mapping between formal method processes and certification activities and this paper tries to contribute in establishing one.

The main novelty of our work, in relation to the comparative dialysis models and other applications of formal methods to medical device software, comes from its rigorous approach to quality assurance and easy to understand formal notation. A comprehensive model analysis approach based on simulation, model review, model checking, and conformance checking, gives a grasp on the notion of correctness far better than the approaches which are comprised of only a subset of the employed analysis techniques in this work. Additionally, ASM method's ease of use, understandability and notion of refinement also help manage the complexity of the development process. The limitation of the approach is that no tool is currently available that is capable of automatically transforming an ASM model into a programming language code. However, a tool translating ASM models to C++ for the Arduino platform has been developed [19].

7. Conclusions

Different certification standards have been proposed for the development of medical device software. However, these standards provide general indications regarding the typical software engineering activities that must be performed, but do not prescribe the use of any particular method, technique, or life cycle model. In the paper, we have shown how a development process based on the Abstract State Machine formal method can be used for

assuring safety and reliability in the development of medical device software. The process consists in an iterative and incremental life cycle model based on model refinement: through a sequence of refined models, all the requirements of the system are considered. Different validation and verification (visualization, simulation, model review, model checking) activities can be performed on each model, and each refinement step can be automatically proved correct. The implementation can be seen as the last refinement step, and its conformance with the formal specification can be checked by model-based testing and/or by runtime verification. We have described how the ASM-based process captures most of the activities required by the standards, and have shown the application of the ASM-based process to the hemodialysis machine case study.

Acknowledgments

The research reported in this paper has been partially supported by the Czech Science Foundation project number 17-12465S, 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.

References

[1] Abrial, J.-R., 2010. Modeling in Event-B: System and Software Engineering. Cambridge University Press.

[2] Abrial, J.-R., Butler, M., Hallerstede, S., Hoang, T., Mehta, F., Voisin, L., 2010. Rodin: an open toolset for modelling and reasoning in Event-B. International Journal on Software Tools for Technology Transfer 12 (6), 447–466.

[3] Arcaini, P., Bonfanti, S., Gargantini, A., Mashkoor, A., Riccobene, E., Sept 2015. Formal validation and verification of a medical software critical component. In: Formal Methods and Models for Codesign (MEMOCODE), 2015 ACM/IEEE International Conference on. IEEE, pp. 80–89.

[4] Arcaini, P., Bonfanti, S., Gargantini, A., Riccobene, E., 2016. How to assure correctness and safety of medical software: the hemodialysis machine case study. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro,

M. (Eds.), Abstract State Machines, Alloy, B, TLA, VDM, and Z. 5th International Conference, ABZ 2016, Linz, Austria, May 23-27, 2016, Proceedings. Vol. 9675 of Lecture Notes in Computer Science. Springer International Publishing, pp. 344–359.

[5] Arcaini, P., Bonfanti, S., Gargantini, A., Riccobene, E., 2016. Visual notation and patterns for abstract state machines. In: Milazzo, P., Varró, D., Wimmer, M. (Eds.), Software Technologies: Applications and Foundations: STAF 2016 Collocated Workshops: DataMod, GCM, HOFM, MELO, SEMS, VeryComp, Vienna Austria, July 4-8, 2016, Revised Selected Papers. Springer International Publishing, Cham, pp. 163–178.

[6] Arcaini, P., Gargantini, A., Riccobene, E., 2010. AsmetaSMV: A way to link high-level ASM models to low-level NuSMV specifications. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (Eds.), Abstract State Machines, Alloy, B and Z. Vol. 5977 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 61–74.

[7] Arcaini, P., Gargantini, A., Riccobene, E., 2010. Automatic Review of Abstract State Machines by Meta Property Verification. In: Muñoz, C. (Ed.), Proceedings of the Second NASA Formal Methods Symposium (NFM 2010). NASA, pp. 4–13.

[8] Arcaini, P., Gargantini, A., Riccobene, E., 2012. CoMA: Conformance monitoring of Java programs by Abstract State Machines. In: Khurshid, S., Sen, K. (Eds.), Runtime Verification. Vol. 7186 of Lecture Notes in Computer Science. Springer, pp. 223–238.

[9] Arcaini, P., Gargantini, A., Riccobene, E., 2013. Combining model-based testing and runtime monitoring for program testing in the presence of nondeterminism. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, Workshops Proceedings, Luxembourg, March 18-22, 2013. IEEE, pp. 178–187.

[10] Arcaini, P., Gargantini, A., Riccobene, E., 2014. Offline Model-Based Testing and Runtime Monitoring of the Sensor Voting Module. In: Boniol, F., Wiels, V., Ait Ameur, Y., Schewe, K.-D. (Eds.), ABZ 2014: The Landing Gear Case Study. Vol. 433 of Communications in Computer and Information Science. Springer International Publishing, pp. 95–109.

[11] Arcaini, P., Gargantini, A., Riccobene, E., 2016. SMT-based automatic proof of ASM model refinement. In: De Nicola, R., Kühn, E. (Eds.), 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.

[12] Arcaini, P., Gargantini, A., Riccobene, E., 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), 247–269.

[13] Arcaini, P., Gargantini, A., Riccobene, E., Scandurra, P., 2011. A model-driven process for engineering a toolset for a formal method. Software: Practice and Experience 41, 155–166.

[14] Arcaini, P., Holom, R.-M., Riccobene, E., 2016. ASM-based formal design of an adaptivity component for a cloud system. Formal Aspects of Computing 28 (4), 567–595.

[15] Arcaini, P., Riccobene, E., Gargantini, A., 2016. Model-based offline and online testing for medical software. In: Industrial proceedings of the 23rd EuroAsiaSPI^2 Conference. WHITEBOX, Denmark, pp. 11.11–11.20.

[16] Arney, D., Jetley, R., Jones, P., Lee, I., Sokolsky, O., 2007. Formal Methods Based Development of a PCA Infusion Pump Reference Model: Generic Infusion Pump (GIP) Project. In: Proceedings of the 2007 Joint Workshop on High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability. HCMDSS-MDPNP '07. IEEE Computer Society, Washington, DC, USA, pp. 23–33.

[17] Banach, R., 2016. Hemodialysis machine in hybrid event-b. In: 5th International Conference on ASM, Alloy, B, TLA, VDM, and Z (ABZ'16). Vol. 9675 of Lecture Notes in Computer Science. Springer, pp. 376–393.

[18] Banach, R., Butler, M., Qin, S., Verma, N., Zhu, H., 2015. Core Hybrid Event-B I: Single Hybrid Event-B Machines. Science of Computer Programming 105, 92 – 123.

[19] Bonfanti, S., Carissoni, M., Gargantini, A., Mashkoor, A., 2017. Asm2C++: A Tool for Code Generation from Abstract State Machines

to Arduino. In: Barrett, C., Davies, M., Kahsai, T. (Eds.), NASA Formal Methods: 9th International Symposium, NFM 2017, Proceedings. Springer, pp. 295–301.

[20] Bonfanti, S., Gargantini, A., Mashkoor, A., 2017. A systematic literature review of the use of formal methods in medical software systems. Journal of Software: Evolution and Process submitted.

[21] Börger, E., Stärk, R., 2003. Abstract State Machines: A Method for High-Level System Design and Analysis. Springer Verlag.

[22] Bowen, J., Reeves, S., 2013. Modelling safety properties of interactive medical systems. In: Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems. EICS'13. ACM, New York, NY, USA, pp. 91–100.

[23] Broadfoot, G. H., 2005. ASD Case Notes: Costs and Benefits of Applying Formal Methods to Industrial Control Software. In: Fitzgerald, J., Hayes, I. J., Tarlecki, A. (Eds.), FM 2005: Formal Methods. Vol. 3582 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 548–551.

[24] Campos, J. C., Harrison, M. D., 2008. Systematic analysis of control panel interfaces using formal tools. In: Graham, T., Palanque, P. (Eds.), Interactive Systems. Design, Specification, and Verification. Vol. 5136 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 72–85.

[25] Campos, J. C., Harrison, M. D., 2011. Modelling and analysing the interactive behaviour of an infusion pump. ECEASST 45.

[26] Carioni, A., Gargantini, A., Riccobene, E., Scandurra, P., 2008. A scenario-based validation language for ASMs. In: Börger, E., Butler, M., Bowen, J. P., Boca, P. (Eds.), Abstract State Machines, B and Z: First International Conference, ABZ 2008, London, UK, September 16-18, 2008. Proceedings. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 71–84.

[27] Curzon, P., Masci, P., Oladimeji, P., Ruknas, R., Thimbleby, H., D'Urso, E., 2014. Human-Computer Interaction and the Formal Certification

and Assurance of Medical Devices: The CHI+ MED Project. EPSRC Programme Grants.

[28] EU, September 2007. Directive 2007/47/EC of the European Parliament and of the Council. Official Journal of the European Union.
URL http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=OJ:L:2007:247:0021:0055:en:PDF

[29] Fayolle, T., Frappier, M., Gervais, F., Laleau, R., 2016. Modelling a Hemodialysis Machine using Algebraic State-Transition Diagrams and B-like Methods. In: 5th International Conference on ASM, Alloy, B, TLA, VDM, and Z (ABZ'16). Vol. 9675 of Lecture Notes in Computer Science. Springer, pp. 394–408.

[30] Frappier, M., Gervais, F., Laleau, R., Fraikin, B., 2008. Algebraic State Transition Diagrams. Université de Sherbrooke, Département d'informatique, Sherbrooke, Québec, Canada, Rapport technique 24.

[31] Gargantini, A., Riccobene, E., 2001. ASM-Based Testing: Coverage Criteria and Automatic Test Sequence Generation. Journal of Universal Computer Science 7, 262–265.

[32] Gargantini, A., Riccobene, E., Rinzivillo, S., 2003. Using Spin to Generate Tests from ASM Specifications. In: Börger, E., Gargantini, A., Riccobene, E. (Eds.), Abstract State Machines 2003. Vol. 2589 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 263–277.

[33] Gargantini, A., Riccobene, E., Scandurra, P., 2008. A Metamodel-based Language and a Simulation Engine for Abstract State Machines. J. UCS 14 (12), 1949–1983.

[34] Gomes, A. O., Butterfield, A., 2016. Modelling the Haemodialysis Machine with Circus. In: 5th International Conference on ASM, Alloy, B, TLA, VDM, and Z (ABZ'16). Vol. 9675 of Lecture Notes in Computer Science. Springer, pp. 409–424.

[35] Hierons, R., Derrick, J., 2000. Editorial: special issue on specification-based testing. Software Testing, Verification and Reliability 10 (4), 201–202.

[36] Hoang, T. S., Snook, C., Ladenberger, L., Butler, M., 2016. Validating the Requirements and Design of a Hemodialysis Machine Using iUML-B, BMotion Studio, and Co-simulation. In: 5th International Conference on ASM, Alloy, B, TLA, VDM, and Z (ABZ'16). Vol. 9675 of Lecture Notes in Computer Science. Springer, pp. 360–375.

[37] IEC, 2005. IEC 60601-1:2005 medical electrical equipment part 1: General requirements for basic safety and essential performance.

[38] IEC, 2006. IEC 62304 - medical device software - software lifecycle processes.

[39] ISO, 2003. ISO 13485:2003 medical devices – quality management systems – requirements for regulatory purposes.

[40] ISO, 2007. ISO 14971:2007 medical devices – application of risk management to medical devices.

[41] Jetley, R., Iyer, Purushothaman, S., Jones, P. L., Apr. 2006. A formal methods approach to medical device review. Computer 39 (4), 61–67.

[42] Jiang, Z., Pajic, M., Connolly, A., Dixit, S., Mangharam, R., 2010. A platform for implantable medical device validation: Demo abstract. In: Wireless Health 2010. WH '10. ACM, New York, NY, USA, pp. 208–209.

[43] Jones, C. B., 1990. Systematic software development using VDM (2nd ed.). Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[44] Jones, P., Jetley, R., Abraham, J., 2010. Formal methods-based verification of medical device software analysis. Electronic Engineering Times 1579, 31–32.

[45] Leuschel, M., Butler, M., 2008. ProB: An Automated Analysis Toolset for the B Method. Journal Software Tools for Technology Transfer 10 (2), 185–203.

[46] Leveson, N. G., Turner, C. S., Jul. 1993. An investigation of the Therac-25 accidents. Computer 26 (7), 18–41.

[47] Macedo, H., Larsen, P., Fitzgerald, J., 2008. Incremental Development of a Distributed Real-Time Model of a Cardiac Pacing System Using

VDM. In: Cuellar, J., Maibaum, T., Sere, K. (Eds.), FM 2008: Formal Methods. Vol. 5014 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 181–197.

[48] Mashkoor, A., 2016. The hemodialysis machine case study. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (Eds.), Abstract State Machines, Alloy, B, TLA, VDM, and Z: 5th International Conference, ABZ 2016, Proc. Springer, pp. 329–343.

[49] Mashkoor, A., 2016. Model-driven development of high-assurance active medical devices. Software Quality Journal 24 (3), 571–596.

[50] Mashkoor, A., Biro, M., 2016. Towards the trustworthy development of active medical devices: A hemodialysis case study. IEEE Embedded Systems Letters 8 (1), 14–17.

[51] Méry, D., Singh, N. K., Jan. 2013. Formal specification of medical systems by proof-based refinement. ACM Trans. Embed. Comput. Syst. 12 (1), 15:1–15:25.

[52] nuXmv, 2017. The nuXmv website. https://nuxmv.fbk.eu/.

[53] Oliveira, M., Cavalcanti, A., Woodcock, J., 2007. A UTP semantics for Circus. Formal Aspects of Computing 21 (1), 3–32.

[54] Osaiweran, A., Schuts, M., Hooman, J., Wesselius, J., May 2013. Incorporating formal techniques into industrial practice: An experience report. Electron. Notes Theor. Comput. Sci. 295, 49–63.

[55] Plagge, D., Leuschel, M., 2007. Validating Z Specifications Using the ProB Animator and Model Checker. In: Davies, J., Gibbons, J. (Eds.), Integrated Formal Methods. Vol. 4591 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 480–500.

[56] Slissenko, A., Vasilyev, P., 2008. Simulation of Timed Abstract State Machines with predicate logic model-checking. Journal of Universal Computer Science 14 (12), 1984–2006.

[57] Snook, C., Butler, M., Jan. 2006. UML-B: Formal modeling and design aided by UML. ACM Trans. Softw. Eng. Methodol. 15 (1), 92–122.

[58] Sommerville, I., 2010. Software Engineering, 9th Edition. Addison-Wesley Publishing Company, USA.

[59] Spivey, J. M., 1988. Understanding Z: a specification language and its formal semantics. Cambridge University Press.

[60] U.S. Food and Drug Administration (FDA), Jan. 2002. General principles of software validation; final guidance for industry and FDA staff, version 2.0.
URL http://www.fda.gov/downloads/MedicalDevices/DeviceRegulationandGuidance/GuidanceDocuments/ucm085371.pdf

[61] Utting, M., Legeard, B., 2006. Practical Model-Based Testing: A Tools Approach. Morgan-Kaufmann.