Formal Validation and Verification of a Medical Software Critical Component

Paolo Arcaini*, Silvia Bonfanti†, Angelo Gargantini†, Atif Mashkoor‡ and Elvinia Riccobene§
*Charles University in Prague, Faculty of Mathematics and Physics, Czech Republic – Email: arcaini@d3s.mff.cuni.cz
†University of Bergamo, Italy – Email: {silvia.bonfanti,angelo.gargantini}@unibg.it
‡Software Competence Center Hagenberg, Austria – Email: atif.mashkoor@scch.at
§Dipartimento di Informatica, Università degli Studi di Milano, Italy – Email: elvinia.riccobene@unimi.it

***Abstract*—Medical device software malfunctioning can lead to injuries or death for humans and, therefore, its development should adhere to certification standards. However, these standards 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 presents a formal development process, based on the Abstract State Machine method, that integrates most of the activities required by the standards. The process permits to obtain, through a sequence of refinements, more detailed models that can be formally validated and verified. Offline and online testing techniques permit to check the conformance of the implementation w.r.t. the specification. The process is applied to the validation of the SAM medical software, that is used to measure the patients' stereoacuity in the diagnosis of amblyopia.

***Index Terms*—medical software, certification standards, Verification & Validation, Abstract State Machines**

I. Introduction

In medical devices, human safety depends upon the correct operation of the software controlling the device: software malfunctioning can cause injuries to, or even the death of, patients. Therefore, medical software certification is a crucial issue. Medical software requires validation and verification to ensure its correct behavior, and its development should adhere to some guidelines prescribed by standards.

Currently, the most relevant standards of medical devices, as ISO 13485 [1], ISO 14971 [2], IEC 60601-1 [3] and the EU Directive 2007/47/EC [4], mainly focus on physical aspects and electrical components rather than on software. The main reference concerning development of medical device software is the standard *IEC 62304* [5] (International Electrotechnical Commission) which classifies medical software in three classes on the basis of the potential injuries caused, and defines the required software documentation of appropriate life cycle activities. Besides standards, the FDA (Food and Drug Administration) has established *General Principles of Software Validation* [6] applicable to medical software. It defines several broad concepts that can be used as guidance for software validation and verification, although no specific life cycle model or specific technique or method is recommended. It mainly requires that software validation and verification are conducted throughout the entire software life cycle.

The research reported in this article has been partially supported by Charles University research funds PRVOUK, and 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.

Both IEC standard and FDA principles aim for more rigorous approaches, based on the use of formal methods, to assure safety and reliability of software for medical devices [7]. 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 device software to behavioral models specifying safe device operation (since, most of the time, software for medical devices is not developed from scratch). Furthermore, it must be endowed with a set of tools for modeling and analysis.

In the wide range of existing formal methods, the Abstract State Machines [8] is a system engineering method able to guide the development of software and embedded systems seamlessly from requirements capture to their implementation. Within a precise but simple conceptual framework, the ASM method allows a modeling technique which integrates dynamic (operational) and static (declarative) descriptions, and an analysis technique that combines validation (by simulation and testing) and verification methods at any desired level of detail. ASMs are an extension of Finite State Machines. The method has, therefore, a rigorous mathematical foundation, but a practitioner needs no special training to use the method since ASMs can be correctly understood as pseudo-code or virtual machines working over abstract data structures.

ASMs allow a design process based on the concept of a *ground model* representing a precise but concise high-level specification of the system, and on the *refinement principle* that allows to capture all details of the system design by a sequence of refined models till the desired level of detail. Validation and verification (V&V) are fully integrated into the ASM design process, and, at any abstract level, a series of tools can be used for different forms of analysis both at model level w.r.t. the requirements, and at code level w.r.t. the models.

By exploiting the advantages that ASMs offer, both in terms of software life cycle activities and in terms of developing methods and tools, in this paper we present an FDA-compliant software development process. It inherits the mathematical rigor and the variety of techniques and tools of the ASMs, and it is compliant with the principles and the concepts required by the FDA principles, especially those regarding the integration of V&V activities. The main contribution of our work is to define a process for rigorous development of software for medical devices where (i) models are a mathematical-base for safety properties assurance; (ii) V&V activities can be planned, performed continuously along the software life cycle, and always aimed at defect prevention; (iii) software quality evaluation can be carried out in an objective and repeatable manner, i.e., by "independence of review" [6]; and (iv) a device manufacturer can always demonstrate that software has been validated and verified.

We show the results of the application of the ASM-based FDA-compliant V&V process to the SAM (Stereo Acuity Measurer) software. This is a component of a medical application, called 3DSAT, developed within the 3D4amb project[1] for the diagnosis of *amblyopia*, a disorder of sight also called *lazy eye*. The SAM component is used to measure young patients' stereoacuity, and belongs to the class B (non-serious injury is possible) on the basis of the *IEC 62304* classification. A local Italian hospital is testing the effectiveness of the 3DSAT software w.r.t. standard opthalmologic methods. The ongoing work is to formally validate and verify the tool in order to avoid wrong diagnoses and possible injuries to patients. The long term goal of this research is to establish rigorous evaluation methods and tools to assure safe and reliable medical software operation.

The paper is organized as follows. Sect. II briefly presents standards available for the certification of medical software. After an introduction to the ASMs in Sect. III, Sect. IV describes an ASM-based formal development method that has the aim to comply with the related standards. Sect. V presents a case study, and Sect. VI and VII show how the ASM-based method has been applied to the case study. Sect. VIII presents some related work, and Sect. IX concludes the paper.

II. Medical software standards and guidelines

Here we very briefly review the main concepts of standards and guidelines on medical software development, skipping those more related to physical aspects of medical devices.

The standard IEC 62304 defines safety classes of the software. The classification is based on the potential to cause an injury to a patient in case of software malfunction:

- Class A: No injury or damage to health is possible
- Class B: Non-serious injury is possible
- Class C: Death or serious injury is possible

IEC 62304 defines the software documentation for each class as shown in Table I. Class A does not require testing and verification since this class of medical device software is not critical. These activities are required for software of classes B and C, since software malfunctioning may cause injuries. Class C also requires a detailed design.

Software documentation	**Class**		
	A	**B**	**C**
Software development plan	X	X	X
Software requirements specification	X	X	X
Software architecture		X	X
Software detailed design			X
Software unit implementation	X	X	X
Software unit verification		X	X
Software integration and integration testing		X	X
Software system testing		X	X
Software release	X	X	X

Table I
IEC 62304 SOFTWARE DOCUMENTATION

FDA recommends an integration of software life cycle management and risk management activities [6]. The FDA guidelines list some general principles that should be considered for software V&V process: 1) A documented *software requirements specification* to be used as baseline for V&V; 2) A continuous attention on *defect prevention*; 3) Preparation for software V&V early and conducted throughout the software life cycle; 4) Software V&V considered within the *software life cycle model*; 5) *Software V&V plan* to guide the activities; 6) *Procedures* for V&V activities, tasks and work items identifying the specific actions to be taken; 7) Software V&V upon any (software) *change*; 8) Validation coverage based on software complexity and safety risk; 9) Independence of review to guarantee the software quality; 10) Device manufacturer has flexibility in choosing V&V principles, but retains ultimate responsibility for demonstrating that the software has been validated and verified.

[1]http://3d4amb.unibg.it/

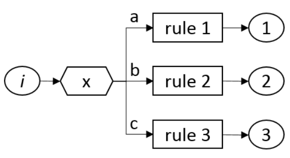
III. Abstract State Machines and the Method

Abstract State Machines (ASMs) are an extension of Finite State Machine where unstructured control states are replaced by states with arbitrary complex data. A *state* represents the instantaneous configuration of the system under development, and *transition rules* describe the change of state. ASM *states* are multi-sorted first-order structures, i.e., domains of objects with functions and predicates defined on them. ASM *transition rules* express how function interpretations are modified from one state to the next one, and therefore describe the system configuration changes. The basic form of a transition rule is the *guarded update*: "**if** *Condition* **then** *Updates*", where *Updates* is a set of function updates of the form $f(t_1, \ldots, t_n) := t$ which are simultaneously executed when *Condition* is true; f is an arbitrary n-ary function and $t_1, \ldots, t_n, t$ are first-order terms.

An ASM state is represented by a set of couples (*location*, *value*). ASM *locations* represent the abstract ASM concept of basic object containers (memory units). Location *updates* represent the basic units of state change.

Besides **if-then**, there is a limited but powerful set of *rule constructors*: **par** for simultaneous parallel actions, **seq** for sequential actions, **choose** for nondeterminism (existential quantification), **forall** for unrestricted synchronous parallelism (universal quantification).

Functions that never change during any run of the machine are *static*. Those updated by agent actions are *dynamic*, and distinguished between *monitored* (only read by the machine



```
if ctl_state = i then
  if x = a then
    rule 1
    ctl_state := 1
  if x = b then
    rule 2
    ctl_state := 2
  if x = c then
    rule 3
    ctl_state := 3
```

Figure 1. Control state ASMs – Alternative definition

and modified by the environment), and *controlled* (read and written by the machine).

A *computation* of an ASM is a finite or infinite sequence $s_0, s_1, \ldots, s_n, \ldots$ of states of the machine, where s_0 is an initial state and each s_{n+1} is obtained from s_n by simultaneously firing all the transition rules which are enabled in s_n. The (unique) *main rule* is a transition rule and represents the starting point of the computation. An ASM can have more than one *initial state*. It is possible to specify state *invariants*.

When modeling control systems, as it is often the case for medical device software, it could be useful to use a particular class of ASMs, called *control state ASMs* [8], suitable to explicitly model system *modes* (or *control states*). Control state ASMs have an intuitive graphical representation by means of control state diagrams. To express a decision point depending on the value of an enumerative function (even boolean), we introduce here a variant of control state diagram in [8]. It is shown in Fig. 1, together with its corresponding mathematical model. Each value taken by x (specified as a label on the arrow) corresponds to a different statement. An arrow without label is to be understood as *x is different from the values reported on all the other arrows exiting from the guard* (at most one unlabelled arrow can exit from a guard).

For system specification, the ASM method builds upon two further concepts [8]:

- *ground model*, an ASM which is a first reference model for the design;
- *model refinement*, a general scheme for stepwise instantiations of model abstractions to concrete system elements, providing controllable links between the more and more detailed descriptions at the successive stages of system development.

The modelling activity is supported by a number of V&V activities on models, already applicable at the ground level and along the chain of refined models, that help to guarantee correctness of the developed system.

IV. ASM-BASED FDA-COMPLIANT V&V PROCESS

A. ASM-based V&V process

A rigorous process for ASM-based development [9], based on the concepts of ground model and model refinement, is here presented for medical device software. The process is depicted in Fig. 2: the modelling activity is complemented with a number of other activities on models and eventually on code. All these activities help the modeler to develop a correct system

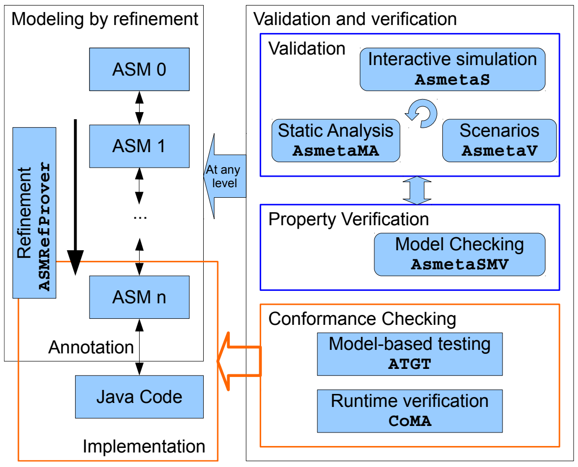


Figure 2. ASM-based development process

in a correct way. A set of tools exists to support the developer in the various activities and to make the ASM method useful in practice. The process is not automatic, but it is automatable if an order among the activities is imposed. Tools are part of the ASMETA (ASM mETAmodeling) framework[2] [10], and are strongly integrated in order to permit reusing information about models during different development phases. The IDE `AsmEE` is available to assist the user when editing an ASM model by using the concrete syntax `AsmetaL` [11].

An abstract *ground model* (ASM0 in Fig. 2) is specified using terms of the application domain by reasoning on the informal requirements (generally given as a text in natural language), possibly with the involvement of all stakeholders. The ground model should be *correct*, i.e., it reflects the intended initial requirements, and *consistent*, i.e., it removes ambiguities of the initial textual requirements. However, it does not need to be *complete*, i.e., it may leave some given functional requirements unspecified.

From the ground model, by step-wise refined models, further details are added to capture all the functional requirements and provide descriptions of the complete software architecture and component design of the system. In this way, the complexity of the system can be always taken under control, and it is possible to bridge, in a seamless manner, the gap between specification and code. Each time a model is specified as a refinement of an abstract one, refinement correctness should be checked. This can be done by hand, but we provide an automatic way to achieve this assurance in case of *stuttering refinement*, a restricted form of ASM refinement. The tool `ASMRefProver` automatically checks stuttering refinement between two ASM models (see Sect. VII-A).

Modelling activity is supported, at each level of refinement, by model *validation* and *verification* (V&V). Model validation should be applied, already at ground model level, in order to ensure that the specification really reflects the user needs and statements about the system, and to detect faults in the

[2]http://asmeta.sourceforge.net/

specification as early as possible with limited effort. ASM model validation is possible by means of the model simulator AsmetaS [11] (see Sect. VII-B) and by the validator AsmetaV [12] (see Sect. VII-C) that allows to build and execute *scenarios* of expected system behaviours. 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 [13] (see Sect. VII-D).

Validation usually precedes the application of more expensive and accurate methods, like formal requirements analysis and verification of properties, that should be applied only when a designer has enough confidence that the specification captures all informal requirements. Formal verification of ASMs is possible by means of the model checker AsmetaSMV [14] (see Sect. VII-E). Both *Computation Tree Logic* (CTL) and *Linear Temporal Logic* (LTL) formulas can be proved.

When an actual code of the system implementation is available, either derived from the model as last low-level refinement step, or externally provided, also *conformance checking* (see Sect. VII-G) is possible. Both *model-based testing* and *runtime verification* can be applied to check if the implementation conforms to its specification. We support conformance checking w.r.t. Java code. The tool ATGT [15] can be used to automatically generate tests from ASM models[3] and, therefore, to check the conformance *offline*; CoMA [16], instead, can be used to perform runtime verification, i.e., to check the conformance *online*.

In Sect. V, we present the application of the proposed process to the SAM case study.

B. How ASM V&V process achieves FDA principles

The proposed process realizes the FDA guideline principles as explained in the following.

Requirements specification includes the identification, analysis, and documentation of information about the device and its intended use. The ASM development process, based on refinement, clearly specifies and documents the requirements as a chain of ASM models that provide a *rigorous baseline for both validation and verification.*

The modeling process permits a continuous *defect prevention*. Indeed, along the chain of refinements proved correct, models preserve the desired properties, starting from a very abstract system description to the actual implementation. Safety properties are formally proved on models. However, this is not enough: would you accept to use a medical software whose model is proved correct but it has never been tested? Our methodology includes software testing for conformance verification of the implementation.

The proposed methodology allows *preparation for software validation and verification* as early as possible, since one can start validation and verification already at the very abstract level of the ground model. The V&V process can be *planned* at different abstract levels, and every activity is clearly documented in the different phases and supported by precise *procedures*.

When a change of the software occurs, in case the change does not involve the model (i.e., it is a local change in the implementation), our methodology requires to rerun only the conformance checking. Instead, if the change affects the ASM specification at a certain level i, it requires to re-prove the refinement correctness w.r.t. the more abstract model $i-1$ (if any) and the more concrete model $i+1$ (if any), and the re-execution of the V&V activities for the models only from level i down to the implementation.

Regarding *validation coverage*, during simulation (user-guided or scenario-based) and during 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. We plan to work on techniques for measuring the coverage of formal property verification like advocated also in [17].

Since validation and verification are performed by exploiting mathematical-based techniques, they facilitate *independent evaluation* of software quality assurance.

Finally, the ASM-based development process always allows a *device manufacturer to demonstrate that the software has been validated and verified*, both when it is a software implementation obtained as the last step of ASM model correct refinement, and when it is external code that has been conformance checked.

V. Case study – SAM

Within the project 3D4amb, we have developed a family of applications usable by optometrists and ophthalmologists to detect visual problems. One of these applications, 3DSAT [18], is currently used in an Italian hospital to measure the stereoacuity of young patients and to detect amblyopia. Based on the IEC 62304 standard, the software belongs to class B (non-serious injury is possible), because a patient could have some future injuries if the doctor gives her/him a wrong treatment. The core component of the 3DSAT application is SAM (Stereoscopic Acuity Measurer), designed to decide the stereo depth of the image to be shown to the patient, when to stop the test, and to provide the final exam result (stereoacuity certification). We are interested in validating and verifying SAM following the FDA principles.

Informal Requirements

To certify the patient's stereoacuity, different stereo random dot images are shown in a 3D stereo monitor at different (6) levels of difficulties. The test starts at level 6, that is the easiest level. Every time a patient recognizes the shown image, the *level decreases* (i.e., the test becomes more difficult) until it reaches level 1, which is the most difficult level. A level is passed if the patient recognizes three times the shown images. When the patient answers incorrectly, (s)he can try another time at the same level. If (s)he fails again, the *level increases*

[3]Note that sequences generated by ATGT could be used to test programs written in any programming language.

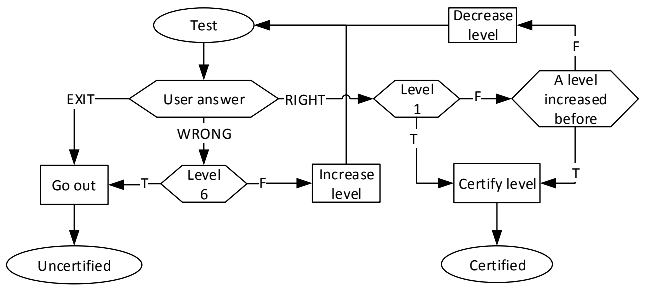


Figure 3. Control state ASM ground model

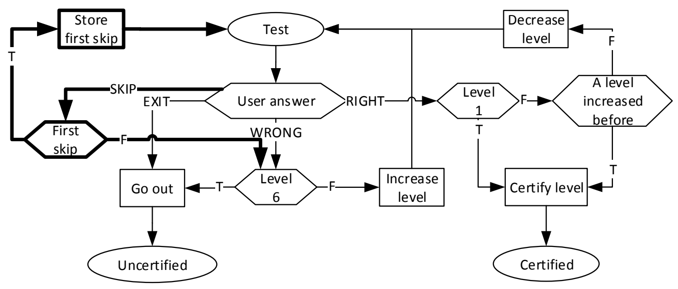


Figure 4. Control state ASM first refinement

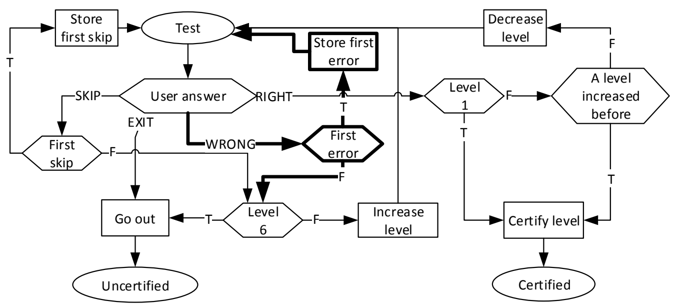


Figure 5. Control state ASM second refinement

(i.e., the test becomes easier), and (s)he cannot be tested at that level anymore: (s)he can try to be certified at the upper level. If the patient fails twice the 6th level, the test stops with no certified level. Besides recognizing images, a user can EXIT the test or SKIP an image. The SKIP answer is treated like the wrong answer: if the patient skips twice at the same level, the level increases.

The SAM component eventually stops: 1) by certifying the patient at level i, if the patient has been able to recognize three images at level i, but has failed at level $i-1$ (if any); 2) without certification, if the patient has been unable to complete the test or the doctor has quit the exam.

VI. Requirement capture by modeling and refinement

The SAM component has been modeled using five levels of refinement. The starting ground model captures the core behavior of the component; the further requirements have been considered incrementally along the subsequent refined models till a level detailed enough to be checked for conformance w.r.t. the code[4].

1) Ground model: In the ground model, we abstract from the explicit answers of the patient, and we consider only if (s)he successfully recognizes an image (regardless of the specific image). We, therefore, model the reaction of the software component to a user input which can be a correct/incorrect patient's answer or the doctor request to quit the system without any certification. Therefore, in the ground model, the patient is certified at a given level if (s)he recognizes the image once. Fig. 3 shows the corresponding ASM control state.

The system can be in three different configurations: in state *Test* when the patient is doing the test, in (the final) state *Uncertified* when the patient is not able to complete the test and the system is not able to certify the patient, in (the final) state *Certified* when the patient finishes the test at a given level and can be certified at that level.

In state *Test* (also initial), the system checks for the user answer. If the doctor wants to EXIT, the test is stopped (Go out) and the system goes in *Uncertified* state. Otherwise, the system checks if the patient's answer is correct or not.

If the answer is RIGHT and the patient is in level 1, (s)he has completed the test. The system provides the corresponding certification (Certify level) and moves to *Certified* state. Otherwise, the software checks if the patient has previously failed a level (i.e., a level has been increased before). If this is the case, the software certifies the patient at the current level and moves to *Certified* state. Otherwise, the level is decreased (Decrease level) and the system returns to *Test* state.

If the answer is WRONG and the patient is doing the test at level 6, the system stops the test and moves to *Uncertified* state. Otherwise, the level testing is increased (Increase level) and the system returns to *Test* state.

2) First refinement: At this level of refinement, we add the SKIP option. At each level the patient can skip the answer once. If (s)he skips the answer twice at the same level, the level is increased. Fig. 4 shows the control state ASM of the first refinement, where the new part is marked in bold. If the user inserts SKIP for the first time, the system memorizes that a SKIP has been made and returns to *Test* state. In case it is the second skip at the same level, the system behaves as in presence of a WRONG answer in the previous model.

3) Second refinement: At this level of refinement, we model the requirement that the patient can give a wrong answer only once; thus, on the second mistake at the same level, the level is increased. Fig. 5 shows the control state ASM of the second refinement. The added part of the model is marked in bold. When the patient gives the WRONG answer for the first time at the same level, the system memorizes that an error has been made and returns to *Test* state. Otherwise, if it is the second error at the same level, the system behaves as in presence of

[4]Complete models are available at http://fmse.di.unimi.it/sw/MEMOCODE15.zip. They are written in `AsmetaL` [11], a concrete syntax for ASM models. An `AsmetaL` model is structured into three sections: a *header* where the signature is declared and external models (i.e., ASM without the main rule) are imported, a *body* in which functions, domains, invariants, and rules (including the *main rule*) are defined, and an *init* which initializes the machine.

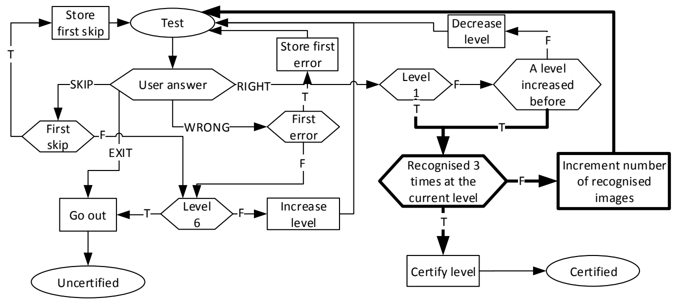


Figure 6. Control state ASM third refinement

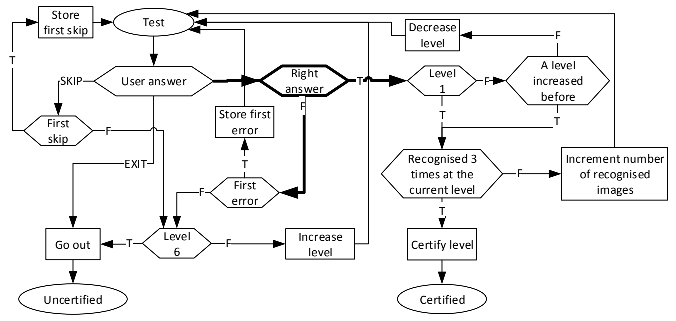


Figure 7. Control state ASM fourth refinement

a WRONG answer in the previous model.

4) Third refinement: At this level of refinement, we specify the requirement that the patient has to recognize the shown image three times at the same level to get certification at that level. Fig. 6 shows the control state ASM of the third refinement. The new part is shown in bold. When the patient gives the RIGHT answer, the system checks for the current level under test. If it is level 1 and the patient has already answered correctly twice (i.e., the current answer is the third correct image recognition), the system issues her/him a certification and moves to the *Certified* state. Otherwise, the number of images recognized at that level is incremented and the patient continues the test. The same behavior is performed when the patient is not in level 1 but (s)he has already incremented the level (i.e., (s)he made two skips or two errors in a lower level).

5) Fourth refinement: This model refines RIGHT and WRONG answers with the images name. Images are randomly chosen and the system checks whether the patient correctly recognizes the shown image or not. Fig. 7 reports the control state ASM of the fourth refinement, where the new part is shown in bold. When the patient answers SKIP or the doctor selects the EXIT answer, the behavior is the same as in the previous refinements. In case the answer is an allowed image name, the system checks whether it is the right one. If the answer is correct, the system behaves as previously modeled in case of RIGHT answer, otherwise it has the same behavior as in case of a WRONG answer (see previous refinements).

VII. V&V OF THE SAM

In this section we show the application to the case study of the V&V activities presented in Sect. IV.

A. Proving refinement

Each step of refinement is proved correct [8] using the SMT-based tool `ASMRefProver`[5]. We check a notion of refinement, called *stuttering refinement*, more restrictive than the definition given in [8]. It comes with a notion of *locations of interest*, namely those state locations one wants to relate in corresponding abstract and refined states, and a notion of *conformance relation* between abstract and refined states having equivalent values of locations of interest. Given a run $\widetilde{\rho}$ of the refined model, there must exist a run ρ of the abstract model such that *(i)* initial states of both runs are conformant, and *(ii)* each refined state in $\widetilde{\rho}$ is conformant with an abstract state in ρ: if a refined state $\tilde{s}$ in $\widetilde{\rho}$ is conformant with an abstract state s in ρ, then its successor state s' in $\widetilde{\rho}$ is conformant with either s or with s', the next state of s in ρ.

If we consider as conformance relation between abstract and refined states the equality of functions `levelTest` (representing the current level) and `certMsg` (representing the status of the certification), all our refinement steps are correct. Let us consider the third refinement in which we model the fact that the certification is granted only when three images are recognized at the same level. The proof of correct refinement in case of (a) wrong answer, (b) right answer to decrease the level, (c) skip answer, and (d) skip command, is straightforward (since there is surely an abstract run equal to the refined run). We must also show that the runs in which a given level is certified (i.e., three shown images are recognized) have corresponding runs in the abstract machine. A refined state in which the shown images have been recognized once or twice is stuttering conformant with the abstract state in which no image has been recognized yet. Fig. 8 shows an example of refined run certifying level 1 and a corresponding abstract run.

Similar arguments for proving refinement correctness can be done for all the refinements. Note that the proof of the refinement correctness is completely automatic: the designer must only indicate which are, in the two models, the "locations of interest" [8] that are involved in the conformance relation.

B. Simulation

Simulation is a validation activity. We have executed the produced ASM models by the `AsmetaS` simulator. It allows interactive simulation and random simulation. In interactive simulation, at each step the user is asked for the values of the monitored functions, whereas in random simulation the simulator itself randomly chooses the values for monitored functions. Moreover, the `AsmetaS` simulator allows to check if some invariants are satisfied during simulation. For example, we introduced the following invariant:

```
(certMsg=CERTIFIED and not loop) implies
levelCertificate=1
```

It states that if the patient has been CERTIFIED and (s)he has never increased the level (i.e., function `loop` is false), (s)he is definitely certified at level 1.

[5]http://asmeta.sourceforge.net/download/asmrefprover.html

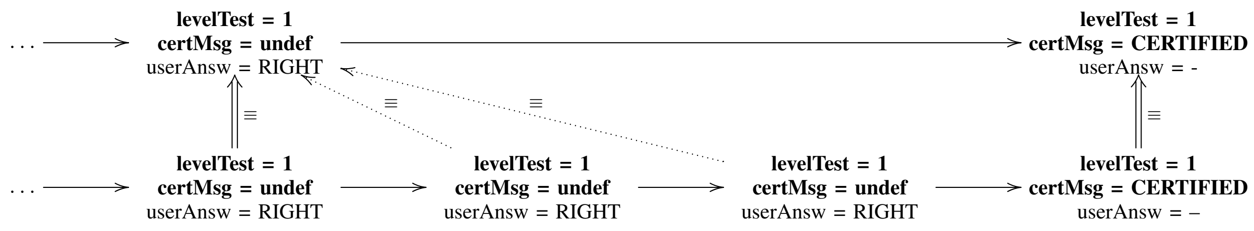


Figure 8. Stuttering refinement from the second refinement model to the third refinement model – Example of refined run

C. Scenario Validation

Simulation is useful in the earlier stages of model development when it is easy to follow the machine executions due to the compactness of the model. When the ASM becomes more complex and large or the simulations become repetitive, `AsmetaV` is a useful tool to support the user in validation. `AsmetaV` permits to specify scenarios describing the interaction between a user (i.e., the environment) and the machine. The `Avalla` language [12] provides constructs to **set** the values of the monitored functions, to execute a **step** of simulation of the ASM, and to **check** that a given closed first-order formula (*assertion*) holds in a given state. The validator `AsmetaV` simulates (using the simulator `AsmetaS`) the ASM model according to the commands of the scenario, and checks if all the assertions are satisfied. As soon as an assertion is not satisfied, the simulation is interrupted reporting the violation. We used `AsmetaV` to write some scenarios for simulating ASM behaviours. Such scenarios have been executed whenever we modified the models to check that the behaviour of the ASM was not altered (in a kind of regression testing). Moreover, we also construct some scenarios for simulating different refined models; indeed, a scenario for an abstract model M should be correct also for a refined model $\widetilde{M}$ if the scenario is related to the elements of M that have not been refined in $\widetilde{M}$. For example, the scenario that simulates the certification of first level without any error has to be the same in the ground model, first, and second refinement. Indeed, first and second refinements add the counting of the wrong answers and the possibility of giving the SKIP answer; however, the handling of correct answers is the same in the three models. As future work, we plan to automatically create a scenario for $\widetilde{M}$ starting from a scenario written for M (possibly also over the elements refined in $\widetilde{M}$), exploiting the correctness proofs produced by `ASMRefProver` (see Sect. VII-A).

D. Model review

Model review is an automatic validation technique that verifies some general properties that any model should guarantee such as *completeness*, *minimality*, and *consistency*. The `AsmetaMA` tool [13] (based on the model checker `AsmetaSMV` presented in Sect. VII-E) allows *automatic* review of ASMs. Typical vulnerabilities and defects that can be introduced during the modeling activity using ASMs are checked as violations of suitable *meta-properties* (*MPs*, defined in [13] as CTL formulae). The violation of a meta-property means that a quality attribute 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). An inconsistent update (meta-property MP1 in [13]), for example, is a signal of a real fault in the model; the presence of functions that are never read nor updated (meta-property MP7 in [13]), instead, may simply indicate that the model is not minimal, but not that it is faulty.

We run the model reviewer on all the developed models. We found that, in the fourth refinement, function `expAnsw` (specifying the expected answer) could not take values SKIP, EXIT (violation of meta-property MP6 in [13]). This violation is not a signal of a real fault in the model, but of a bad choice at design time. Indeed, in our models (starting from the ground model), we use the same domain `Answers` for modeling the patient's choices (i.e., RIGHT and WRONG till the third refinement, the image names in the fourth refinement) and the commands to skip a step and to exit the test (i.e., SKIP and EXIT). In the fourth refinement, we model the random choice of an image by the machine and we store the chosen value in `expAnsw`: since for the function codomain we use the same domain `Answers` used for the patient's choice, we must avoid to select SKIP and EXIT as chosen images. A better design choice would have been to separately model the available images and the test execution commands.

In a preliminary version of our ground model, instead, we found a real fault. We wrongly wrote `levelTest < 7` instead of `levelTest < 6` in the guard of a conditional rule and this did not allow the else branch of the conditional rule to be ever executed (violation of meta-property MP3 in [13]).

These small examples show that model review is a quite powerful push-button validation activity that can be used since the first stages of model development to find faults (e.g., inconsistencies) and/or stylistic detects (e.g., minimality violations) of our models.

E. Property verification

In Sect. VII-D we have shown how model checking is used as a push-button validation activity (i.e., model review) that checks *application-independent* properties, namely properties that any formal specification should assure.

In this section, instead, we use model checking (by means of the `AsmetaSMV` tool) for verifying *application-dependent* properties, i.e., properties specific to our case study. Some have been derived directly from the requirements of the system, others have been added during the verification activity for increasing the requirements completeness. We here report some CTL properties common to all models. Note that, for each developed model, we have also specified more specific properties regarding the requirements considered by that model.

As first property, we check that it is always possible to terminate a test (note that the boolean function `test` is true when the test is running).

AF`(not test)`

Moreover, we check that if the test is terminated, it cannot start again (for us, a test corresponds to a run of the ASM).

AG`(not test implies` **AG**`(not test))`

The decision whether or not to certify a level for a patient can only be made when the test is finished. Therefore, we verify that, during the test, the message containing the certification decision (i.e., function `certMsg`) is undefined and becomes defined when the test is finished.

AG`(test implies isUndef(certMsg))`
AG`(isDef(certMsg) implies not test)`

Note that the previous two properties were also specified as invariants for simulation[6].

Then we check that both decisions can be taken:

EF`(certMsg = CERTIFIED)`
EF`(certMsg = NOTCERTIFIED)`

We further check that, once a decision has been taken (either *certified* or *not certified*), it cannot be changed:

AG`(certMsg = CERTIFIED implies` **AG**`(certMsg = CERTIFIED))`
AG`(certMsg = NOTCERTIFIED implies` **AG**`(certMsg = NOTCERTIFIED))`

F. Scenario and test generation

We have also exploited model checking tools for identifying interesting runs of the models by introducing *trap properties*, which are not actual system properties to be verified. A trap property has form $never(\phi)$, where ϕ is a predicate over the state that we want to *cover* with a system run, and *never* is translated to a corresponding model checker operator. If a state S satisfying ϕ exists, the trap property is false and the returned counterexample is a trace leading to S. Such counterexamples are used in the scenario-based validation for constructing execution scenarios (see Sect. VII-C) and in the test framework for testing the implementation (see Sect. VII-G). Some trap properties we have specified are:

never`(certMsg = NOTCERTIFIED)`
never`(certMsg = CERTIFIED and levelCertificate = 1)`
...
never`(certMsg = CERTIFIED and levelCertificate = 6)`

[6]The tool AsmetaSMV, for each invariant φ, automatically creates the CTL property $\mathbf{AG}(\varphi)$.

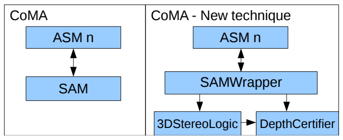


Figure 9. Wrap class for testing

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

@Asm(asmFile = "models/SAM_ref4.asm")
public class SAMWrapper {
    @FieldToLocation(func = "certMsg")
    OutMessage outMessage;
    @FieldToLocation(func = "test")
    boolean test;
    @Monitored(func = "expAnsw")
    ShapeW chosenShape;
    @Monitored(func = "userAnsw")
    AnswersW userAnswer;
    3DStereoLogic logic = new 3DStereoLogic(5);

    @StartMonitoring
    public SAMWrapper() {
        logic.startTest(new ExperimentSetupData(6), false);
        test = (certifier.getMode() == TEST);
    }

    @RunStep
    public void chooseShape() { ... }

    @MethodToFunction(func = "levelTest")
    int getCurrentLevel() {
        return certifier.getCurrentDepth();
    }
}
```

Code 1. Java implementation of SAM Wrapper

By means of these properties, we are able to generate a scenario (test) in which the exam is terminated without certifying the patient and six different scenarios (tests) in which the patient is certified in one of the six levels.

G. Conformance checking

In this section, we show how we have used the formal specifications for testing the Java implementation of SAM. First of all, we have to link the implementation with the formal specification: in order to do this, we use a set of Java annotations originally introduced in the runtime verification framework `CoMA` [16]. Such annotations link both the data part (some fields and pure methods of the implementation are linked to functions of the specification) and the behaviour (the execution of some selected methods corresponds to a step of the ASM). In the original version of `CoMA`, only one class can be connected with the formal specification. In this work, we extend `CoMA` to handle more complex programs: If a program is composed of several classes, the tester writes a wrapper class and connects it with the specification. In our case, SAM is composed of the main `3DStereoLogic` class, and the `DepthCertifier` class, which is internally used by `3DStereoLogic`. A wrapper is used as shown in Fig. 9. Code 1 reports its Java implementation annotated for testing and connected to the last refined model. For example, for the data part, the field outMessage is connected to the ASM function `certMsg` (in the implementation, the certification result is shown as an output message) and the pure method getCurrentLevel is connected to the ASM function `levelTest`; for the behavioural part, the execution of method chooseShape corresponds to an ASM step.

To produce tests from formal specifications, we use the model checking approach described in Sect. VII-F; we use the ATGT tool [15] which is based on the model checker SPIN.

The produced tests are *abstract* and they need to be concretized in tests for the implementation (e.g., JUnit tests): in order to do this, we use the technique introduced in [19] that exploits the linking provided by the Java annotations.

	#tests	total length of tests	Achieved coverage (%)	
			3DStereoLogic	DepthCertifier
Handmade	15	872	62.1	60.8
BFS	14	1181	62.5	69.3
DFS	8	1688	64.4	83.4

Table II
CODE COVERAGE

In a preliminary version of the Java implementation, we found an error related to the counting of the recognized images. The requirements prescribe that a level i is certified if the patient recognizes three times the shown images, *without changing the level between two recognitions*. Thus, the number of recognized images in a level is reset when the patient change the level. The preliminary Java implementation, instead, did not reset the counters for the different levels: a patient could be wrongly certified at level 2 by recognizing the image at level 2, moving to level 1, failing twice at level 1, and guessing twice at level 2 (i.e., recognizing three times at level 2 but moving to level 1 between two recognitions).

We have tested the implementation (whose size is 2.6 KLOCs) with three different test suites: one we generated manually by reasoning on the requirements (*Handmade*), one generated by ATGT using a Breadth-First-Search (*BFS*) approach in generating the tests, and the last one generated by ATGT using a Depth-First-Search (*DFS*) approach. Table II reports the obtained results in terms of number of tests, global test length, and achieved coverage. The coverage is split between the two main components of the program SAM. We can observe that the handmade test suite has the minimum number of test instructions, but also obtains the lowest coverage. DFS obtains better coverage than BFS, although both test suites have been built for covering the same set of test predicates over the ASM; this is due to the fact that a depth-first approach usually builds longer tests that are able to exercise code instructions not reached by the usually shorter tests generated by a breadth-first approach. Finally, we observe that the advantage of using an automatic approach becomes significant when the program under test is particular complex (as the `DepthCertifier` component); for simple programs (as the `3DStereoLogic` component), instead, the handmade approach can still obtain good results.

Note that the linking of the implementation with its formal specification can also be used for checking the conformance at runtime [16] (in a kind of *online testing*).

VIII. Related work

In recent times, the use of formal methods is escalating for the development of software-intensive medical systems. For example, Osaiweran et al. [20] use the formal Analytical Software Design (ASD) [21] approach for developing the power control service of an interventional X-ray system. Jiant et al. [22] present a methodology based on timed automata to extract timing properties of a heart that can be used for the validation and verification of implantable cardiac devices. Tuan et al. [23] provide a solution for the pacemaker challenge using model checker PAT (Process Analysis Toolkit) [24]. Méry et al. [25] and Macedo et al. [26] present a model of pacemakers in Event-B and VDM methods, respectively. Arney et al. [27] present a reference model of PCA (Patient Control Analgesia) infusion pumps and test the model for structural and safety properties. José et al. [28] present a formal model in MAL (Modal Action Logic) [29] that helps comparing different infusion devices and their provided functionalities. Bowen et al. [30] use the ProZ model checker [31] to test various safety properties of infusion pumps. Mashkoor et al. [32] use the Event-B method to specify the behavior of dialysis machines, and verify and validate various system properties using theorem proving, model checking and animation.

The approach for medical software components development we have followed in this paper shares with some of the others the use of a rigorous formalism for system specification, but the V&V activities are more integrated into the development process as compared to the aforementioned works.

We cover a multitude of model analysis activities, e.g., model checking, simulation, model review, testing, conformance checking, that give us a grasp on the notion of correctness far better than approaches which are comprised of only a subset of analysis techniques we have employed. Additionally, ASM method's ease of use, understandability and notion of refinement helped us to manage the complexity of the critical software development process while remaining fully compliant with the guidelines of FDA and recommendations of related medical standards. The basic principle of ASM's model-based development via refinement is discussed and compared in detail in [33].

IX. Conclusions

Medical device software, due to its critical application, is required to be thoroughly validated and verified in order to avoid (serious) injuries that may occur in case of malfunctioning. IEC 62304 classifies medical software based on possible injuries caused by the medical device malfunctions, while FDA principles describe the activities that must be carried out in order to certify a medical software.

In this paper, we have shown a formal development process, based on the Abstract State Machine method, that integrates the software life cycle activities required by the standard and fulfills the FDA principles. Starting from a very abstract model of the system, the developer can obtain, through a sequence of refinements, more detailed models till a level where the model is very close to the final implementation (that could be considered as last step of refinement). Each step of refinement can be proved correct in an automatic way. Moreover, different validation (simulation and model review) and verification (model checking) activities can be carried out, at any level of abstraction, on all the specified models. The process allows for an early planning of the validation and verification activities, and repeating them along the steps of software development and upon any software change. Software safety properties can be guaranteed in a rigorous and human independent way. Finally, the conformance of the implementation w.r.t. the specification can be checked through testing and runtime verification.

We have shown the application of the proposed process to the development of a real medical software that is used for measuring the patient's stereoacuity.

In the future, we plan to apply the proposed formal process for the rigorous development and validation of the other components of the 3DSAT medical software. Although using control state ASMs and scenario-based validation proved to be very useful for communicating with non-expert users, we plan to provide more facilities for facilitating the communication with the stakeholders. We also plan to improve the process, especially the techniques regarding model refinement and automatic generation of simulation scenarios.

REFERENCES

[1] *ISO 13485:2003 Medical devices – Quality management systems – Requirements for regulatory purposes*, International Organization for Standardization Std., 2003.

[2] *ISO 14971:2007 Medical devices – Application of risk management to medical devices*, International Organization for Standardization Std., 2007.

[3] *IEC 60601-1:2005 Medical electrical equipment – Part 1: General requirements for basic safety and essential performance*, International Electrotechnical Commission Std., 2005.

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

[5] *IEC 62304 - Medical device software - Software lifecycle processes*, International Electrotechnical Commission Std., 2006.

[6] U.S. Food and Drug Administration (FDA), "General principles of software validation; final guidance for industry and fda staff, version 2.0," Jan. 2002. [Online]. Available: http://www.fda.gov/MedicalDevices/DeviceRegulationandGuidance/GuidanceDocuments/ucm085281.htm

[7] R. Jetley, S. Purushothaman Iyer, and P. Jones, "A formal methods approach to medical device review," *Computer*, vol. 39, no. 4, pp. 61–67, April 2006.

[8] E. Börger and R. F. Stark, *Abstract State Machines: A Method for High-Level System Design and Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2003.

[9] 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*, pp. 1–23, 2015.

[10] P. Arcaini, A. Gargantini, E. Riccobene, and P. Scandurra, "A model-driven process for engineering a toolset for a formal method," *Softw., Pract. Exper.*, vol. 41, no. 2, pp. 155–166, 2011.

[11] A. Gargantini, E. Riccobene, and P. Scandurra, "A Metamodel-based Language and a Simulation Engine for Abstract State Machines," *J. UCS*, vol. 14, no. 12, pp. 1949–1983, 2008.

[12] 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)*, ser. LNCS, vol. 5238. Springer-Verlag, 2008, pp. 71–84.

[13] 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)*, C. Muñoz, Ed. NASA, 2010, pp. 4–13.

[14] ——, "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)*, ser. LNCS, vol. 5977. Springer, 2010, pp. 61–74.

[15] A. Gargantini, E. Riccobene, and S. Rinzivillo, "Using Spin to Generate Tests from ASM Specifications," in *Abstract State Machines 2003*, ser. LNCS, E. Börger, A. Gargantini, and E. Riccobene, Eds., vol. 2589. Springer Berlin Heidelberg, 2003, pp. 263–277.

[16] P. Arcaini, A. Gargantini, and E. Riccobene, "CoMA: Conformance monitoring of Java programs by Abstract State Machines," in *Runtime Verification*, ser. LNCS, vol. 7186. Springer, 2012, pp. 223–238.

[17] P. DasGupta, "A roadmap for formal property verification," in *A Roadmap for Formal Property Verification*. Springer Netherlands, 2006, pp. 217–241.

[18] A. Gargantini, G. Facoetti, and A. Vitali, "A random dot stereoacuity test based on 3d technology," in *Proceedings of the 8th International Conference on Pervasive Computing Technologies for Healthcare*, ser. PervasiveHealth '14. Brussels, Belgium: ICST, 2014, pp. 358–361.

[19] P. Arcaini, A. Gargantini, and E. Riccobene, "Combining Model-Based Testing and Runtime Monitoring for Program Testing in the Presence of Nondeterminism," in *2013 IEEE 6th Int. Conference on Software Testing, Verification and Validation, Workshops Proceedings, Luxembourg, March 18-22, 2013*. IEEE, 2013, pp. 178–187.

[20] A. Osaiweran, M. Schuts, J. Hooman, and J. Wesselius, "Incorporating formal techniques into industrial practice: an experience report," *Electronic Notes in Theoretical Computer Science*, vol. 295, no. 0, pp. 49 – 63, 2013, proceedings the 9th Int. Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA).

[21] G. Broadfoot, "ASD case notes: Costs and benefits of applying formal methods to industrial control software," in *FM 2005: Formal Methods*, ser. LNCS, J. Fitzgerald, I. Hayes, and A. Tarlecki, Eds. Springer Berlin Heidelberg, 2005, vol. 3582, pp. 548–551.

[22] Z. Jiang, M. Pajic, A. Connolly, S. Dixit, and R. Mangharam, "A platform for implantable medical device validation: demo abstract," in *Proceedings of Wireless Health 2010, WH 2010, San Diego, CA, USA, October 5-7, 2010*, I. M. Jacobs, P. Soon-Shiong, E. Topol, and C. Toumazou, Eds. ACM, 2010, pp. 208–209.

[23] L. A. Tuan, M. C. Zheng, and Q. T. Tho, "Modeling and verification of safety critical systems: A case study on pacemaker," in *Secure Software Integration and Reliability Improvement (SSIRI), 2010 Fourth International Conference on*, June 2010, pp. 23–32.

[24] J. Sun, Y. Liu, and J. Dong, "Model checking CSP revisited: Introducing a process analysis toolkit," in *Leveraging Applications of Formal Methods, Verification and Validation*, ser. CCIS, T. Margaria and B. Steffen, Eds. Springer Berlin Heidelberg, 2008, vol. 17, pp. 307–322.

[25] D. Méry and N. K. Singh, "Formal specification of medical systems by proof-based refinement," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 1, pp. 15:1–15:25, Jan. 2013.

[26] H. Macedo, P. Larsen, and J. Fitzgerald, "Incremental development of a distributed real-time model of a cardiac pacing system using VDM," in *FM 2008: Formal Methods*, ser. LNCS, J. Cuellar, T. Maibaum, and K. Sere, Eds. Springer Berlin Heidelberg, 2008, vol. 5014, pp. 181–197.

[27] D. Arney, R. Jetley, P. Jones, I. Lee, and O. Sokolsky, "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*, ser. HCMDSS-MDPNP '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 23–33.

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

[29] J. C. Campos and M. Harrison, "Systematic analysis of control panel interfaces using formal tools," in *Interactive Systems. Design, Specification, and Verification*, ser. LNCS, T. Graham and P. Palanque, Eds. Springer Berlin Heidelberg, 2008, vol. 5136, pp. 72–85.

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

[31] D. Plagge and M. Leuschel, "Validating Z specifications using the ProB animator and model checker," in *Integrated Formal Methods*, ser. LNCS, J. Davies and J. Gibbons, Eds. Springer Berlin Heidelberg, 2007, vol. 4591, pp. 480–500.

[32] A. Mashkoor, M. Biro, M. Dolgos, and P. Timar, "Refinement-based development of software-controlled safety-critical active medical devices," in *Software Quality. Software and Systems Quality in Distributed and Mobile Environments*, ser. LNBIP, D. Winkler, S. Biffl, and J. Bergsmann, Eds. Springer International Publishing, 2015, vol. 200, pp. 120–132.

[33] R. Banach, "Model based refinement and the tools of tomorrow," in *Abstract State Machines, B and Z*, ser. LNCS, E. Börger, M. Butler, J. P. Bowen, and P. Boca, Eds. Springer Berlin Heidelberg, 2008, vol. 5238, pp. 42–56.