Developing medical devices from Abstract State Machines to embedded systems: a smart Pill Box case study

Andrea Bombarda[1], Silvia Bonfanti[1], and Angelo Gargantini[1]

Department of Economics and Technology Management,
Information Technology and Production
University of Bergamo, Italy
{andrea.bombarda,silvia.bonfanti,angelo.gargantini}@unibg.it

Abstract. The development of medical devices is a safety-critical process, because a failure or a malfunction of the device can cause serious injuries to the patients whom use it. The application of a rigorous process for their development reduces the risk of failures since validation and verification activities can be performed in a objective, reproducible, and documentable manner. In this paper we present an approach based on the Abstract State Machine (ASM) formal method. Starting from the model, validation and verification (V&V) techniques can be applied. Furthermore, by step-wise refinement, a final model can be obtained, which can be automatically translated to C++ code. The process is applied to the smart pill box case study. Starting from the ASM model, we generate C++ code for the Arduino platform after the application of V&V activities. Furthermore, we introduce regulation (IEC62304) and guidelines (FDA General Principles of Software Validation) that support the developer in medical software development. In particular, we explain how ASMs formal process can be compliant with them.

1 Introduction

Software is becoming an essential part of medical devices, so it is very important that its development process adheres to certification standards. All the standards available provide only general description of common software engineering activities, but nothing is said about the techniques that have to be used to guarantee the safety of the devices and the correctness of their software. The main references concerning the regulation of medical software are the standard IEC 62304 (International Electrotechnical Commission) [12] (see Sect. 6.1) and the FDA guidelines [15] in which several concepts that can be used as guidance for software validation and verification are defined (see Sect. 6.2). The regulation and the guideline aim for more rigorous approaches for software development and validation, but neither of them recommend a particular method or technique.

In this paper we propose a formal approach that can be used to develop and validate the software of an embedded medical device, in compliance with the IEC regulation and FDA guidance for software validation as shown in Sect 6.

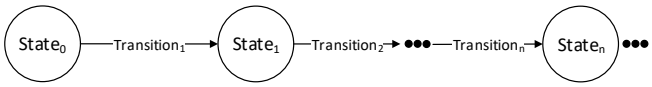


Fig. 1. An ASM run with a sequences of states and state-transitions (steps)

Our formal approach is studied over a simple example of a smart portable pill box, called e-Pix (electronic PIll boX), modelled with Abstract State Machines (ASM) by using the `Asmeta` framework. We have applied several validation and verification (V&V) techniques [13], such as model simulation (see Sect. 4.3), scenario-based testing (see Sect. 4.4) using the `Avalla` language, and property verification (see Sect. 4.5). As final step, we have used `Asm2C++` to generate the C++ code to be executed by Arduino.

The paper is organized as follows. Sect. 2 introduces the ASMs and all the tools provided by the `Asmeta` framework. In Sect. 3 we explain the e-Pix case study. Sect. 4 presents modelling by refinement, validation, testing and verification procedures applied to the case study. Sect. 5 explains how we have built the prototype of e-Pix and generated C++ code. Sect. 6 gives a comprehensive review about how our approach can be used to comply the main regulations concerning the development of medical software. Sect. 7 presents works related to the use of rigorous approaches in medical software development, and Sect. 8 concludes the paper.

2 Abstract State Machines and `Asmeta` framework

Abstract State Machines (ASMs) [7] are an extension of Finite State Machines (FSMs), where unstructured control states are replaced by states with arbitrarily complex data. ASM *states* are mathematical structures, i.e., domains of objects with functions and predicates defined on them. An ASM *location* - defined as the pair (*function-name*, *list-of-parameter-values*) - represents the abstract ASM concept of basic objects container. The ordered pair (*location*, *value*) represents a machine memory unit.

Location values are changed by firing *transition rules*. They express the modification of functions interpretation from one state to the next one. Note that the algebra signature is fixed and that functions are total (by interpreting undefined locations $f(x)$ with value *undef*). Location *updates* are given as assignments of the form $loc := v$, where loc is a location and v is its new value. They are the basic units of rule construction. There is a limited but powerful set of *rule constructors* to express: guarded actions, simultaneous parallel actions, sequential actions, nondeterminism, and unrestricted synchronous parallelism.

An ASM *computation* or *run* is, therefore, defined as a finite or infinite sequence of states $s_1, s_2, \ldots, s_n, \ldots$ of the machine. s_1 is an initial state and each s_{i+1} is obtained from s_i by firing the unique *main rule*, which could fire other transitions rules (see Fig. 1).

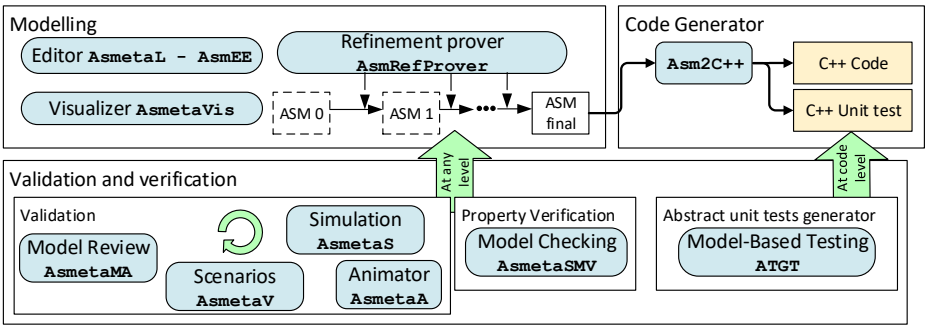


Fig. 2. The ASM development process powered by the `Asmeta` framework

During a machine computation, not all the locations can be updated. Functions are classified as *static* (never change during any run of the machine) or *dynamic* (may change as a consequence of agent actions or *updates*). Dynamic functions are distinguished between *monitored* (only read by the machine and modified by the environment) and *controlled* (read in the current state and updated by the machine in the next state). A further classification is between *basic* and *derived* functions, i.e., those coming with a specification or computation mechanism given in terms of other functions.

An ASM can be *nondeterministic* due to the presence of monitored functions (*external* nondeterminism) and of choose rules (*internal* nondeterminism).

`Asmeta` framework. The ASM method can facilitate the entire life cycle of software development, i.e., from modeling to code generation. Fig. 2 shows the development process based on ASMs. The process is supported by the `Asmeta` (ASM mETAmodeling) framework[1] [4] which provides a set of tools to help the developer in various activities:

– **modeling**: the system is modeled using the language `AsmetaL`. The user is supported by the editor `AsmEE` and by `AsmetaVis`, the ASMs visualizer which transforms the textual model into a graphical representation. The user can directly define the last ASM model or s/he can reach it through refinement. The refinement process is adopted in case the model is complex. In this case, the designer can start from the first model (also called the ground model) and can refine it through the refinement steps by adding details to the behavior of the ASM. The `AsmRefProver` tool ensures whether the current ASM model is a correct refinement of the previous ASM model.

– **validation**: the process is supported by the model simulator `AsmetaS`, the animator `AsmetaA`, the scenarios executor `AsmetaV`, and the model reviewer `AsmetaMA`. The simulator `AsmetaS` allows to perform two types of simulation: interactive simulation and random simulation. The difference between the two

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

types of simulation is the way in which the monitored functions are chosen. During interactive simulation the user inserts the value of functions, while in random simulation the tool randomly chooses the value of functions among those available. `AsmetaA` allows the same operation of `AsmetaS`, but the states are shown using tables which make the readability of the state easier. `AsmetaV` executes scenarios written using the `Avalla` language. Each scenario contains the expected system behavior and the tool checks whether the machine runs correctly. The model reviewer `AsmetaMA` performs static analysis in order to check quality attributes like minimality, completeness, and consistency.

– **verification**: the properties derived from the requirements document are verified to check whether the behavior of the model complies with the intended behavior. The `AsmetaSMV` tool supports this process.

– **testing**: the tool `ATGT` generates abstract unit tests starting from the ASM specification by exploiting the counterexamples generation of a model checker.

– **code generation**: given the final ASM specification, the `Asm2C++` automatically translates it into C++ code. Moreover, the abstract tests, generated by the `ATGT` tool, are translated to C++ unit tests.

3 The e-Pix case study

Adherence to pharmacological therapy [8] is one of the most well-known problem in medical field. Sometimes it happens that the patient is not adherent to the therapy because he does not remember to take the medicine or he does not remember if he has already taken it. For this reason the patients have the need to adopt a system that can help them to follow the therapy. The device introduced in the market is the pill box, where pills are inserted based on the scheduled doses of medications. The first pill boxes were simply multicompartment boxes where each compartment was filled with the corresponding medicine. The simplest boxes have one section for each day, while the most complicated have multiple sections corresponding to different times of the day. The box helps the user to prevent/reduce medication errors because once the pills are in the correct section the user has only to remember to take it at the right time. With the introduction of technology in the medical field, even the pill boxes have evolved. They are integrated with electronic components that provide alerts to patients when the time of medicine comes. Usually the pill box is provided with a memory where the list of pills with the therapy time are saved and at the right time the box notifies to the user. The notifications can take place with a sound/light signal or, for smarter pill boxes, they can be displayed on the smartphone. In this paper, we consider a pill box developed using Arduino[2], an open-source electronic prototyping platform. We were asked by a local company to re-engineer the software of an existing pocket portable pillbox called e-Pix, following the guidelines discussed in Sect. 6. In particular, the company wants to certify its product w.r.t. the FDA guidelines and IEC regulation and, because of that, needs

[2] https://www.arduino.cc/

to be sure it works properly. Furthermore, they have provided some functional requirements which the prototype has to satisfy, e.g. if the patient does not take the pill in time the red light of the corresponding compartment has to blink.

Requirements. The existing e-Pix has an array of *compartments* each containing an unique type of unpackaged pills and having a sensor able to signal the opening of the related window. Each compartment is provided with a red led, used as output to indicate which pill has to be taken (red led turns on until the patient opens the compartment) and if the pill has been taken. When the pill time has passed and the set timeout expires the red led starts to blink for a certain period of time to attract patient's attention because he forgot the pill. After that the red led is turned off and a message is shown on the e-Pix display. In case the patient takes the pill but he forgot closing the compartment window, the red led starts to blink for a certain period of time. The prescription file is generated by the user by interacting with some buttons on the e-Pix and stored inside e-Pix as JSON file (see Code 1). It contains for each pill the compartment in which it is contained, the name of the pill and the time at which the pill has to be taken (expressed as the number of seconds passed since 01/01/1970). e-Pix loads the JSON file containing the times at which the pills have to be taken during the initialization phase and, following the schedule, indicates when the patient has to take the pill from the compartment.

```
{ "patient" : "patient_name",
   "pills" : [
      { "compartment" : "
            compartment_number",
         "name" : "pillName",
         "time_consumption" :
                  ["t1", "t2", ...],
      },
      {...}
   ]
}
```

Code 1. Example of the JSON file containing the prescriptions

4 Modeling and V&V

Starting from the informal requirements of the e-Pix, we have applied the process described in Figure 2. Using the editor `AsmEE` we have implemented the AsmetaL specifications[3] with different refinement levels. Then validation and verification tools have been used to validate and to verify the model.

Refinement	Time management	# Compartments	# controlled functions	# monitored functions
Level 0	Monitored boolean function that indicates the overpassing of the time threeshold	0	4	3
Level 1	Controlled by the system	0	8	1
Level 2	Controlled by the system	3	9	1
Level 3	Monitored function	3	8	2

Table 1. Refinement levels of e-Pix (0 ompartments means only one type of pills).

[3] The specifications are available at https://foselab.unibg.it/asmeta/PillboxASM.zip

4.1 Modeling by refinement

We have modeled e-Pix starting from a simple model and then applying stepwise refinement. At each refinement step we have introduced some controlled and monitored functions, we have gradually added compartments, and we have managed the time differently - using some abstractions at level 0, having a controlled time at levels 1 and 2, and as monitored value at the final level (see Table 1).

In the following paragraphs we explain the main characteristics of each refinement level and we analyze how we have modeled the switching on of the red led when the time of a pill comes.

Level 0. In the first model, i.e. the *ground model*, we have considered only one pill and no compartments. Instead of using an actual timer, a boolean monitored function `takeThePill` reports when the pill has to be taken. Similarly, the overpassing of all the timeouts (used to switch from a state of the LED to another state) is indicated by the boolean function `timeDiffOver600`. The red led is switched on when a pill has to be taken and it is managed by the following AsmetaL rules:

```
main rule r_Main =
  [...]
  if redLed = OFF and takeThePill then
    r_pillToBeTaken[] endif
  if redLed = ON and not timeDiffOver600 and
  opened and not openSwitch then
    r_pillTaken_compartmentOpened[] endif
  [...]
```

```
rule r_pillToBeTaken =
  par
    redLed := ON
    outMess := TAKE_PILL
  endpar
rule r_pillTaken_compartmentOpened =
  par
    redLed := OFF
    outMess := NONE
  endpar
```

Level 1. At this refinement level, we have continued considering only one pill and no compartments. The time management has been realized using the function `systemTime` as `Natural`, controlled by the system and increased at each machine step. Also at this refinement level we have not considered the list of prescriptions, but only a single deadline for the contemplated pill: we have used a boolean function, `requestSatisfied` to check whether the pill has already been taken or not. The possible output and log messages are taken from an enumerative domain `OutMessages`. The condition according to which the red led is switched on checks the value of the actual timer of e-Pix, whose value is controlled by the system:

```
main rule r_Main =
[...]
if redLed = OFF and (time_consumption<=
      systemTime and not requestSatisfied)
then  r_pillToBeTaken[]
endif
[...]
```

```
rule r_pillToBeTaken =
par
if redLed != ON then
compartmentTimer := systemTime endif
redLed := ON
outMess := TAKE_PILL
endpar
```

Level 2. The second refinement level introduces three compartments, each with a single type of pill. Other features are similar to the previous level: we have used a single deadline for each pill, the output and log messages come from the enumerative domain `OutMessages`, the timer `systemTime` is managed by the system

and takes value in a bounded range. Compared to the previous refinements, the red led switch on condition is checked for each single compartment as follows:

```
main rule r_Main =
[...]
if redLed($compartment) = OFF and
  (time_consumption($compartment)<=systemTime and not requestSatisfied($compartment)) then
  r_pillToBeTaken[$compartment]
endif
[...]

rule r_pillToBeTaken($compartment in Compartment) =
par
  if redLed($compartment) != ON then compartmentTimer($compartment) := systemTime endif
  redLed($compartment) := ON
  if ($compartment=compartment1) then
    outMess($compartment) := TAKE_TYLENOL
  else if ($compartment=compartment2) then
    outMess($compartment) := TAKE_ASPIRINE
  else
    outMess($compartment) := TAKE_MOMENT
  endif endif
endpar
```

Level 3. The 3rd model we have considered has three compartments and we have included all the features of the system:

- The `systemTime` is monitored from the machine and updated by the environment.
- Every string can be used as output and log message.
- It is possible to assign a list of time prescriptions for each compartment, stored in the function `time_consumption`.

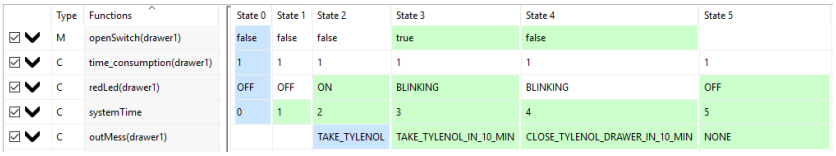
The guard that makes the red led switch on, when it is time to take the pill, has been modified with respect to the previous levels, because we have to manage more prescriptions for each pill. The correct item in the sequence containing the prescription times, i.e. the current time threshold to be considered, is selected by the function `drugIndex`. Therefore, for the compartment d, when `systemTime` passes `time_consumption(`d`)` at position `drugIndex(`d`)`, then the pill in d should be taken.

```
main rule r_Main =
[...]
  if redLed($compartment) = OFF and
  (at(time_consumption($compartment),drugIndex($compartment))<systemTime) then
    r_pillToBeTaken[$compartment]
endif
[...]

rule r_pillToBeTaken($compartment in Compartment) =
par
  if redLed($compartment) != ON then
    compartmentTimer($compartment) := systemTime endif
  redLed($compartment) := ON
  outMess($compartment) := "Take " + name($compartment)
endpar
```

4.2 Automatic refinement proof

To automatically prove the correctness of the model refinement process, used in our ASM formal approach, we have employed the tool `AsmRefProver`, which is



	Type	Functions	State 0	State 1	State 2	State 3	State 4	State 5
☑	M	openSwitch(drawer1)	false	false	false	true	false	
☑	C	time_consumption(drawer1)	1	1	1	1	1	1
☑	C	redLed(drawer1)	OFF	OFF	ON	BLINKING	BLINKING	OFF
☑	C	systemTime	0	1	2	3	4	5
☑	C	outMess(drawer1)			TAKE_TYLENOL	TAKE_TYLENOL_IN_10_MIN	CLOSE_TYLENOL_DRAWER_IN_10_MIN	NONE

Fig. 3. Simulation steps with the animator `AsmetaA` at the last refinement level

based over the Satisfability Modulo Theories (SMT). With the execution of this software, presented in [3], one can specify two refinement levels and ensure that an ASM specification is a correct refinement of a more abstract one.

In our case study we have proven the correctness of the refinement process. To make this possible, since `AsmRefProver` maps refined functions to abstract ones with the same name, we had to introduce in the refined level, some `derived` functions representing predicates over the abstract or refined states. For example, in the first refinement level, to prove the correctness of the refinement process, we have added two derived functions: `takeThePill` that indicates if the patient has to take the pill and `timeDiffOver600` to represent if the patient has forgot taking the pill within a certain time.

```
function takeThePill = (time_consumption<=systemTime)
function timeDiffOver600 = (systemTime−compartmentTimer>tenMinutes)
```

4.3 Validation

Validation activity consists in the execution of different tools. Initially we have validated the specification using the simulator `AsmetaS` and the animator `AsmetaA`. In particular we have intensively used the animator because it provides a graphical interface which is more readable for the user during the model execution.

In Fig. 3 we have reported some simulation steps using the animator `AsmetaA`. Specifically, after the system initialization, we have simulated the scenario in which the time is controlled by the ASM and we have only a pill in the first compartment. The red led goes `ON` when it is time to assume the pill (`systemTime > time_consumption`) and turns to `BLINKING` when the timeout has passed. When the compartment is closed the red led turns `OFF`. Also the display message (`outMess`) changes according to the state of e-Pix.

4.4 Scenario-based testing

In the scenario-based testing activity we have checked the behaviour of e-Pix against the expected one by simulating all the possible states, and transitions between them.

We have written our scenarios using the `Avalla` language [9] and tested each scenario with the validator `AsmetaV`, which checks if the machine runs as expected. We have also

```
// Setting−up the initial state
set openSwitch(comp1) := false;
set openSwitch(comp2) := false;
set openSwitch(comp3) := false;

step

check redLed(comp1) = OFF;
check outMess(comp1) = NONE;
check logMess(comp1) = NONE;

// Time to take the pill in comp1
step until systemTime = 2;

check redLed(comp1) = ON;
check outMess(comp1) =
      TAKE_TYLENOL;
check logMess(comp1) = NONE;
```

Code 2. Example of an Avalla

checked, with the coverage evaluation tool included into `AsmetaV`, that our scenarios execute all the rules of the ASM model. An example of the tested scenarios is shown in Code 2. Initially all the compartments are closed and after the ASM step the red led is off and no messages are shown. When the time to take the pill is reached (step until command) the state changes, the red led turns on and the message shows which pill the patient has to take.

4.5 Property Verification: AsmetaSMV

Once the modeler is confident enough that the model correctly reflects the intended requirements, heavier techniques can be used for property verification. In the proposed case study we have identified four CTL (*Computational Tree Logic*) properties that we have tested in the refined models:

P1 If the pill has to be taken, red led must lights up.
P2 If the patient does not take the pill or the compartment has to be closed, the red light has to blink.
P3 The red light has to change value after 10 minutes if the patient does not take the pill.
P4 If the patient takes the pill and closes the compartment, red light becomes off.

We have generated SMV models from the ASM specification using `AsmetaSMV` and we have verified the properties by means of the model checker NuSMV[4]. Table 2 reports the first property **P1** verified in the models, all the others are available online.

level	CTLSPEC
0	**ag**((takeThePill and redLed = OFF)**implies** ax(redLed = ON))
1	**ag**((takeThePill and not requestSatisfied and redLed = OFF)**implies** ax(redLed = ON))
2	(**forall** $d **in** Compartment **with ag**((time_consumption($d)<systemTime and not requestSatisfied ($d)and opened($d)and not(openSwitch($d))and not(redLed($d)= OFF)and not(systemTime− compartmentTimer($d)>=tenMinutes))**implies** ax(redLed($d)= OFF)))

Table 2. The property P1 in different refinement levels

The property is different from one model to the other because we have managed the time differently (initially it was a monitored function, then we have

[4] http://nusmv.fbk.eu/

used the function `systemTime` controlled by the system and increased at each machine step). Furthermore, in the last case we have added more than one compartment, for this reason the property has been verified over each compartment. It is not possible to test the property on *Level 3* because the model contains unlimited domains (like natural numbers and strings) which are not supported by our model checker.

5 From Asmeta specification to C++ code for Arduino

In addition to the validation and verification activities, we have created an hardware prototype of e-Pix and we have automatically generated the C++ code. The hardware used in our implementation is:

- Arduino Mega 2560
- 3 reed switches, used to signal the opening of each compartment
- 3 red LEDs to signal the state of each compartment
- 1 LCD (Liquid Crystal Display) to interact with the user
- 1 DS3231 module to get the current time
- Arduino SD card reader module, used to store the JSON prescription file and the log ones
- Potentiometers and resistors

Using the `Asm2C++` tool, we have generated from the last ASM refinement level the following files: the `ino`, wich contains the execution policy to run an ASM on Arduino (see Code 3), the `a2c` and the `hw.cpp` files that contain hardware information, the `.h` and `.cpp` files, which contain the translation of the ASM model into C++ code.

```
#include"pillbox.h"
void setup(){
}

pillbox pillbox;

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

Code 3. Example of the ino file containing the implementation of the ASM execution

The `a2c` configuration file is automatically generated by the `Asm2C++` tool to bind each ASM function to an Arduino physical pin. The file must be completed by the user who has to insert the correspondence between Arduino physical pins and functions defined in the ASM model (see Code 4). Then the `hw.cpp` file, which contains C++ code to load the inputs and set the outputs, is automatically produced (see Code 5) to allow the interaction between the software and Arduino physical pins.

6 IEC regulation and FDA guidance application

As reported in Sect. 1, the main references concerning the development of medical software are the IEC 62304 regulation [12] and the FDA General Principles of Software Validation [15]. Afterwards, we map the two documents in the ASM process using the `Asmeta` framework.

```
{
  "arduinoVersion": "
        MEGA2560",
  "stepTime": 0,
  "bindings": [
    { "mode": "DIGITAL",
      "function": "redLed(
            comp1)",
      "pin" : "D1"
    },
    { "mode": "DIGITAL",
      "function": "redLed(
            comp2)",
      "pin" : "D2"
    },
    [...]
  ]
}
```

Code 4. Example of the a2c configuration file

```
#include "pillbox.h"
#include <Arduino.h>
void pillbox::getInputs(){
        openSwitch[comp1] = (digitalRead(7) == HIGH);
        [...]
        systemTime = analogRead(A1)*(double)(1.0/1024.0);
}
void pillbox::setOutputs(){
        if(redLed[1][comp1] == OFF)
                digitalWrite(1, LOW);
        else
                digitalWrite(1, HIGH);
        if(redLed[2][comp1] == OFF)
                digitalWrite(2, LOW);
        else
                digitalWrite(2, HIGH);
        [...]
}
```

Code 5. Example of the `hw.cpp` file

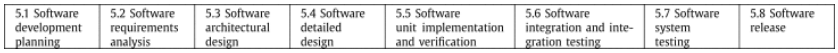


Fig. 4. IEC 62304 development process

6.1 IEC 62304 standard

The standard IEC 62304 [12] does not prescribe a specific life cycle model, it defines process, activities and tasks that the life cycle model has to follow. In particular, we will focus on the characteristics of the software development process (Fig. 4) described in Section 5 of the standard. We have identified how ASMs can be used to satisfy the process.

– *Step (5.1) consists in defining a life cycle model and planning all procedures.* ASMs can supply a precise iterative and incremental life cycle model, based on model refinement. With the ASMs, the developers can perform modeling, validation, verification and conformance checking, which we have performed in Sect. 4 for the e-Pix.

– *Step (5.2) consists in defining and documenting functional and non-functional software requirements.* ASMs can be used to define the system requirements with a mathematical model that can be also analyzed and checked before the implementation development. Informal requirements, which are the results of the requirements gathering activity, are out of the scope of the ASM method. ASMs do not deal natively with non-functional requirements like performance, fault tolerance and reliability either. Thus complementary techniques should be used for these purposes.

– *Step (5.3) regards the specification of the software architecture from the software requirements.* In the e-Pix, the verification of software requirements is executed along all the ASM development process using the property verification tool `AsmetaSMV` (see Sect. 4.5). Risk control can be performed also during

this phase, by verifying the required functional safety properties and executing critical scenario-based testing written in `Avalla` (see Sect. 4.4).

– *Step (5.4) regards the refinement of the software architecture into software units.* The software refinement can be obtained by means of the model refinement mechanism, typical of our ASM approach. We have applied the software refinement to the e-Pix and we have checked the correctness of refinement using the `AsmRefProver` tool (see Sect. 4.2).

– *Steps (5.5)–(5.7) regard the refinement of the software architecture into software units, software implementation and testing at unit, integration, and system levels.* With our ASM-based development process, the actual code can be obtained by the automatic translator `Asm2C++` as last model refinement step, so if the model has been correctly tested, the developers can be sure about the correctness of the C++ code. However the developer can change something in the generated code, so the ASM process cannot fully cover these development steps. For the e-Pix, in Sect. 5 we have automatically generated the Arduino code that we have deployed on the real system.

– *Step (5.8) includes the demonstration, by a device manufacturer, that software has been validated and verified.* If the development process adopts the ASM process, demonstration that the software has been validated and verified is straightforward, since V&V are continuous activities during all the process.

6.2 FDA General Principles of Software Validation

FDA accepts the standard IEC 62304 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, by defining in [15] the list of general principles. For each FDA principle we have identified how ASMs can be used to satisfy the requests.

– *A documented software requirements specification should provide a baseline for both V&V*: in ASM it is provided by means of a chain of models (or single model in case of simple specifications). The models are written using `AsmetaL` language as partially reported in Sect. 4 for the e-Pix models.

– *Developers should use a mixture of methods and techniques to prevent and to detect software errors*: in ASM safety properties are proved on models at each modeling level. In particular `Asmeta` framework provides the `AsmetaSMV` tool that verifies the properties defined by the developer showing if they are satisfied or not. We have applied the property verification to the e-Pix models as reported in Sect. 4.5.

– *Software V&V should be planned early and conducted during all the software life cycle; software V&V should take place within the environment of an established software life cycle; software V&V process should be defined and controlled through the use of a plan*: as shown in Fig. 2 the V&V process can be applied at each model. V&V activities can be integrated in the V model of software development. In particular it is possible to insert them in the module design, coding and unit testing phases.

– *Software V&V process should be executed through the use of procedures*: V&V are supported by precise procedures defined for each tool which have been followed during the application to the e-Pix.

– *Software V&V should be re-established upon any software change*: if software changes do not affect the model, it is required to re-run unit tests on the changed software and verify if the behavior has been modified or not. In case the software changes have effects on the model V&V activities must be re-executed.

– *Validation coverage should be based on the software complexity and safety risks*: during validation activity of an ASM model, it is possible to provide the coverage report in terms of rules, which points out how many lines of code have been covered. It can be used by the designer to estimate if the validation activity is commensurate with the risk associated with the use of the software. The coverage of e-Pix models was 100%, all rules have been covered using the validation activity, in particular the scenario-based testing, as reported in Sect. 4.4.

– *V&V activities should be conducted using the quality assurance precept of "independence of review"*: this can be obtained because V&V are performed by exploiting unambiguous mathematical based techniques.

– *Device manufacturer has flexibility in choosing how to apply these V&V principles*: all the presented V&V activities can be executed at the discretion of the manufacturer because they can be executed independently of each other. Even if the software has been developed by an external developer, the manufacturer can apply the activities presented to guarantee the correctness w.r.t the verified model.

7 Related work

As shown by [6], formal methods are increasingly used in the development of medical software and devices because human safety depends upon the correct operation of the product. Even automatic code generation is already available into commercial solutions (such as MATLAB/Simulink[5]) or UML-based solutions but none of them is based on the ASM method and permits the verification and validation of the written models. In [2], the ASM method has been used to show how an hemodialysis machine can be designed providing a rigorous approach for medical software validation and verification. Despite this, the code to be executed by the final embedded system has not been produced.

The process that allows the automatic code generation has been described into [5] where the car panel case study is analyzed.

Most of the other works related to the approach used into this paper are based on Event-B [1]. These solutions use a multi-formal development paradigm: the requirements are modeled by using UML-B [16] and then the verification is executed into the framework of Event-B using theorems proving the model checking or using model animation. This framework is used into [14] where a hemodialysis machine is developed by specifying the requirements using a refinement-based

[5] https://it.mathworks.com/products/simulink.html

modeling approach. Subsequently model checking and animation techniques are applied to check the consistency and the conformance to the formal requirements. A code generator produces, at the end, the code from the model. The major cons of this solution are that the tool is able to translate only a limited set of the B syntax and it lacks of a formal proof that the produced code maintains all the safety properties of the initial requirements.

There are several papers presenting the design and development of pill box or smart pill dispenser for individual use. Some of them, such as [10], are also Arduino-based. However, no one at the best of our knowledge has adopted a rigorous approach like ours. In [17], the authors present the architecture and the implementation of an automatic medication dispenser. Part of the system is actually generated from models that define user behavior. They have tackled the problem of validating such models mainly by simulation. During simulation, events in interactions of the user, controller and scheduler are registered in a database. They then check the correctness by processing and analyzing the logged events to find errors. A formal modeling has been applied to the design of a mobile prescription application [11]. However, the author has used only UML for modeling of the mobile application.

8 Conclusion

The development of a safe and reliable medical device can be very challenging because it is a safety-critical process. To address the software development in a safer manner, different regulations have been released. However, all these documents are limited to describe only general software engineering activities that have to be executed but they do not require the use of specific method or technique.

In this paper, we have applied the ASM based development process to the smart pill box e-Pix case study. The approach consists in an iterative life cycle model realized by model refinement: starting from a ground model, which considers only the simplest features of the system, the developer can release many incremental models, considering step by step all the characteristics. Along this process, different validation and verification activities (such as model animation, scenario-based validation and property verification) can be performed over each refinement step, to prove the correctness of each produced model compared to the requirements. The final model of the system can be seen as the last refinement step, from which one can obtain the C++ code to be used in the embedded system, thanks to `Asm2C++` tool. In addition, we have developed a simple hardware prototype using Arduino on which we have loaded the generated C++ code, the hardware configuration file and the main Arduino file (all of them automatically generated using the `Asm2C++` tool).

Finally, we have shown how the proposed process aims to guarantee safety and reliability of the final product by remaining compliant with the IEC 62304 regulation and FDA General Principle of Software Validation guidelines.

References

1. Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
2. Paolo Arcaini, Silvia Bonfanti, Angelo Gargantini, Atif Mashkoor, and Elvinia Riccobene. Integrating formal methods into medical software development: The asm approach. *Science of Computer Programming*, 158:148–167, jun 2018.
3. Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. SMT-based automatic proof of ASM model refinement. In *Software Engineering and Formal Methods*, pages 253–269. Springer International Publishing, 2016.
4. Paolo Arcaini, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A model-driven process for engineering a toolset for a formal method. *Software: Practice and Experience*, 41:155–166, 2011.
5. Silvia Bonfanti, Marco Carissoni, Angelo Gargantini, and Atif Mashkoor. Asm2C++: A tool for code generation from abstract state machines to arduino. In *Lecture Notes in Computer Science*, pages 295–301. Springer International Publishing, 2017.
6. Silvia Bonfanti, Angelo Gargantini, and Atif Mashkoor. A systematic literature review of the use of formal methods in medical software systems. *Journal of Software: Evolution and Process*, 30(5):e1943, feb 2018.
7. E. Börger and Robert F. Stark. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag New York, Inc., 2003.
8. Marie T. Brown and Jennifer K. Bussell. Medication adherence: WHO cares? *Mayo Clinic Proceedings*, 86(4):304–314, apr 2011.
9. Alessandro Carioni, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A Scenario-Based Validation Language for ASMs. In *Lecture Notes in Computer Science*, pages 71–84. Springer Berlin Heidelberg.
10. Shih-Chang Huang, Hong-Yi Chang, Yu-Chen Jhu, and Guan-You Chen. The intelligent pill box - design and implementation. In *2014 IEEE International Conference on Consumer Electronics - Taiwan*. IEEE, may 2014.
11. Nicholas Ikhu-Omoregbe. Formal modelling and design of mobile prescription applications. *Journal of Health Informatics in Developing Countries*, 2(2), 2008.
12. P. Jordan. Standard iec 62304 - medical device software - software lifecycle processes. In *2006 IET Seminar on Software for Medical devices*, pages 41–47, Nov 2006.
13. R.A. Kemmerer. Testing formal specifications to detect design errors. *IEEE Transactions on Software Engineering*, SE-11(1):32–43, jan 1985.
14. Atif Mashkoor and Miklos Biro. Towards the trustworthy development of active medical devices: A hemodialysis case study. *IEEE Embedded Systems Letters*, 8(1):14–17, mar 2016.
15. A. Ohne Autor Fd. General Principles of Software Validation; Final Guidance for Industry and FDA Staff, Version 2.0. FDA document formal, January 2002.
16. Colin Snook and Michael Butler. UML-b: Formal modeling and design aided by uml. *ACM Transactions on Software Engineering and Methodology*, 15(1):92–122, jan 2006.
17. Pei-Hsuan Tsai, Tsung-Yen Chen, Chi-Ren Yu, Chi-Sheng Shih, and Jane W. S. Liu. Smart medication dispenser: Design, architecture and implementation. *IEEE Systems Journal*, 5(1):99–110, mar 2011.