Universitè de Lorraine

Erasmus Mundus

Dependable Software Systems

Hemodialysis in Event-B

Author:
Anders Olav
Candasamy

Supervisor:
Dominique Méry

June 8, 2016

1 Introduction

1.1 Programmable Electronic Medical Systems

Programmable electronic medical systems (PEMS) are as the names implies systems that can be programmed. Their goal is to provide medical assistance to a patient. A benefit of such a system is that they to not require constant supervision. The systems can be programmed with the logic required for its successful application. A common example of a PEMS is a insulin pump. PEMS are classified into different categories based on what damages they could cause. A higher classification means that the device must abide to stricter regulation.

1.1.1 Safety

The perhaps most challenging part of creating a PEMS is ensuring its safety. The devices must obey a multitude of requirements to ensure that they are safe to use. As a system bug or failure could cause a loss of life, products that are released to market must have gone through strict procedures that detect and prevent defective devices from reaching the market. As there is no one thing that can be done to ensure safety, the development of a device needs to be documented. This documentation help show that every effort has been made to ensure the safety of a device.

One of the ways that can be used to verify a devices safety is by certification. A company producing a PEMS could send their software, along with their process for creating it, to a certification company that can certify the safety of the product. The certifiers are able to trace through the development of the device and determine if the process for creating the device adheres to the strict safety requirements required.

The disadvantage of all these security features are that they greatly increase the time and effort required for developing the product. Naturally this leads to a higher end price that the end users are going to have to pay. This is an expected outcome, but it is far from being desirable. Users that require these devices might not be able to afford the end price and might have a lower quality of life. A potential solution to both of these problems, safety and cost, would be to use a formal language such as Event-B.

1.2 Event-B

Event-B is a modelling language that allows us to define a model and assert if the model upholds invariants that we define. We can define either Machines or Contexts. The context contain the static information of the system such as Sets, constants and axioms. The machines however define variables, events and invariants. The events modify the variables of the machine. While the invariants tells us something about the machine. Event can contain guards that prohibit the event from being called. By the use of guards, we can limit the availability of events. The limitations of the events means that we can add invariants into our model. For all invariants defined, proof obligations are created for all the events of the model. The proof obligations can only be discharged if we are able to prove that the given event upholds all the models invariants. For example, our model could define a variable "x". An invariant we have could state that a predicate "P" always holds for the variable "x" ie. "P(x)" is always true. To uphold this invariant, all events that modify "x" can have a guard that ensures that any new value for "x" will satisfy "P(x')". The invariants that we add can be directly related to the systems requirements. This means we can extract our models invariants from the systems requirements and check if our model satisfies them. If our model satisfies the invariants, we can say that we have proved that it is correct with regards to the requirements. Being able to write these invariants requires thought on how you model the system. Modelling decisions must be made that help write the invariants required. The benefit of using such a modelling language is that we can provide these models to a certification company. The certifier will be able to look at the model and have an easier job of verifying safety as invariants are provided and proved.

1.2.1 Code Generation

What is perhaps the strongest selling point of Event-B is the code generation. After having a complete model, we can transform the model into machine code. This process is for the most part automatic, but would normally entail some form of "gluing". Unlike some modelling languages where the model and machine code are two completely separated entities, they have a close relation in Event-B. This gives us more confidence that the invariants in the model are also present in the machine code.

The hope of using Event-B will hopefully mean that the process of creat-

ing new PEMS will be simplified. Reduced cost while maintaining the safety of the system. However, creating a Event-B model is a non-trivial problem. Many decisions made during modelling require a deep understanding of the system being modelled. It is not uncommon to restart the process of modelling a system. This is often due to initial abstractions of the system not being easily refined into more concrete events.

1.3 Co-modelling - Discrete and Continuous

There exist two types of systems: discrete and continuous systems. An easy way to distinguish the two is to remember that discrete machines can be represented with finite number of bits. For example a boolean that is either 0 or 1 is a discrete variable. Throwing a 6-sided dice is also discrete as there are exactly 6 possibilities. Continuous variables however require an infinite amount of bits to represent. Typically these are variables that deal with real world physics. As an example, imagine a bucket of apples. We have two pieces of information from this bucket: the number of whole apples and their total weight. We can imagine that we have 10 apples and they weigh a total of 1 kg. However, it is highly unlikely that we have exactly 1 kg of apples. We could have 1.1 or 1.0000001 kgs of apples. However, we can be certain that we have exactly 10 whole apples.

Co-modelling is the combination of using a discrete system together with a continuous one. Typically, some electronic controller is the discrete system while the physical components or environment is the continuous one. We can imagine a temperature sensor connected to a controller that is reporting the temperature. The sensor is constantly, continuously, sending new readings to the controller. The controller however only reads the new temperature at some interval.

A insulin pump is connected to a patient and regulates the level of insulin in the blood. This pump is required when a patients body is no longer able to successfully regulate its insulin levels itself. This means that the insulin pump must be able to inject insulin whenever it deems it necessary. As with all systems, errors can and do occur. However, errors in a PEMS are unacceptable and must be avoided at all cost.

2 Definitions

Machine ==¿ Hemodialysis machine

3 Hemodialysis Machine

Cleans blood

3.1 Kidneys

Remove toxins from the blood, generates urine.

3.1.1 Failure

Unable to filter the blood stream.

3.2 Hemodialysis treatment

Replacement for the kidney, simulates a functioning kidney.

4 Variables

This section will detail various techniques that have been used to extract the required Event-B variables from the requirements.

4.1 Variable categories

In the case of the hemodialysis machine, there are a large amount of variables. This can be confirmed by reading the case study by in [x]. The variables required can be grouped into different categories. Categorising these variables can greatly help in improving our knowledge of the system. The concrete definition of the categories are left informal. We want our

Controlled are variables that the system has full control over

Monitored are variables that can only be observed and not directly modified.

User Input are monitored variables inserted by a nurse. Strictly $UserInput \subseteq Monitored$

Timed are monitored variables that have a dependency on time. Strictly $Timed \subseteq Monitored$

The two major categories of interest are the *controlled* and *monitored* variables. The User Input category is created to separate the internally monitored variables from the variables set by a nurse. This is done because the list of variables added by a nurse is such a large percentage of total variables. We have also decided to separate the monitored variables that include an element of time. As Event-B does not directly support time, we will extract them to their own category. These timed variables will be added at the very last machine refinement.

An example of a *controlled* variable is our blood pump. The software is in full control of the state of the pump. An example of a monitored variable is the *blood flow direction*. Although we directly control the pump, we do not control the blood flow direction. A *controlled* variable modifies the *environment* that again modifies the *monitored* variables. In other words, the system is unable to directly control the blood flow. An argument could be made that the if the pump is on, it is guaranteed that the blood flows rotation is positive. This is however making assumptions about the system that we do not know. In any case, our paper will define *controlled variables* as the variables the system has full and direct control over.

The idea of splitting variables into Controlled and Monitored came from reading [x]. Although Parnas presented this as a minor point in his paper, it is an interesting approach with regards to Event-B. This list is not an exhaustive list as other machines may require additional categories.

5 Machine Refinements

A defining characteristic of Event-B is its refinement of machines. At the start of the modelling process, we have our most abstract representation of the system. This abstract machine is then refined into a new machine that contains more details. The choice of the details is upto the modeller. Refinements usually fall within two categories; either a feature extension or a data refinement. Feature extensions usually implies that a machine has additional events or variables introduced. A data refinement is, as its name

implies, refining the data of a machine. It is used when we have defined some data very abstractly and want to refine it to be more concrete. For example, a traffic light might initially only be either on or off. We can later refine it to have red and green lights. We can link these variables together by adding an invariant that shows their equivalence. This is known a gluing invariant as it is gluing/connecting invariants.

$$abstractLight = ON \Leftrightarrow concreteLight = \{GREEN\}$$

$$abstractLight = OFF \Leftrightarrow concreteLight = \{RED\}$$

In this example, when the *abstractLight* is turned on the *concreteLight* must be GREEN. While OFF must be RED.

The choice of both abstraction level and incremental refinements is one that should be taken with great care.

6 EventB Models

– Notes on Event-B machine –

6.1 HD2

Ignore all values, focus only on the very abstract level. Don't have physical entities, like pumps that can be on or off. HDM1 - Patient blood never goes below 1.

Fails at refinement 2. Can not clean the blood before giving it is pumped back to the patient.

HD3

HDM10 If bloodpump in on, the system increases its amount of blood currently inside the system. Blood inside the system can be cleaned if the ultrafiltration pump is turned on and there is blood in the system. Cleaned blood is removed from the system. IN real life it is returned to the patient.

HDM100 For any reason, we may turn off the Blood pump

HDM1000 Can only turn on blood pump when the alarm is off. Alarm can be turned on for any reason

HDM10000

Need a function that takes a SystemState and return its substates. Cleaning, connecting etc. Added SystemPhaseCtx for the 3 system phases and new

csystemPhase variable. Also added an event to change a the variable to different phases. Added events that contains monitored variables that cause the blood pump to stop and activate an alarm. For now, the alarm is turned on and the BP can only be turned off when the alarm is on.

SystemPhaseCtx1 Added the SUBPHASES of the HD treatment. SUBPHASE in SystemPhases longrightarrow SubPhases

SystemPhaseCtx1 + HD100000 Added SystemSubPhases to the machine that allows for changing of the software state, only forwards.

Subphases are given an order and they must follow the order when the system changes states.

HD10000000 BloodPump and requirements regarding the System state has been added.

HD10000001 Added SAD events. Air volume limit, flow through the SAD.

Refactored all the machines to remove UFPump and blood pump variables. BP in now abstracted.

HD10000002 UFPump causes DF to flow and create pressure that removes toxins and water from the blood. Waste bucket contents - Used DF = water extracted.

Main problem: I do not know how the dialysis machine truly works. Why they use certain components.

HD10000003

Added bypass chamber. Left and right side of a chamber.

HD10000004 Adding Dialyzer.

Dialyzer has inner tube with blood, and the other tube with dialyzing fluid. The blood tube is encased inside the dialyzing fluid.

Do I need to monitor that blood is in the tubes? The real system will not know. Only if the pump is on and that the VRD is detecting blood by the exit.

Note: Gradual implementation or Refined implementation: Can implement "isBloodPumpOn" and use as guard for new events(Gradual implementation). Or do it all in one single refinement that adds a guard to all events. End result is the same, but doing everything in one refinement seems like the correct approach.

To build a machine: Create a machine that has the required events and variables to perform a valid/safe dialysis treatment. This machine should be thought of as a state graph were events trigger transitions. Our exit nodes should contain at least one valid/safe and successful run of a treatment.

Further refinements should be dedicated to trimming the graph from nodes that are illegal w.r.t requirements.

The KEEP method is very hard to do. If there exists independent components, it might be easier. In this system, all the components are linked together. The state of one component has an effect on all other components (mostly). One could select the system that does not have any outbound dependencies, but this will require a full understanding of the system from the start. An incremental approach could allow for learning a system piece by piece.

New machines: HD4 Using prefix 'A' to say that an event or variable is Abstract. Currently means that the variable or event is either: Breaking some physical law or using abstract variables. Typically the event is receiving some amount of fluid from no physical location. This might make it easier to separate finished events with

HDM01 to HDM04 does not separate fluid going out and fluid going in.

HDM01_EBC01, HDM02_Dialyzer01, HDM03_Bypass01, HDM04_BalanceChamber - Pumps in and out.

HDM05_EBC02 - Split variable "total fluid" to cleaned and uncleaned.

HDM06_Dialyzer02 - Separated blood from dialysate.

HDM07_Bypass02 - Separated DFluid to two independent streams