Evaluation of a Guideline by Formal Modelling of Cruise Control System in Event-B

Sanaz Yeganefard
University of Southampton
United Kingdom
sanaz_yeganeh@yahoo.com

Michael Butler
University of Southampton
United Kingdom
mjb@ecs.soton.ac.uk

Abdolbaghi Rezazadeh
University of Southampton
United Kingdom
ra3@ecs.soton.ac.uk

Abstract

Recently a set of guidelines, or cookbook, has been developed for modelling and refinement of control problems in Event-B. The Event-B formal method is used for system-level modelling by defining states of a system and events which act on these states. It also supports refinement of models. This cookbook is intended to systematise the process of modelling and refining a control problem system by distinguishing environment, controller and command phenomena. Our main objective in this paper is to investigate and evaluate the usefulness and effectiveness of this cookbook by following it throughout the formal modelling of cruise control system found in cars. The outcomes are identifying the benefits of the cookbook and also giving guidance to its future users.

1 Introduction

Systems which consist of parts to interact with and react to the evolving environment continuously are known as embedded systems. They are complex and often used in life critical situations which means costs of their failure are usually high. Thus, reliability, safety and correctness in such systems are important [7, 11] and techniques such as formal methods can help to examine the behaviour of these systems in early development stages [12]. However the process of modelling itself can present considerable challenges and following modelling guidelines [8, 9] and patterns [16] can be helpful.

One of these is a set of guidelines, or cookbook, which has been developed recently for modelling and refinement of control problems in Event-B. Event-B formal method is used for system-level modelling. The main objective of this paper is to investigate how effective and useful having such a set of guidelines is by applying it to a real application. Cruise control system (CCS) is the chosen real application for the modelling, since the attempt of this cookbook is to outline the necessary steps of modelling an embedded system which consists of a controller, a plant and an operator. Also, in order to make this model a good example for the future users of the cookbook, some of the main points which helped us during the modelling are explained as tips.

This paper is organised into 5 sections. In Section 2 the background of this work is discussed. Here we look at Event-B and its tool Rodin. After that an outline of CCS and the cookbook are given. In Section 3 the modelling process is explained in more details. The most abstract level of the model is described in Section 3.1 and the refinement steps in 3.2 and 3.3. The refinement proofs related to these steps have been verified with Rodin tool. Finally in Section 4 and 5 we evaluate the cookbook and consider limitations and future work.

2 Background

2.1 Event-B and Refinement

Formal methods are mathematical based techniques which are used for describing the properties of a system. They provide a systematic approach for the specification, development and verification of software and hardware systems and because of the mathematical basis we can prove that a specification is satisfied

by an implementation [19]. The formal method used in our work is Event-B [5] which is extended from B-method [2]. It has simple concepts within which a complex and discrete system can be modelled. The main reasons for choosing Event-B are firstly that it is the language used in the cookbook and secondly, it has advantages such as simplicity of notations and extendibility and thirdly its tool support.

Structure of Event-B Event-B models consist of two constructs, *contexts* and *machines* [5, 13]. Contexts which specify the static part of a model and provide axiomatic properties, can contain the following elements: *carrier sets*, *constants* and *axioms*. Machines represent the dynamic part of a model and can contain *variables*, *invariants* and *events*. An event consists of two elements, *guards* which are predicates to describe the conditions need to hold for the occurrence of an event, and *actions* which determines how specific state variables change as a result of the occurrence of the event. A context may extend an existing context and a machine may see a context.

Event-B Tool Unlike programming languages, not all formal methods are supported by tools [19]. However, one of the advantages of Event-B is availability of an open source tool which is implemented on top of an extension of the Eclipse platform. This tool, known as RODIN, provides automatic proof and a wide range of plug-ins such as the ProB model checker and Camille Text Editor which were used in our work [1, 6, 13].

Refinement In some systems because of the size of states and the number of transitions, it is impossible to represent the whole system as one model. Here, refinement can help us to deal with the complexity in a stepwise manner by working in different abstract levels [19]. There are two forms of refinement; firstly, feature augmentation refinement (also known as horizontal refinement [10]) where in each step new features of the system are introduced. Secondly, data refinement (also known as vertical or structural refinement [10]) which is used to enrich the structure of a model to bring it closer to an implementation structure.

2.2 Cruise Control System

In order to have a better understanding of CCS an overview of its external observation is given in Figure 1 (based on [4]). This figure shows the relation between cruise control with the driver and the car. The role of the cruise control that we are interested in is maintaining the car speed as close as possible to the target speed which is set by the driver. The ways a driver and the car can interact with CCS are categorised as:

- Driver can switch CCS on or off, define a target speed for the system and increase or decrease this target speed. Also, the driver can regain the control of the car by using the accelerator, brake, or clutch.
- The car sends the actual speed as a feedback to the cruise control system.
- CCS signals the desired acceleration to the motor.

2.3 Overview of Cookbook

As mentioned the focus of the cookbook is on control systems which consist of plants, controllers and in some cases operators who can send commands to the controller (Figure 2[1]). The modelling steps suggested in the cookbook are based on the four-variable model of Parnas [18] and can be divided into

[1] The diagram uses Jackson's Problem Frame notation [15].

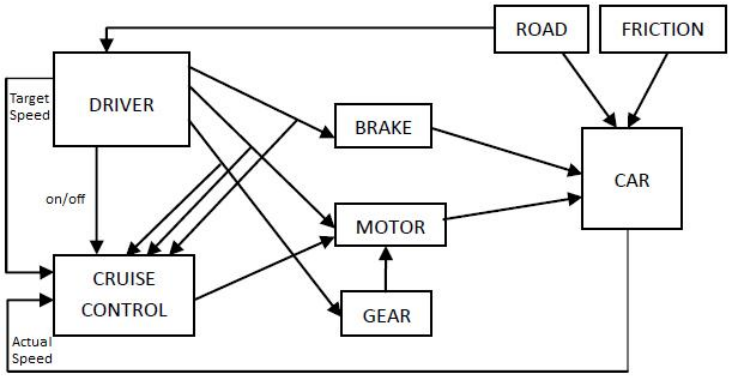


Figure 1: Interaction between driver, cruise control and car (based on [4]).

two major categories. Firstly, identifying the phenomenon in the environment and secondly representing the phenomena used for interaction between the controller and the environment (plant and operator) [8, 9].

Variables shared between a plant and a controller, labelled as 'A' in Figure 2, are known as environment variables [8] and are categorised into *monitored variables* whose values are determined by the plant and *controlled variables* whose values are set by the controller. There are also *environment events* and *control events* which update/modify monitored and controlled variables respectively. Also, in order to add design details the cookbook defines two steps of vertical refinement for introducing internal controller variables. Firstly adding *sensed variables* which defines how a controller receives the value of monitored variable. Secondly, adding *actuation variables* which sets the value of controlled variable [8, 9].

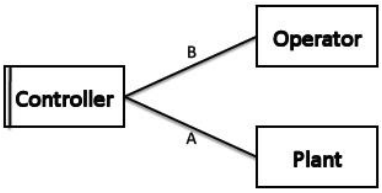


Figure 2: A control problem system [8, 9].

In addition to the variables introduced in four-variable model, the cookbook suggests the identification of the phenomena shared between controller and the operator in the cases where the system involves an operator, labelled as 'B' in Figure 2. These phenomena are represented by *command events* which are the commands from an operator and *commanded variables* whose values are determined by command events and can affect the way other events behave. The refinement step for defining when a controller receives a request from the operator is through introducing *buttons* [8, 9].

3 Cruise Control Model

The process of modelling CCS is divided into 3 main sections; First of all, in Section 3.1 M0 as the most abstract machine and C0 as the first context are defined. In order to do this we start with identifying the monitored, controlled and some of the commanded variables and also their corresponding events as suggested in the cookbook. In Section 3.2 we discuss horizontal refinements which represent the requirement specification of the system by introducing the remainder of the command events and commanded

variables. Lastly, in Section 3.3 vertical refinements are defined to introduce the design steps through sensing, actuation and operator requests.

3.1 M0 and C0

The process of modelling in the most abstract level starts with identifying the monitored, controlled and commanded variables. As was mentioned the role of the CCS which we are interested in is maintaining the actual speed of the car as close as possible to the target speed by controlling the acceleration of the car.

Monitored Variable and Environment Event Based on the role of the CCS, we identified the actual car speed as the monitored variable. This variable is represented by *sa*. Since the value of *sa* cannot be bigger than the maximum car speed, a constant named *n* was defined to represent the maximum car speed ($n \in \mathbb{N}$). Therefore, *sa* can be defined as $sa \in 0..n$. It is also necessary to add the environment event *UpdateActualSpeed* which can update or modify the value of the monitored variable *sa*.

Tip 1: The source which determines the value of a variable is important and a variable will be categorised as monitored, if the environment determines its value.

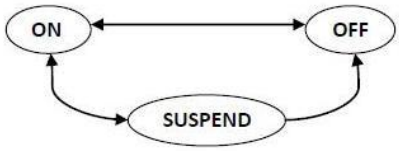
Commanded Variables and Command Events One of the identified commanded variables is *st* which represents the target speed determined by the driver. Based on the requirement, the target speed must be within a specific range. To model this the two *lb* and *ub* constants were defined to demonstrate the minimum and maximum of target speed. This results in having the invariant $stFlg = TRUE \Rightarrow st \in lb..ub$, showing that when *st* is defined it must be within the accepted range. The variable *stFlg* shown in the previous invariants is a boolean variable which turns to TRUE when *st* is set by the driver ($stFlg \in BOOL$). This flag is defined to represent whether the target speed has been defined or not. This is necessary as the CCS will control the car speed only when it has been switched on ($status = ON$) and the target speed has been defined ($stFlg = TRUE$).

The other commanded variable is named *status*. This variable is an element of a set called STATUS and shows the current status of CCS ($status \in STATUS$). The set STATUS represents the three possible statuses that CCS can be in at one moment of time. These three statuses are ON, OFF and SUSPEND which means CCS does not control the car speed as the result of the driver using one on the pedals. This is discussed in more details in Section 3.2.2.

Tip 2: Variables such as target speed can also be seen as monitored variables. However, we suggest defining them as commanded variables. This is because firstly these variables are internal to the controller and secondly their values are determined by an operator rather than environment.

Corresponding to these commanded variables the following command events were defined:

- *SetTargetSpeed*: when cruise control is on, driver can set *st* to the actual speed of the car.
- *ModifyTargetSpeed*: modifying *st* when CCS is on and *stFlg* is TRUE. Notice that at this level of abstraction we do not separate increment and decrement.
- *StatusOn, Suspend, and SwitchOff*: update *status* variable. In this level of abstraction we consider the relation between different values of *status* variable regardless of what causes changes (Figure 3 (a)). For instance, it is possible to get to ON from status OFF. This is shown in Figure 3 (b) where we added the guard *grd1* that *status* can be OFF and the action is status := ON.



(a)

```
StatusOn in M0
event StatusOn
  where
   @grd1 status ∈ {OFF, SUSPEND}
  then
   @act1 status ≔ ON
end
```

(b)

Figure 3: (a) The relation between different statuses of the cruise control system and (b) event StatusOn.

Controlled Variable and Control Event Based on the role of the CCS we identified *acceleration* as the only controlled variable ($acceleration \in \mathbb{Z}$). Also, in order to update its value, we added the control event *UpdateAcceleration*. To do this a function named *accFun*, which returns the value of *acceleration* based on actual speed and target speed, is defined. This is a total function since there must be a value defined for every tuple of *st* and *sa* ($accFun \in lb..ub \times 0..n \rightarrow \mathbb{Z}$). Therefore, the action which sets the value of *acceleration* is defined as $acceleration := accFun(st \mapsto sa)$.

Tip 3: The source which determines the value of a controlled variable is important and a variable will be categorised as controlled, if the controller determines its value.

3.2 Horizontal Refinements

In order to simplify the refinement steps, firstly all horizontal refinements were done and then vertical refinements. Horizontal refinements resulted in adding context C1 and machines M1 to M3 to the model where every machine refines its previous machine and sees the extended context C1. In machine M1 the action of increment and decrement of variable *st* are separated. According to the requirement document driver can interact with CCS through the pedals. The pedals are introduced to the model in machine M2. In the final step, the horizontal refinement gear is added.

Tip 4: Separation of horizontal and vertical refinement simplifies the process of modelling.

3.2.1 C1 and M1

The context C1 extends C0 in order to add the constant *unit* which defines the value added to/subtracted from *st* to increase/decrease it. It was possible to define this constant in C0 and eliminate C1. However, we prefer to have a more readable model. Based on this constant, in machine M1 the *ModifyTargetSpeed* event is refined to two separate increase and decrease events.

3.2.2 M2

In this machine pedals which are a way for a driver to interact with CCS [4] are introduced to the model. Notice that all the variables introduced in this section are commanded variables and in the chain of horizontal refinements we only deal with commanded variables and command events.

Based on the requirement, pressing the accelerator, when CCS controls the car speed, will suspend the system temporarily and when the driver stops using this pedal, the CCS will regain the control of the car speed. However, using either brake or clutch causes permanent suspension and this suspension can be resumed by the driver. To distinguish the two types of permanent and temporary suspension, we defined a variable which shows sub-states of the super status SUSPEND. This variable, named *permanentSusp* is of type of boolean and is set to TRUE when CCS is permanently suspended. The other approach would be to introduce a new set consists of $\{$on, off, permanentSusp, tempSusp$\}$ which refines the set STATUS.

However this can make the process of refinement proof slightly more complicated. Also, we defined two boolean variables *brkCltchPdl* and *accelerationPdl* to represent the relevant pedals. These variables are set to TRUE when their relevant pedals are pressed.

In the first modelling attempt we defined four events to represent actions of pressing and releasing of each accelerator pedal and brake/clutch pedals when CCS controls the car speed. In order to update the value of *status*, some of these events must refine one of the events *StatusOn* or *Suspend* from M0. Firstly, the event *PressAcceleratorPdl* refines *Suspend* and causes the value of *status* to change from ON to SUSPEND. Also, event *StopAcceleratorPdl* which represents the release of the accelerator refines *StatusOn* and returns the value of *status* to ON. In order to ensure this event can happen only when CCS is temporarily suspended, a guard showing *permanentSusp* must be FALSE is added to the event.

In a very similar way event *PressBrkCltchPdl* refines event *Suspend* and causes the value of *status* to change from ON or SUSPEND to SUSPEND. Notice that because pressing the brake/clutch is stronger than pressing the accelerator, using the brake or clutch while accelerator is pressed must cause permanent suspension. Finally, event *StopBrkCltchPdl* does not change variable *status* and only updates the variable *brkCltchPdl*. This is because CCS stays suspended when the driver releases brake/clutch. The other event which was defined to model the resume of the CCS from permanent suspension is *Resume*. This event allows the CCS to regain the control of the car speed by changing variable *status* to ON and variable *permanentSusp* to FALSE. Also this event can only happen when CCS is permanently suspended. Therefore, the flag *permanentSusp* must be TRUE in order for this event to be enabled.

3.2.3 M3

The last machine of horizontal refinements is M3 where the gear is introduced, because CCS can be switched on only when the vehicle is in second or higher gear. This is modelled by adding variable *gear* whose type is a number from -1 to 5 where -1 represents reverse gear, 0 neutral and 1 to 5 represent first to fifth gear. Also, event *ChangeGear* is introduced to be able to change the value of *gear*.

3.3 Vertical Refinements

M3 was the last machine of feature augmentation (horizontal) refinements. The remainder of our model consists of machines M4 to M6 which represent the added design details to M3. In the same way as horizontal refinement every machine refines its previous one and sees the context C1. These structural (vertical) refinements are based on the cookbook [8, 9].

The first vertical refinement suggested in the cookbook is to introduce sensors through which the controller receives the value of monitored variable. This is modelled in machine M4 by defining an internal variable which gets the value of *sa* by sensing it through an event. In the same way, CCS has an internal variable which sets the value of controlled variable. This internal variable acts as an actuator for the controlled variable. This design detail is introduced in machine M5. Finally, M6 represents the design of buttons through which the driver sends a request to the CCS. According to the cookbook every button is modelled as a boolean variable and the action of pressing that button is modelled as an event. We will discuss these refinements further in the remainder of this section.

Tip 5: Introduce each of the three steps of vertical refinement in a separate machine in order to have less complicated Event-B machines and consequently less proof obligations in each step.

3.3.1 M4

CCS receives the value of a monitored variable through sensors. The sensed/received value needs to be defined as an internal variable for the CCS in order to distinguish values of a sensed variable and a

monitored variable [8, 9]. We defined *sensedSa* as the sensed variable and an event called *SenseSa* which sets *sensedSa* to the current value of *sa*. Since the two monitored and sensed variables are not always equal, their equality is represented as a boolean flag [9]. This flag is called *sensFlg* and becomes TRUE when event *SenseSa* sets variable *sensedSa* to *sa*.

Tip 6: For every monitored variable, defining one sensed variable and one boolean flag is necessary. The sensed variable is of the same type as its corresponding monitored variable and usually is initialised to the same value as the monitored variable is initialised to.

Based on the cookbook, variable *sa* in the control event *UpdateAcceleration* was substituted with *sensedSa* (Figure 4 (a), @act1) in this refinement. Also, this event can only happen when *sensedSa* is equal to *sa* which is the reason for adding @grd2 in Figure 4 (a). The cookbook also suggests adding the invariant $sensFlg = TRUE \Rightarrow sensedSa = sa$ in order to ensure that when *sensFlg* is TRUE, *sensedSa* represents the value of *sa* [9]. This results in a proof problem in *UpdateActualSpeed*, since it can change the value of *sa* while *sensFlg* is TRUE. Therefore, it is necessary to add the guard *sensFlg = FALSE* to this event. Notice that we assumed in between the CCS sensing the value of monitored variable *sa* and setting the *acceleration*, the monitored variable does not change. This is an engineering simplification which helps us to reduce some of the complexity of the modelling of the system.

```
event UpdateAcceleration
refines
UpdateAcceleration
  where
   @grd1 status = ON
   @grd2 stFlg = TRUE
   @grd3 sensFlg = TRUE
  then
   @act1 acceleration ≔
     accFun(st↦sensedSa)
   @act2 sensFlg ≔ FALSE
end
```

(a)

```
event Update_actAcc

 where
   @grd1 status = ON
   @grd2 stFlg = TRUE
   @grd3 sensFlg = TRUE
  then
   @act1 actAcc ≔
  accFun(st ↦ sensedSa)
   @act2 actFlg ≔ TRUE

end
```

(b)

```
event
ActuatingAcceleration
refines
UpdateAcceleration
  where
   @grd1 actFlg = TRUE
  then
   @act1 acceleration ≔
         actAcc
   @act2 sensFlg ≔ FALSE
   @act3 actFlg ≔ FALSE
end
```

(c)

Figure 4: (a) Event UpdateAcceleration in M4, (b) events Update_actAcc in M5 and (c) event ActuatingAcceleration in M5.

3.3.2 M5

In this machine we discuss that CCS decides on the value of *acceleration* distinctly from actuating it [9]. Based on the cookbook, to do this we need to define an actuation variable and a boolean flag. These are named *actAcc* and *actFlg* respectively. The flag *actFlg* will be set to TRUE when the internal process of determining the value of a controlled variable is finished and this variable can be actuated.

Tip 7: For every controlled variable, defining one actuation variable and one boolean flag is necessary. Also, the actuation variable is of the same type as its corresponding controlled variable and usually is initialised to the same value as the controlled variable initialisation.

According to the cookbook, we also defined two events. Firstly, an internal event called *Update_actAcc*, to set the value of actuation variable *actAcc* and set the flag *actFlg* to TRUE (Figure 4 (b), @act1 and @act2). Secondly, *ActuatingAcceleration* which sets the value of *acceleration* to the value of the internal variable *actAcc* and turns *actFlg* to FALSE (Figure 4 (c), @act1 and @act3). This event refines the control event *UpdateAcceleration*, since it modifies the value of *acceleration*. Note that *ActuatingAcceleration* can happen only when the value of *actAcc* is decided by the controller, therefore the guard *actFlg = TRUE* is added to this event (Figure 4 (c), @grd1).

In addition to these variables and events, based on the cookbook it is necessary to define two invariants. The first invariant represents that in between control decision on the value of *actAcc* and actuation of *acceleration*, we assume *st* and *sensedSa* do not change. The second invariant shows that after actuation of *acceleration* the value of this variable and *actAcc* are equal. Although it is not mentioned in the cookbook, this invariant is only needed when the value of the controlled variable changes depending on its previous value as well as some other variables. Since this is not the case in this model, defining the second invariant is unnecessary.

Tip 8: Invariant actFlg = FALSE $\Rightarrow$ actAcc = acceleration mentioned in the cookbook is unnecessary if the value of controlled variable is set independent of its previous value.

3.3.3 M6

In this section the operator's command request and CCS's response are distinguished by introducing buttons. According to the cookbook, a boolean variable representing a button, needs to be defined for every command event. Also the action of pressing a button should be introduced through an event which sets the button variable to TRUE [9]. We introduced the followings as buttons: *switchBtn, setBtn, incBtn, decBtn* and *rsmBtn*. Because CCS responds to a request only after the relevant button has been pressed, a guard which requires the relevant button to be TRUE is added to each command event. Also, CCS turns the relevant button back to FALSE when it responds to the request. Notice, there is no button defined for the command events related to pedals, since pedal variables *brkCltchPdl* and *acceleratPdl* in Section 3.2.2 count as buttons. Also, there are cases where CCS cannot respond to a coming request, for instance when CCS is OFF. These cases are not mentioned in the cookbook, but we prefer to model these situations as ignorance of CCS to the button's request.

Tip 9: For every button an event can be defined to represent cases where there should be no response to the pressed button. We add this event by defining its guards as the negation of the conjunction of all the guards in the command event corresponds to the button.

This is the last machine of our model and we have modelled the CCS based on the requirement document and the cookbook. In the remainder of this paper we reflect on the results of this work.

4 Results and Limitations

4.1 Evaluation of the Cookbook

The cookbook is mainly a guideline on vertical refinement. In addition to vertical refinement guidance, the cookbook suggests identifying monitored, controlled and commanded variables and their corresponding events at the most abstract level of a model. Once the variables are found, identifying events which modify and update them in horizontal refinements becomes straightforward. Also, the focus of the cookbook is mainly on the discrete aspects such as status, pedals and buttons and less on continuous, since many of the complexity of the requirements are related to discrete aspects.

One of the other advantages of using cookbook is that almost all the necessary variables, events and invariants for every step of vertical refinement are described. This can be helpful for the designers with not a lot of knowledge on formal modelling in Event-B. In addition, some proof problems caused by the invariants mentioned in the cookbook can help to identify errors of the model. In our work, the process of modelling machines M4 to M6, which was done based on the cookbook, was reasonably easy. In particular, M4 where the sensor was introduced had the most effortless refinement. However, the cookbook lacks a means of dealing with some issues which can raise during modelling process, such as modelling ignorance of a button, mentioned in Section 3.3.3.

Finally, based on the achieved results we believe the main advantage of following the cookbook is the structure that it gives to the process of modelling and refinement. While deciding on how to organise the refinement steps is known as a source of difficulty in the usage of refinement [3], modelling a control problem domain based on the cookbook can help to identify the required steps of refinement quicker and easier. In the case of this work the steps of vertical refinements in machines M4 to M6 were decided purely based on the cookbook.

4.2 Limitations

The limitations of this work can be categorised into two types. Firstly, limitations imposed by formal methods themselves, although we should consider that gained benefits may outweigh these limitations. A detailed discussion is beyond the scope of this paper but as an outline one of the limitations is that models can only cover some aspects of a system's behaviours, because mapping formal model and the real world in limited [12]. The second types of limitations are what we have not considered in the process of modelling. First of all we have not considered fault-tolerance and failure of the hardware and it is assumed that the components do not fail. Also, timing and time constraints are not discussed. It is important to notice that our intention was not to prepare a model which is ready to be implemented. Therefore, such limitations will be considered in future work.

5 Related work, Future work and Conclusion

5.1 Related Work

One of the other approaches for development of embedded systems is Problem Frames (PFs) developed by Michael Jackson. This approach focuses on the separation of problem (what the system will do) and solution (how it will do it) domain. PFs describe any system engineering problem through the concept of a *machine* which is going to be design by a software application, *problem world* and *requirement* [17]. As part of the Deploy project a cruise control system was modelled using PFs. Here, the concepts of PFs are as followings: Machine is the cruise control software; Problem world is anything that cruise control software interacts with, such as pedals and driver; Requirement is controlling phenomena which otherwise would be controlled by the driver, here controlling the car speed [17].

The other work which is related to the cookbook is SCR (Software Cost Reduction) [14]. SCR is a requirement method for real-time embedded systems which is, in the same way as the cookbook, based on the four-variable model of Parnas [18]. As well as identifying the four variables of the Parnas model, SCR defines the following four constructs [14]: *modes* which represent states of a monitored variable; *terms* which are auxiliary functions defined to make the specification more concise; *conditions* to represent predicates and *events* to show the changes of the values in the model.

5.2 Future Work

The model of cruise control system represented in this paper contains the platform, the environment and the software application. Separation of these concepts through decomposition in later steps of design allows us to derive a specification of the control system, since the software and the hardware are being separated. In addition, other aspects of an embedded system are usually analysed and modelled through different techniques to formal methods. In order to ensure that cruise control system and other models of the system such as a model of car engine are consistent, meta-modelling can be used.

5.3 Conclusion

This work has achieved its main objectives in evaluation of the cookbook and preparation of the best design model for the cruise control system. We showed how the cookbook can make the process of modelling simpler and how it can help to find modelling errors. Also, the model of cruise control system represented in this paper and the given tips can be used by future users of the cookbook. We believe the outcomes of this work have contributed to the research in refinement-based methods such as Event-B and have the potential of leading to improved patterns and guidelines.

Acknowledgement: This work is partly supported by the EU research project ICT 214158 DEPLOY (Industrial deployment of system engineering methods providing high dependability and productivity) www.deploy-project.eu.

References

[1] Roadmap for rodin platform. `http://wiki.event-b.org/images/Roadmap.pdf`, 2009. cited: 2009 Sep.

[2] Jean-Raymond Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 1996.

[3] Jean-Raymond Abrial. Formal methods in industry: achievements, problems, future. In *ICSE*, pages 761–768, 2006.

[4] Jean-Raymond Abrial. Cruise control requirement document. Technical report, Internal report of the Deploy project, 2009.

[5] Jean-Raymond Abrial. Modeling in Event-B. To be published, 2010.

[6] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. An open extensible tool environment for Event-B. In *ICFEM*, pages 588–605, 2006.

[7] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to event-b. *Fundam. Inform.*, 77(1-2):1–28, 2007.

[8] Michael Butler. Chapter 8 modelling guidelines for discrete control systems, Deploy deliverable d15, d6.1 advances in methods public document. `http://www.deploy-project.eu/pdf/D15-D6.1-Advances-in-Methodological-WPs..pdf`, 2009. cited 2009 7th July.

[9] Michael Butler. Towards a cookbook for modelling and refinement of control problems. in Working Paper. ECS, University of Southampton, 2009.

[10] Kriangsak Damchoom and Michael Butler. Applying event and machine decomposition to a flash-based filestore in event-b. In *SBMF*, pages 134–152, 2009.

[11] Stephen Edwards, Luciano Lavagno, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. In *Proc. of the IEEE*, pages 366–390, 1997.

[12] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, 1990.

[13] Stefan Hallerstede. Justifications for the Event-B modelling notation. *Lecture Notes in Computer Science*, 4355:49–63, 2006.

[14] Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw. Automated consistency checking of requirements specifications. *ACM Trans. Softw. Eng. Methodol.*, 5(3):231–261, 1996.

[15] Michael Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[16] Xin B. Li and Feng X. Zhao. Formal development of a washing machine controller model based on formal design patterns. *WTOS*, 7(12):1463–1472, 2008.

[17] Felix Lösch. Chapter 6 problem frames, Deploy deliverable d15, d6.1 advances in methods public document. `http://www.deploy-project.eu/pdf/D15-D6.1-Advances-in-Methodological-WPs..pdf`, 2009. cited 2009 3rd Aug.

[18] David Lorge Parnas and Jan Madey. Functional documents for computer systems. *Sci. Comput. Program.*, 25(1):41–61, 1995.

[19] Jeannette M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–24, 1990.