Project DEPLOY
Grant Agreement 214158

*"Industrial deployment of advanced system engineering methods for high productivity and dependability"*

# *DEPLOY Deliverable D19*

# D1.1 Pilot Deployment in the Automotive Sector (WP1)

## *Public Document*

28 January 2010

http://www.deploy-project.eu

## Contributors:

| | |
|---|---|
| Felix Loesch | Robert Bosch GmbH |
| Rainer Gmehlich | Robert Bosch GmbH |
| Katrin Grau | Robert Bosch GmbH |
| Manuel Mazzara | University of Newcastle |
| Cliff Jones | University of Newcastle |

## Reviewers:

| | |
|---|---|
| Michael Butler | University of Southampton |
| Elena Troubitsyna | Åbo Akademi University |

# Contents

**1 Introduction** **5**
- 1.1 Overview . . . 5
- 1.2 Outline . . . 6

**2 Pilot Description** **7**
- 2.1 Overview . . . 7
- 2.2 Functionality . . . 8
  - 2.2.1 System Modes . . . 8
  - 2.2.2 Control Interface . . . 8
- 2.3 Signal Evaluation Subsystem . . . 10

**3 Pilot Deployment Strategy** **13**
- 3.1 Overview . . . 13
- 3.2 Development process with Event-B . . . 13

**4 Pilot Deployment** **17**
- 4.1 Overview . . . 17
- 4.2 Informal Requirements Specification . . . 17
- 4.3 Semi-formal Requirements Specification . . . 18
- 4.4 Hazard Analysis . . . 20
- 4.5 Formal Specification . . . 22

**5 Results of Pilot Deployment** **27**
- 5.1 Overview . . . 27
- 5.2 Evidence . . . 27
- 5.3 Feedback to Methods . . . 29
  - 5.3.1 Requirements Engineering using Problem Frames . . . 29
  - 5.3.2 Mapping Problem Frames to Event-B . . . 30
  - 5.3.3 Requirements Traceability . . . 31
  - 5.3.4 Modelling of Control Systems in Event-B . . . 31
  - 5.3.5 Refinement Strategy and Decomposition . . . 32

5.3.6 Modelling the Flow of Events . . . . . . . . . . . . . . . 32
5.3.7 Modelling Structured Types . . . . . . . . . . . . . . . . 32
5.3.8 Summary . . . . . . . . . . . . . . . . . . . . . . . . . 32
5.4 Feedback to Tools . . . . . . . . . . . . . . . . . . . . . . . . 33
5.4.1 Rodin Platform . . . . . . . . . . . . . . . . . . . . . . 33
5.4.2 Text Editor . . . . . . . . . . . . . . . . . . . . . . . . 33
5.4.3 Automated Provers . . . . . . . . . . . . . . . . . . . . 33
5.4.4 Requirements Plugin . . . . . . . . . . . . . . . . . . . 33
5.4.5 Tool Support for Flow of Events . . . . . . . . . . . . . 34
5.4.6 Tool Support for Structured Types . . . . . . . . . . . . 34
5.4.7 Summary . . . . . . . . . . . . . . . . . . . . . . . . . 34

**6 Open Research Issues 35**
6.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
6.2 Linking Problem Frames and Event-B . . . . . . . . . . . . . . 36
6.2.1 The Bosch Approach - Open Issues . . . . . . . . . . . . 37
6.2.2 Jackson's diagrams extensions . . . . . . . . . . . . . . . 38
6.2.3 The UML-B way . . . . . . . . . . . . . . . . . . . . . 38
6.2.4 Comparsion of the Approaches . . . . . . . . . . . . . . 40
6.3 The HJJ Approach . . . . . . . . . . . . . . . . . . . . . . . . 40
6.3.1 The HJJ method and its Steps . . . . . . . . . . . . . . 41
6.3.2 LFTS and the Automotive Case Study . . . . . . . . . . 42
6.4 Future Strategy for Deployment . . . . . . . . . . . . . . . . . 44

**7 Conclusions 47**

**A Cruise Control - Semiformal Specification 49**
A.1 Identification of Abstract Context Diagram . . . . . . . . . . . 49
A.2 Elaboration of Context Diagram . . . . . . . . . . . . . . . . . 51
A.3 Identification of Subproblems . . . . . . . . . . . . . . . . . . 54
A.4 Projection into Subproblems . . . . . . . . . . . . . . . . . . . 54
A.5 Elaboration of Subproblems . . . . . . . . . . . . . . . . . . . 55

**B Cruise Control - Formal Specification 59**
B.1 Abstract Model . . . . . . . . . . . . . . . . . . . . . . . . . . 59
B.2 Refinement of Abstract Model . . . . . . . . . . . . . . . . . . 65
B.3 Decomposition of Refined Model . . . . . . . . . . . . . . . . . 69
B.4 Refinement of Signal Evalaution Model . . . . . . . . . . . . . 72

**Bibliography 76**

# Chapter 1

# Introduction

## 1.1 Overview

WP1 deals with the deployment of formal engineering methods (Event-B) in the Automotive Sector. The work package is lead by Robert Bosch GmbH in close cooperation with the following partners:

- Åbo Akademi
- University of Newcastle
- University of Southampton
- ETH Zürich
- CETIC

Our main objective in WP1 is threefold: (i) the deployment and detailed assessment of formal engineering methods in the context of automotive system development, (ii) the development of a methodology that is specific and applicable for automotive systems, (iii) the development of concepts for adaptation of our development process in order to efficiently use the methodology. Our objectives in detail are:

- Provide evidence that refinement-based formal engineering methods are applicable to Bosch systems. The key priorities for Bosch are:
  - Structured development of system requirements and systematic construction and validation of formal models from requirements
  - Effective reuse and evolution of formal models and analysis

  - Provide evidence of the applicability of formal methods to the development of automotive systems

- Develop a specific methodology for automotive systems and provide evidence for applicability by close-to-production implementation of relevant parts of the pilot application

- Identify changes to the current development process as well as concepts for assimilation

In order to achieve these objectives, the following deployment strategy has been pursued:

- Minipilot: The minipilot is a small Event-B model, focused on specific aspects (in the case of WP1, modelling of continuous behaviour and time)

- Pilot: The goal of the pilot is to develop a specific methodology for automotive systems including an industrial process for formal development (necessary for large scale deployment) as well as to provide evidences for sector acceptance (by developing a close-to-production implementation of relevant parts of the cruise control system)

- Enhanced deployment: The enhanced deployment will result in the application of the methodology in the context of other domains having different characteristics (e.g. air system of the engine control).

The purpose of this deliverable is to provide a description of the pilot, the pilot deployment strategy, the technical steps undertaken, and especially the results of the pilot deployment including evidence, feedback to methods and tools as well as an outline of future work towards full deployment.

## 1.2 Outline

The remainder of this deliverable is structured as follows: Chapter 2 gives an overview of the pilot application, namely the cruise control system, and describes its main functions. In Chapter 3 we explain the overall strategy for pilot deployment. Chapter 4 describes the technical steps undertaken for pilot deployment and Chapter 5 presents the results of the pilot deployment including evidence, feedback to methods and tools. Chapter 6 reports on open research issues for deployment of formal methods in the automotive sector. This deliverable is concluded by Chapter 7. Chapters 1-5 and Chapter 7 have been written by the deployment partner (Bosch). Chapter 6 has been written by the University of Newcastle.

# Chapter 2

# Pilot Description

## 2.1 Overview

For pilot deployment within WP1 we have chosen the *cruise control system*, an automotive system implemented in software which automatically controls the speed of a car. The cruise control system is part of the engine control software which controls actuators of the engine (e.g. injectors, fuel pumps, throttle valve, fuel pumps) based on the values of specific sensors (e.g. gas pedal position sensor, airflow sensor, lambda sensor).

Since the cruise control system automatically controls the speed of a car there are some safety aspects to be considered and it needs to fulfill a number of safety properties. For example, the cruise control system must be deactivated upon request of the driver or in case of a system fault.

In the following we briefly describe the main functionality of the cruise control system as well as the subsystem *signal evaluation* which has been selected for a detailed description in the pilot deployment report. It is important to note that we are focusing on the software part of the cruise control system for pilot deployment. The environment of the cruise control software, i.e., the engine control software, the sensors, the actuators, and the control interface are external to the cruise control software we want to develop formally. However, as the requirements of the cruise control system are stated in terms of aspects of the environment we still need to model the environment in our pilot deployment in order to verify and validate the requirements.

## 2.2 Functionality

### 2.2.1 System Modes

Essentially, the cruise control system can be in one of the following three modes.

1. **NO_CONTROL:** In this mode the cruise control is not controlling the vehicle speed. The vehicle speed is only controlled by the driver using the accelerator pedal. Based on the position of the accelerator pedal the engine control software calculates the required amount of fuel being injected into the combustion chamber. Due to the combustion of fuel, the engine is moving and provides a physical torque which directly influences the speed of the vehicle.

2. **CONTROL:** In this mode the cruise control system actively controls the vehicle speed. It is either maintaining a target speed defined by the driver or approaching a previously set target speed from above or from below.

3. **ACONTROL:** In this mode the cruise control system is either accelerating or decelerating the car with a predefined acceleration/deceleration.

The behaviour of the cruise control system is only determined by the three modes defined above. For the CONTROL and ACONTROL modes the cruise control system provides control algorithms designed by control engineers which control the acceleration demand of the cruise control system on the vehicle. The engine control software converts this acceleration demand into physical engine parameters (e.g. amount of fuel, injection parameters, amount of air) which will cause the engine to provide a physical torque. This torque is then converted by the powertrain of the car into a physical acceleration of the car.

The modes described above can be switched by the driver using the *control interface* or by the *control software* in case the control software detects an error. In this case the mode is always switched to NO_CONTROL.

### 2.2.2 Control Interface

There are two ways of a driver to control the behaviour of the cruise control system: (i) using the pedals to (temporarily) deactivate the cruise control system, and (ii) using the control elements provided by the operating lever. The control elements provided by the operating lever may vary slightly between car manufacturers. However, the functions that can be controlled using

the control elements of the operating lever (e.g. switching cruise control on or off, setting a target speed, resuming a previously set target speed, increasing/decreasing the target speed, and accelerating/decelerating) are always the same.

Using the pedals, the driver can influence the behaviour of the cruise control system in the following ways:

- **Brake Pedal / Clutch Pedal:** If the driver presses the brake or clutch pedal, the cruise control system is deactivated (NO_CONTROL mode) if it has been active before (CONTROL or ACONTROL mode).
- **Accelerator Pedal:** If the driver presse the accelerator pedal, the cruise control system is temporarly deactivated (NO_CONTROL mode) if it has been active before (CONTROL or ACONTROL mode). In this case the vehicle speed is controlled by the accelerator pedal position only. As soon as the driver releases the accelerator pedal, the cruise control system resumes the previously set target speed (CONTROL mode).

The basic cruise control operating lever provides an on/off button or switch, a set button, a resume button, a tip-up and tip-down button, and buttons to accelerate or decelerate manually.

- **ON/OFF:** The ON/OFF button or switch is used to switch the cruise control system to STANDBY or OFF, respectively. STANDBY and OFF are substates of the NO_CONTROL mode. As soon as the cruise control system is switched to STANDBY, it is ready to accept inputs from the control elements and pedals in order to influence the behaviour of the vehicle accordingly. Otherwise, the interaction of the driver with the control elements will be ignored. It is important to note that the ON/OFF button/switch does not change the cruise control mode from NO_CONTROL to CONTROL.
- **SET:** Pressing the set button results in the cruise control system saving the current vehicle speed as the target speed and switching mode to CONTROL if the system has been in NO_CONTROL mode. The defined target speed is maintained as long as possible and desired by the driver.
- **RESUME:** If the driver presses the resume button, the vehicle will accelerate or decelerate to a previously set target speed. In case there is no target speed set, the current vehicle speed will be taken as target speed and maintained. Pressing the resume button results in a switch

to the CONTROL mode if the current mode is NO_CONTROL and the cruise control has been switched ON before.

- **TIP-UP/TIP-DOWN:** If the tip-up or tip-down button is pressed, the target speed is increased respectively decreased by a predefined value. Pressing TIP-UP or TIP-DOWN does not influence the mode of the cruise control system.
- **ACC/DEC:** The driver may also manually accelerate or decelerate the car within certain limits using the ACC or DEC button, respectively. The cruise control system will switch to mode ACONTROL and remain in this mode until the ACC or DEC button is released or an error is detected.

Depending on the car manufacturer the operating lever may not provide all of the buttons/switches above. In this case the operating lever provides a reduced number of buttons. The corresponding cruise control function (e.g. SET, TIP-UP) is determined based on the button presses and the current mode and/or state of the cruise control system.

## 2.3 Signal Evaluation Subsystem

For the pilot deployment report we selected the *signal evaluation subsystem*. Besides the *signal evaluation subsystem*, the cruise control system consists of the *velocity control subsystem* which calculates an acceleration demand based on the current vehicle speed and the stored target speed and the *display subsystem* which is responsible for displaying the status of the cruise control system to the driver and to other control units. The *signal evaluation subsystem* is responsible for evaluating the signals generated by the control elements of the operating lever and by the pedals. Based on the evaluation of the signals the *signal evaluation subsystem* changes the internal state of the cruise control system and sets or deletes the target speed. Contrary to the *velocity control subsystem* the *signal evaluation subsystem* only consists of discrete behaviour and thus is especially amenable for formal modelling with Event-B.

As there exists a large number of operating lever variants we decided to choose one operating lever variant for pilot deployment which only provides the following control elements:

- ON/OFF Switch: The ON/OFF switch can be used to switch the cruise control system ON or OFF.

- SET / TIP-DOWN / DEC Button: When this button is pressed the cruise control system needs to determine whether to execute SET or TIP-DOWN based on the current state. The DEC function is being activated if the button is hold for a certain time.

- RESUME / TIP-UP / ACC Button: When this button is pressed the cruise control system needs to determine whether to execute RESUME or TIP-UP based on the current state. The ACC function is being activated if the button is hold for a certain time.

As the number of functions to be distinguished is higher than the number of provided buttons, it is obvious that signals generated by the control elements need to disambiguated in order to identify the function requested by the driver. This disambiguation is done in two separate steps:

1. Disambiguation by the low-level hardware driver: The hardware driver is able to disambiguate single button presses from holding a specific button. Thus the ACC and DEC functions can be disambiguated.

2. Disambiguation by the signal evaluation subsystem: The signal evaluation subsystem of the cruise control disambiguates signals generated by the low-level hardware driver based on the internal cruise control state.

The signal evaluation subsystem distinguishes between the following internal states of the cruise control system:

As shown in Table 2.1, for each mode there exists a number of different states which record the history of input signals which have been evaluated. It is important to note that state transitions within one of the three modes do not change the external behaviour of the cruise control. For example, if the cruise control transitions from state OFF to state STANDBY the invariant of the NO_CONTROL mode still holds. This means, that the vehicle speed is still only controlled by the accelerator pedal position. If the cruise control transitions from the state STANDBY to the state CRUISE which indirectly also involves switching modes from NO_CONTROL to CONTROL, then the externally visible behaviour of the cruise control is changing, i.e., after the state switch the vehicle speed is controlled by the cruise control and not anymore by the accelerator pedal position.

| **Mode** | **State** | **Description** |
|---|---|---|
| NO_CONTROL | UBAT_OFF | Ignition is off and engine not running |
| | INIT | Ignition is ON and cruise control is being intialized |
| | OFF | Ignition is ON, cruise control has been initialized and is switched OFF |
| | ERROR | An irreversible error has occurred |
| | STANDBY | Cruise control has been switched ON |
| | R_ERROR | A reversible error has occurred |
| CONTROL | CRUISE | Cruise control is maintaining the target speed |
| | RESUME | The target speed is approached from above or from below |
| ACONTROL | ACC | Cruise control is accelerating the car |
| | DEC | Cruise control is decelerating the car |
| | RAMP_DOWN | Cruise control is being switched off |

Table 2.1: Modes and states of the cruise control system.

# Chapter 3

# Pilot Deployment Strategy

## 3.1 Overview

Increasing comfort-, emission-, and safety requirements for future motor vehicles are leading to an increasing number of powerful and complex systems within a car. Currently interrelations have tightened between previously independent domains inside a vehicle. Therefore, reliability and safety of our overall systems are essential and have to meet highest standards. Dependability for Bosch means the absence of errors in all operating points as well as controlled behaviour in the presence of partial failures. Today, systematic testing is used to achieve dependability of our systems. However, the increase in system complexity means that the effort of systematic testing will grow exponentially and, hence, will become uneconomical. In order to face these challenges, we are convinced that dependability of our future systems can be ensured only by using formal engineering methods (quality by design). In WP1, we will deploy formal methods on the engineering of a cruise control system. A cruise control system is not the first target product for which formal methods are needed (A cruise control system is well know and was and is succesfull developed with traditional methods), but development of a cruise control system includes all technical challenges (i.e. a complex user interaction, variants, real time requirements, closed loop controller, ..) we have to deal with. Because of that a cruise control is an ideal pilot to learn how to develop a system with Event-B.

## 3.2 Development process with Event-B

System development with Event-B is quite a big change in the development process. Hence the strategy of introducing it should make transition from

the current development process to the new process as easy as possible. The used strategy is a cascade deployment starting from the requirements and going through the development life cycle to target code.

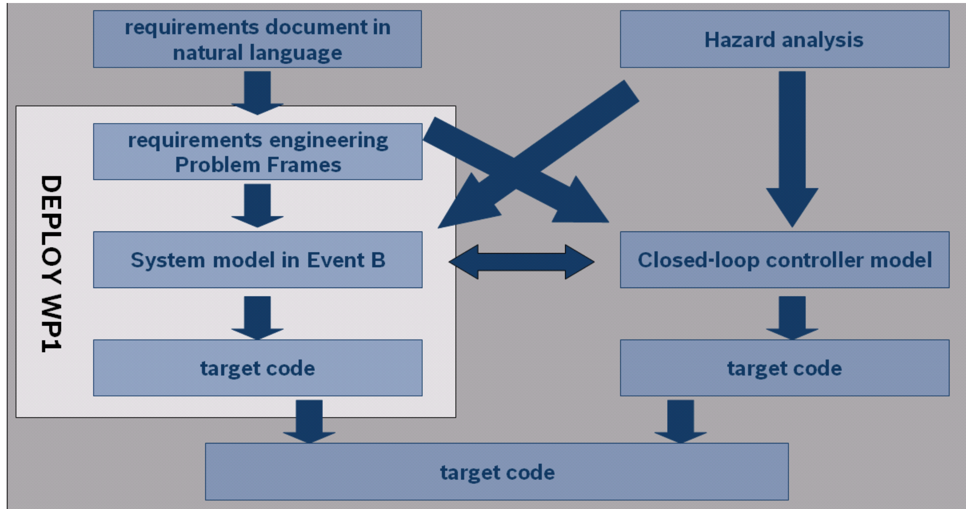


Figure 3.1: Overall development process.

In Figure 3.1 the embedding of the system development with Event-B in the overall development process is shown. In the beginning requirements engineering is done (see Section 4.2). The result of this process step is a requirements document usually in more or less natural language. Parallel to the requirements engineering process a hazard analysis is performed (see Section 4.4). Both process steps are not part of the WP1. In our pilot deployment we are basically using the results of these process steps (with some aditonal work done concerning the hazard analysis). Ideally the requirements engineering process produces functional requirements which are later transformed into events and the hazard anaylsis produces safety requirements which are transformed into invariants in the Event-B model.

Systems in the automotive industry are usually embedded real time applications which contain a closed loop controller as an essential part. The development of the closed loop controller is done by control engineers. Verifying closed loop controllers requires reasoning about continuous time behavior and this is not supported by Event-B. Therefore, we do not try to include detailed models of closed loop controllers Event-B. However, we do model the discrete part of the system in Event-B. The suggested development process thus is a parallel development of the discrete part of the system in Event-B and a development of the closed loop controllers with their well known methods and tools.

The open question at the moment is to which detail assumptions about the closed loop controllers must/should be included in the Event-B model.

The synthesis of the two development processes will be done on target code level (probably with one or more iterations). In a traditional validation process it has to be shown that the assumptions about the closed loop controllers we have inlcuded in the Event-B model are valid.

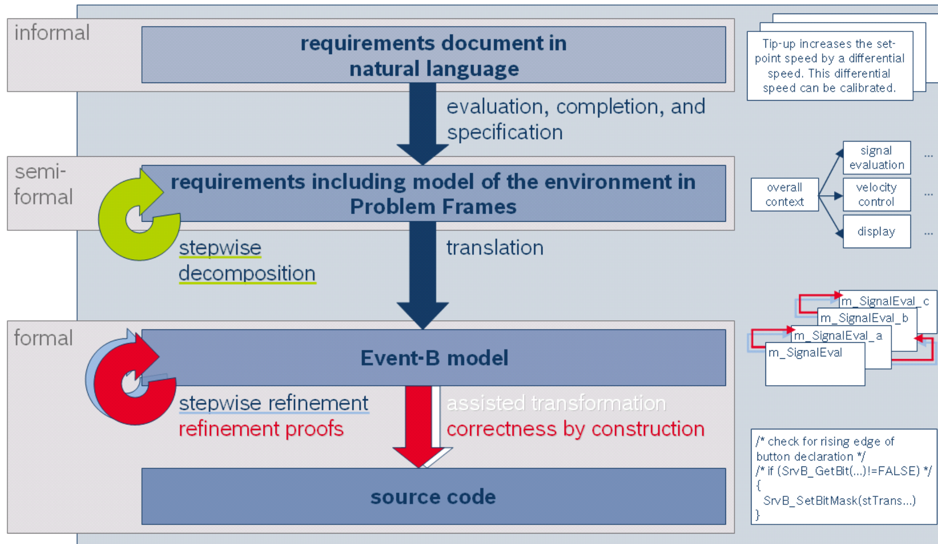


Figure 3.2: The deployment approach of WP1.

In Figure 3.2 the parts of the development process are shown which are of most interest in WP1. As already described above, the starting point is a requirements engineering phase (see Section 4.2). The result of this process step is a document in natural language which describes the functional requirements of the system. From the experiences we gained during the first two years, the gap between a usual requirements document and a Event-B model is huge. The problem of this gap is that you have to validate (with e.g. reviews) that the Event-B model is a correct model of the stated requirements. To overcome this problem we decided to introduce a pre or semiformal step in the requirements engineering process (for a detailed description see Section 4.3). We have chosen the Problem Frames method [Jac01] for this purpose. As described in the following chapters we extended the exsisting Problem Frame approach with the notion of hierarchy and refinment and will call them in the following extended Problem Frames. These extensions support the translation of an Problem Frame model in Event-B (see Section 4.5). Without the introduction of the extended Problem Frames approach we would have a problem in order to validate whether the Event-B

model is the correct model of the requirements. With the introduction of the Problem Frames approach we have two simpler validation problems: the first one is to validate whether the Problem Frames model is correct with respect to the requirments document and the second one is to validate whether the Event-B model is correct with respect to the Problem Frames model. From our point of view these two validation problems are easier to solve than the big one.

Besides the fact that in the graphical presentation of the development process only arcs in one direction are shown it is obvious that there are some iterations included in a real development process.

# Chapter 4

# Pilot Deployment

## 4.1 Overview

In this chapter we present a description of the pilot deployment, i.e., the modelling of the cruise control system in Problem Frames and Event-B. For each activity defined in Chapter 3, we describe the technical steps we performed.

The remainder of this chapter is structured as follows: Section 4.2 gives an overview of the starting point for pilot deyploment and describes the informal requirements specification as well as its weaknesses with regard to formal modelling. Section 4.3 presents the technical steps undertaken for deriving a semi-formal requirements specification of the informal requirements including the restructuring of the requirements. In Section 4.4, we describe our process of deriving safety requirements using a hazard analysis. We conclude this chapter by describing the technical steps for translating the semi-formal requirements specification into a formal specification in Event-B in Section 4.5.

The application of the technical steps described in Section 4.3 and Section 4.5 to parts of the cruise control system (signal evaluation) are described in Appendix A and Appendix B.

## 4.2 Informal Requirements Specification

The starting point for pilot deployment was an informal requirements specification of the cruise control system in natural language. We soon found out that the gap between the informal requirements specification and a formal model in Event-B is very large which makes it quite hard to validate whether the formal model in Event-B correctly fulfills the requirements stated in the informal requirements specification. Therefore we decided to introduce

an additional phase between the informal requirements specification and the formal modelling which is called semi-formal requirements engineering. Thus we are able to break the validation into two smaller steps. In the first step we have to validate whether the requirements stated in the semi-formal requirements specification fulfill the requirements in the informal requriement specification. In the second step we have to validate whether the formal model meets the requirements stated in the semi-formal requirements specification. The details of the semi-formal requirements engineering phase are described in Section 4.3.

## 4.3 Semi-formal Requirements Specification

The purpose of the semi-formal requirements specification is to describe the informal requirements of the cruise control system in a semi-formal way such that they can be easily modelled with Event-B.

During pilot deployment we decided to use the *Problem Frames approach* [Jac95, Jac01] and several extensions we developed for semi-formal requirements engineering because it allows to structure the problem into requirements, machine, and the physical world. Furthermore, the problem frames approach and our extensions allows us to decompose a given problem, e.g., the development of a cruise control system, into different subproblems which can be handled separately and later recombined in a subsequent phase. The development task in problem frames is to design a *machine* by building software that is then executed on a general-purpose computer, specialising the computer to serve a particular purpose [Jac95]. That purpose is to meet a *recognised need*, which is called *requirement*. Satisfying the requirement involves transforming the *physical world* around us. In Problem Frames, the part of the world to be transformed is called the *environment*. Jackson states that the parts of any *systems engineering problem* are the machine, the problem world (environment), and the requirement. Figure 4.1 shows a generalized Problem Frame diagram and its application to the cruise control system.

In our application of Problem Frames to the cruise control system, the *machine* we want to build is the cruise control software running on the engine controller. The physical world around us, i.e., the *problem world* the cruise control software is interacting with, consists of the *human machine interface*, i.e., the pedals, the lever, and the ignition, as well as the *vehicle* including engine, engine control, and wheels which need to be controlled by the cruise control. The *requirements* relate phenomena controlled by the human machine interface, e.g., control signals with phenomena controlled by

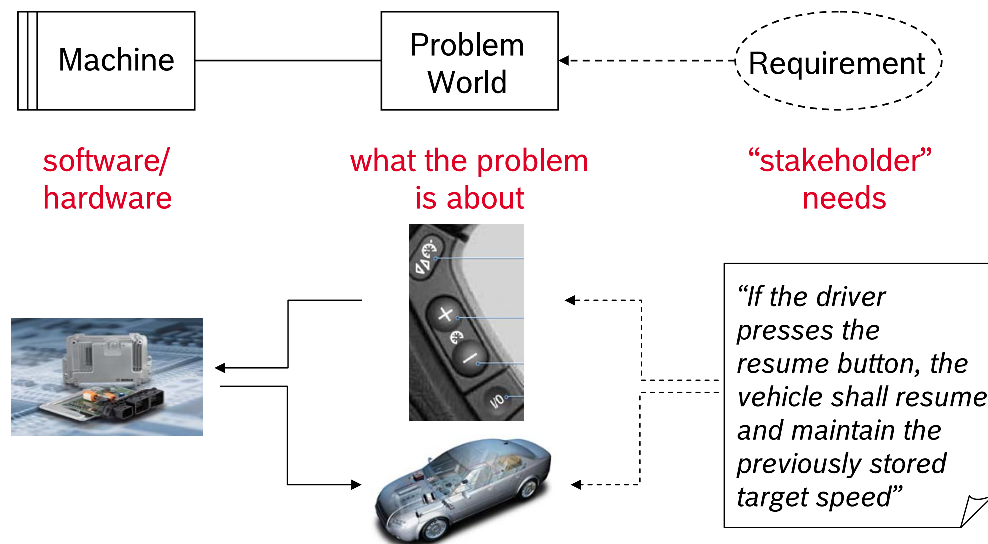


Figure 4.1: Problem frames and its application to the cruise control system.

the vehicle, e.g., vehicle speed.

The Problem Frames approach enforces that the terms used in the requirements are precisely defined using so called *phenomena* which need to be unique throughout the whole requirements document. Moreover, the Problem Frames approach helped us to clearly distinguish between the *requirements*, i.e., a description of the desired behaviour in terms of the environment, the *environment*, i.e., a description of the environment and its assumptions, and the *machine*, i.e., the actual cruise control system we want to built.

In order to derive a semi-formal requirements specification of the cruise control system we applied the process steps shown in Figure 4.2 which schematically describe our extended Problem Frames approach. Each process step answers specific questions.

The result of the semi-formal requirements specification is a hierarchy of *problem diagrams* which describe the requirements, the environment, and the cruise control system itself at different abstraction levels. Figure 4.3 shows the resulting hierarchy of problem diagrams of the cruise control system after the application of the process steps described in Figure 4.2.

As you can see from Figure 4.3 the hierarchy starts with the context diagram *Context 0* which describes the cruise control system, its environment, and the requirements from a very abstract point of view. In the first step, this abstract context diagram is *elaborated* into a more concrete context diagram (*Context 1*) which itself is *projected* into three different subproblems, namely *signal evaluation*, *velocity control* and *display*. In an *elaboration* of

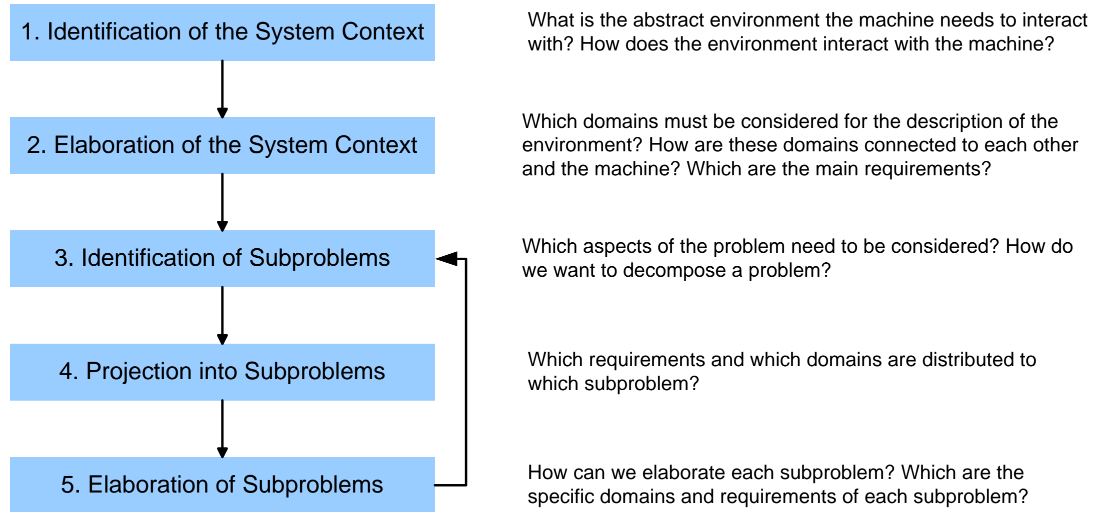


Figure 4.2: Semi-formal requirements specification process using Problem Frames.

a problem diagram, the environment and the requirements are refined, i.e., they are described in a more concrete way. During a *projection* of a problem diagram, the machine, i.e., the system we want to built is split into different aspects. A more detailed description of *elaboration* and *projection* can be found in [LGR09].

As you can see from Figure 4.3, an *elaboration* is followed by a *projection* which is then followed by an elaboration and so on. This process is iterated, until the system is fully described.

For a more detailed description the semi-formal specification of the cruise control, we refer the interested reader to Appendix A.

## 4.4 Hazard Analysis

The purpose of the *hazard analysis* is to analyse possible hazards that can occur when the functions provided by the cruise control system are used by the driver in order to derive safety requirements for the system which can then be proved in the formal specification using Event-B. To systematically analyze possible hazards and to identify safety requirements the following process is used:

1. Identify main functions of the cruise control system from the semi-formal requirements specification

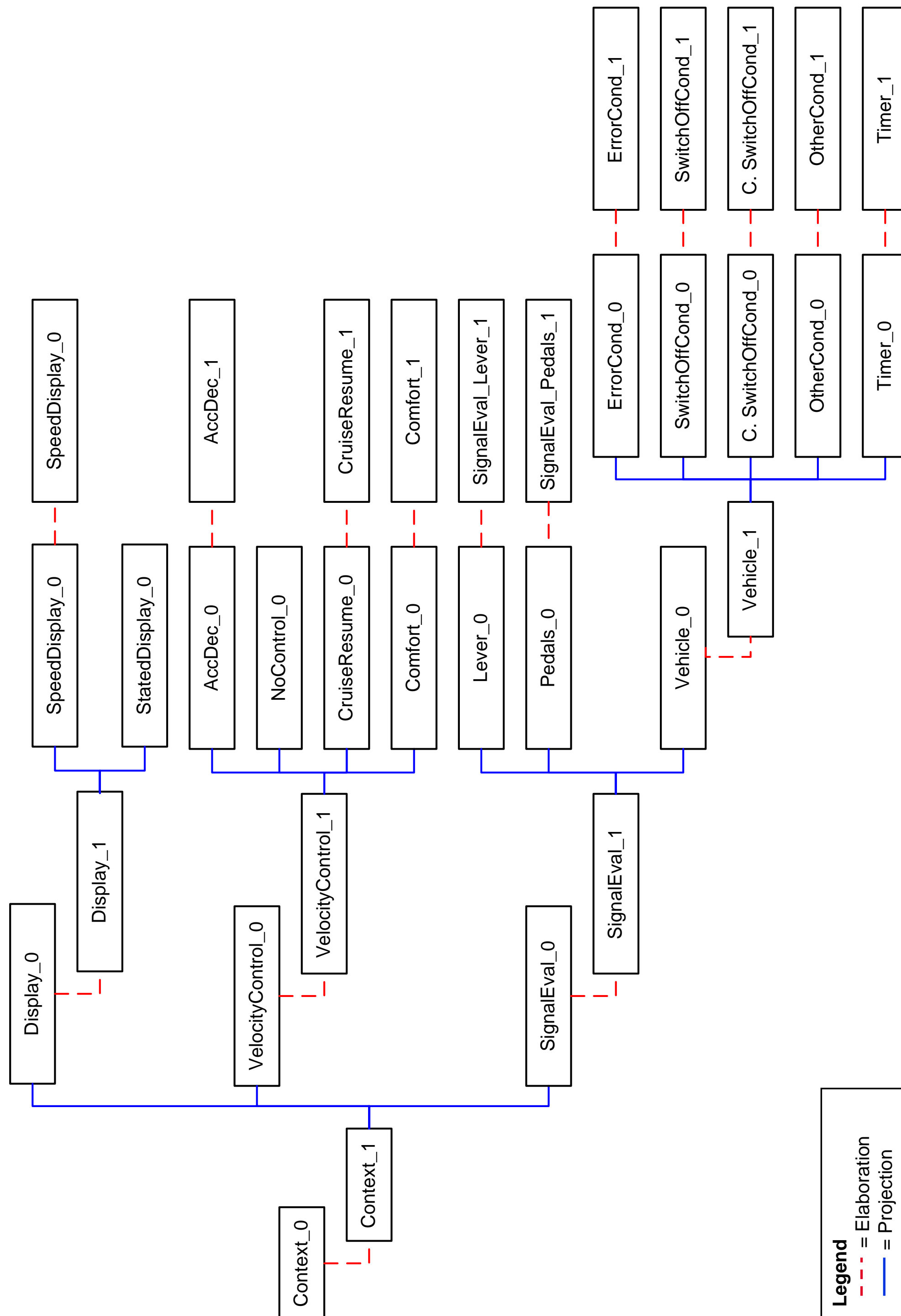


Figure 4.3: Hierarchy of problem diagrams for cruise control system.

| **Main Function** | **Hazards** | **Safety Requirements** |
|---|---|---|
| Pressing brake pedal deactivates CrCtrl (temporarly) | Pressing brake pedal does not deactivate CrCtrl | Brake pedal pressed implies no acceleration demand of CrCtrl |
| Main-switch OFF switches CrCtrl OFF | Main-switch OFF does not switch CrCtrl OFF | Main-Switch OFF implies no influence of the CrCtrl on the acceleration demand |

Table 4.1: Hazard Analysis Table

2. For each main function of the cruise control: identify and note down possible hazards / malfunctions that can occur

3. For each of the hazards / malfunction: identify safety requirements which will prevent the hazard

It is important to note that not every malfunction / hazard will be critical with regard to system safety. Furthermore, some identified hazards / malfunctions might be outside the system boundaries we are considering during pilot deployment. This is especially true for hazards / malfunctions which are caused by hardware failures since we are looking at the software part of the cruise control only.

Table 4.1 shows an excerpt from the hazard analysis table for some of the functionality provided by the cruise control system. Using the safety requirements shown in the last column in Table 4.1 we can derive possible invariants which can then be proven using the theorem provers provided by Event-B. For example, from the safety requirement *Main-Switch OFF implies no influence of the CrCtrl on the acceleration demand* we can derive the following safety invariant:

$$\texttt{inv1}: P_Env_ControlInterfaceSignals_MainSignal = FALSE \Rightarrow P_CrCtl_Acceleration = NO_ACCELERATION$$

If we can prove that this invariant holds we have proven that the safety requirement stated above holds in the model.

## 4.5 Formal Specification

The purpose of the formal specification phase is to model the cruise control system in Event-B [Abr09b]. Our input for a formal specification of the cruise

control system in Event-B is the semi-formal specification which has been derived from the informal specification by the process described in Section 4.3. During pilot deployment we developed several guidelines for constructing formal specifications in Event-B from semi-formal specifications in our extended Problem Frames approach which will be described in the following.

The first step in constructing a formal specification of a control system in Event-B is to think of the *refinement strategy*, i.e., a strategy about structuring the refinement levels of a formal specification in Event-B. Since Event-B supports so called *contexts* which describe the static aspects of the system to be modelled and so called *machines* which describe the dynamic aspects of the system to be modelled one has to think about a refinement strategy for *contexts* as well as one for *machines*. During our first experiments of formally modelling the cruise control system in Event-B we found out that our semi-formal requirements specification with different abstraction levels can be mapped to a formal specification in Event-B in the following way:

- Each *problem diagram* is modelled as a separate *machine* with its associated *context*.
- *Elaborations* of an abstract diagram into a more concrete one are realized in Event-B by *refinement* of the machine and its associated context.
- *Projections* of a problem diagram into two or more subproblems are realized in Event-B by *shared-variable decomposition* [Abr09a] with some changes.
- Each *phenomenon* defined in a problem diagram is modelled either as a *constant* or a *variable* in Event-B.
- Abstract phenomena which will be elaborated later are realized in Event-B using *records*.
- *Elaborations of phenomena* in problem diagrams are realized in Event-B using *data refinement*.
- *Requirements* stated in problem diagrams are realized in Event-B by *events* and/or *invariants*.

Table 4.2 shows this mapping of Problem Frame elements to Event-B elements in a compact form.

We applied this *refinement strategy* on the cruise control system. Figure 4.4 shows the refinement hierarchy of Event-B machines for the first levels of the problem diagram hierarchy shown in Figure 4.3.

| **Problem Frames** | **Event-B** |
|---|---|
| Problem Diagram | Machine / Context |
| Phenomenon | Variable, constant or predefined type (e.g. NAT) |
| Types of phenomena | Carrier set or constant |
| Elaboration of a problem diagram | Refinement of a machine or context |
| Projection of a problem diagram | Decomposition of a machine or context |
| Elaboration of phenomena | Data refinement of variables / defined types |
| Requirements | Events / invariants |

Table 4.2: Mapping of Problem Frame Elements to Event-B Elements

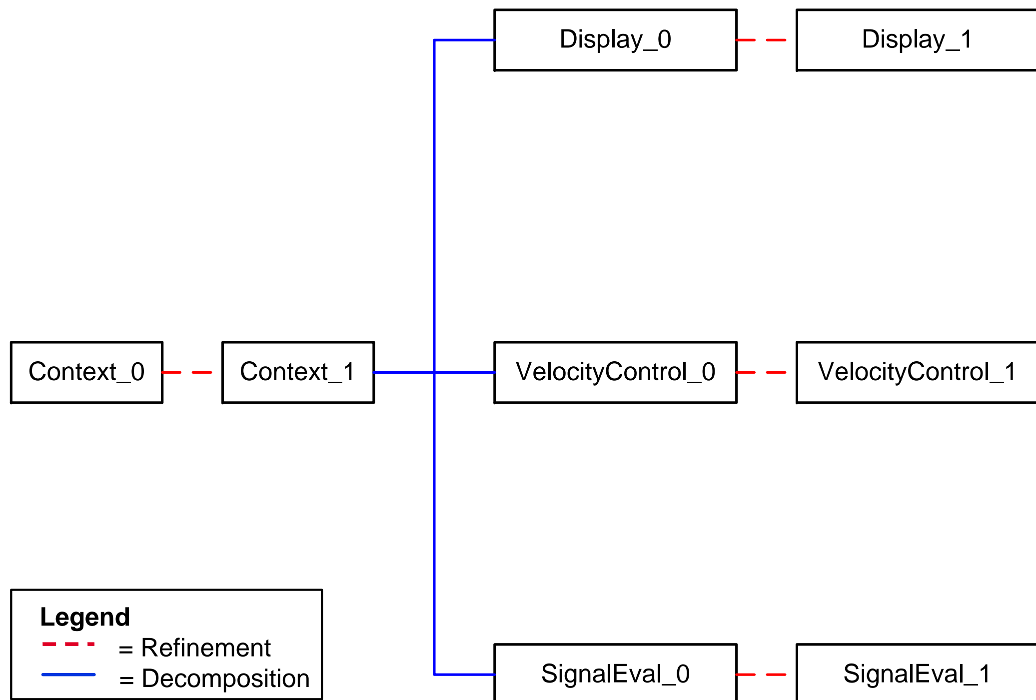


Figure 4.4: Cruise Control - Hierarchy of machines in Event-B.

As you can see from Figure 4.3 and Figure 4.4, each problem diagram is modelled as a separate *machine* in Event-B. Elaborations of problem diagrams are realized using refinement of the machine and projections of problem diagrams are realized by *decomposition* of the abstract machine in Event-B.

For a more detailed description of mapping problem diagrams to Event-B of the cruise control, we refer the interested reader to Appendix B.

# Chapter 5

# Results of Pilot Deployment

## 5.1 Overview

In this chapter we report on results of the pilot deployment and give feedback to the DEPLOY methods and tools.

In Section 5.2 we present evidence that the use of our semi-formal requirements engineering method is very helpful to close the gap between a requirements specification in natural language and a formal model in Event-B. However, the conrete mapping of extended Problem Frames to Event-B model elements (see Section 6) is not fully solved. Therefore, we need more experience with Event-B before we are able to report evidence about the use and deployment of Event-B.

Section 5.3 and Section 5.4 gives feedback on the DEPLOY methods and tools we applied during pilot deployment. For each method and tool we describe its strengths and weaknesses as well as open issues that need to be addressed for full deployment in the automotive sector.

## 5.2 Evidence

Due to the fact that we have worked most on the semi-formal requirements engineering process and have not finished the complete Event-B model of the complete cruise control we want to restrict the evidence results for this report on the use of Problem Frames and will postpone the results on evidence we gathered with Event-B.

To evaluate the extended Problem Frame method we measured several key parameters (number of modelled requirements, number of additional requirements, number of rejected requirements, effort spent). The starting point of the evaluation was the original requirements set for the cruise con-

trol. The aim of applying our method was to get a requirements set of the cruise control system which is complete, precise, consistent, hierachical and redundance-free. Furthermore, this requirements set should be amenable for formal modelling.

Because the original requirements set is not structured hierarchically, as a result of the requirements elicitation process the same functionality is getting described from different perspectives. This leads naturally to some redundant requirements.

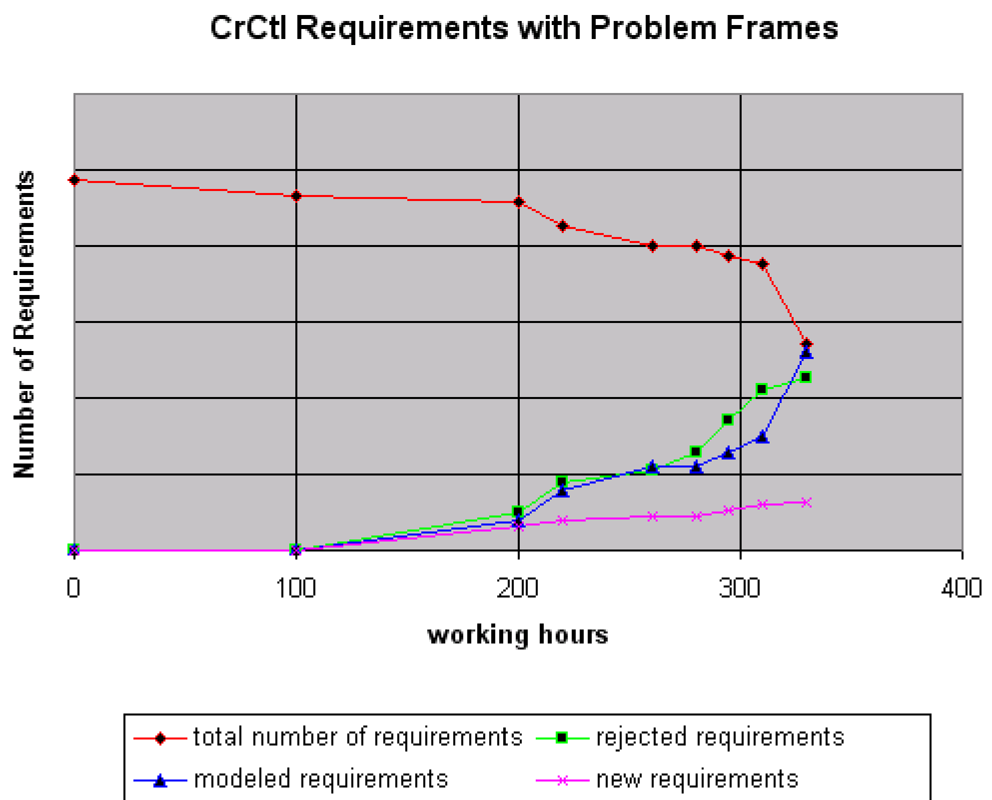


Figure 5.1: Quantitative results

Figure 5.1 shows the quantitative results of applying our requirements engineering method to the cruise control system. On the x-axis the total effort in working hours is shown. On the y-axis four curves are plotted. The first curve shows the total number of requirements as a refernce. This curve changes over time due to the addition of new requirements (shown in a seperate curve) and the rejection of original requirements (shown in the third curve).

In total we ended up with nearly half the number of text units (requirements) needed to describe the required functionality of the cruise control system. Thus, by applying our requirements engineering method, we were able to reduce the total number of text units (requirements) by more than 40 percent. The total effort we spent on the restructuring and improvement of the requirements amounts to approximately 300 working hours.

The evaluation results clearly show that by applying our structured requirements engineering method the understandability and coherency of the requirements can be increased while at the same time the total number of requirements can be reduced by more than 40 percent. Furthermore, our concept of hierachical requirements allows us to differentiate between a complete requirements set on a system level and a more detailed requirements set on a functional level.

As a result of this we are not always expecting to reduce the number of requirements by 40 percent. But we are convinced that we are usually able to reduce the number of requirements and to increase in the same step the quality.

## 5.3 Feedback to Methods

The general idea of this section is to give feedback to method-developers to improve their methods. In particular the strengths and weaknesses as well as open issues of the used methods are discussed.

### 5.3.1 Requirements Engineering using Problem Frames

A well-developed and structured requirements document which allows us to identify different abstraction levels is a key to successful formal modelling with a refinement-based method such as Event-B. Furthermore, the method used for requirements engineering shall provide means for handling large and complex systems by decomposing a problem into smaller parts and later recomposing these individual parts. Finally, it should support a semi- or preformal notation of requirements to make the step from informal requirements to a formal model manageable.

We decided to use Problem Frames for this process and found the following strenghts and weaknesses:

**Strengths**

- Clear separation of requirements, environment, and system to be built

- Supports elaboration and projection of the system into subproblems / different aspects
- Supports a description of requirements at different levels of abstraction
- Requirements are more precise than informal requirements

**Weaknesses**

- Sometimes requirements are still not as precise as needed for formal modelling
- The refinement strategy in Problem Frames is not necessarily compatible with Event-B (see 5.3.5)
- Separation of model information and requirements is not easy

Still an open issue is the question how to specify non-functional requirements like timing requirements. This should be addressed in the future.

### 5.3.2 Mapping Problem Frames to Event-B

With semiformal requirements in Problem Frames the mapping to Event-B is very promising because of the similar notation of refinement in Event-B and elaboration in Problem Frames approach. We have some guidelines for the translation (see Section 4.5) but there are still some challenges left, i.e., the problem of tracing the implementation of requirements in the Event-B specification.

Having started with Problem Frames, the use of abstract functions is quite usual. It is easy to state that two or more phenomena are in some kind of relation or influence each other. In Event-B modelling this kind of functions is painful because it complicates their refinement in more concrete models and makes the proofs very difficult. Therefore, we decided not to use abstract functions for modelling these kind of relations. Instead, we used non-deterministic assignments knowing that they have a weaker expressiveness than abstract functions.

Another open question is how to derive interesting invariants (besides the type invariants). In our pilot we derive invariants from the requirements in Problem Frames and from a hazard analysis. Especially if we want to systematically derive invariants from Problem Frames we need some form of guidelines to derive them.

### 5.3.3 Requirements Traceability

With the translation of semiformal requirements into a formal modal there is the challenge of traceability between requirements in Problem Frames and their implementation in Event-B. Some of the requirements in Problem Frames might be modelled as invariants, some might be guards of an event or actions. Some of them might not only occur at a specific point (such as an event) but could be spread over a whole model. A requirement in Problem Frames describes how a domain should be influenced under certain conditions by the machine which makes the mentioned possibility of spreaded requirements in the Event-B model more likely. There is methodogical work to do to solve this problem.

However, ordered and structured requirements at different levels of abstraction in a semiformal notation instead of completely informal requirements are helpful to achieve traceablity.

### 5.3.4 Modelling of Control Systems in Event-B

Although modelling in Event-B is usally straightforward there are some challenges we found modelling a control system:
There is the general idea that a change of the signals in the environment should lead to a reaction of the controller, which will effect the environment with actors. For this we need some kind of ordering of events. A simple nondeterministic choice of events in not sufficient for this kind of problem.

In order to generate C-Code from the Event-B model, it is required to have some kind of support for scheduling of events as well as the possibiliy to distinguish between the part of the Event-B model concerning the environment and the part concerning the actual machine.
These are strengths and weaknesses we found modelling a control system in Event-B:

**Strength**

- The *cookbook strategy* for modelling control systems can be applied in principle (see [But09]).

**Weaknesses**

- In order to prove safety invariants that result from the hazard analysis we need to define the order of events

- Specification of flow of events using auxiliary variables is very cumbersome (without adequate tool support)

- Timing issues and continuous behaviour (control theory, see 3.2) have not been included yet
- Need to distinguish between parts of the machine and parts of the environment in Event-B model

### 5.3.5 Refinement Strategy and Decomposition

The refinement stategy in Problem Frames is not necessarily the same as in Event-B, but we get some orientation of the existing refinements, which is quite helpful. It is difficult to decide on a refinement strategy before the actual work on modelling is performed.

The possiblity to decompose the model in Event-B is in particular required if the model is large and more than one person has to work on it. The two decomposition styles (A-style and B-style) are not sufficient for our model. We chose A-style decomposition, but if a variable is shared, it cannot be refined further. Although there is the obvious difficulty to synchronize such a decomposition we require to have support for it in order to model the cruise control system.

### 5.3.6 Modelling the Flow of Events

The possiblity of modelling the flow of events in not yet used, but as mentioned in Section 5.3.4 we need some kind of ordering events. Auxiliary variables are too complicated for large projects, so using flows could be a solution, but we will need tool support (see 5.4.5). The interesting question will be how and when to introduce flows in the model.

### 5.3.7 Modelling Structured Types

There is the need for refining data structures (records) during the development of the pilot. Therefore we would like to have records and tool support for data refinement (see Section 5.3.5 and Section 5.4.6), which is already work in progress.

### 5.3.8 Summary

Although there are some open questions and challenges left, the use of the general DEPLOY methods is very promising.

## 5.4 Feedback to Tools

The general idea of this section is to give feedback to the tool-developers to improve the tools. In particular strengths as well as weaknesses we found using the tools are discussed.

### 5.4.1 Rodin Platform

Using the RODIN Platform without any help is quite difficult. Some documentation is available but there is the strong need for further documentation and teaching material for the use of the tool. The available documents should be extended to ease the usage.

Furthermore we would like to have the possibility to add multiline comments which can be written between events. The current way of handling comments in which they only can be attached to Event-B model elements is too restrictive.

### 5.4.2 Text Editor

We found the text editor very helpful and prefer to use this representation, but the use of labels with their naming conventions can be improved. The text editor should generate labels for invariants, actions, guards etc. automatically. At least it should be possible to fix name clashes of labels in abstract machines and concrete machines via Eclipse Quick-Fix.

### 5.4.3 Automated Provers

For the automated prover extended teaching material is needed. A detailed description of when to use which prover and how to get hints from the proof tree is needed.

### 5.4.4 Requirements Plugin

Although there exists a requirements plugin for RODIN, the support for tracing requirements to model elements in Event-B is not yet sufficient. Furthermore, it would be very nice to have a flexible requirements plugin which can be used with our Problem Frames approach. Currently, the requirement plugin just supports natural language requirements which have to be provided by a simple list. Having a generic interface to different requirements engineering methods would be very helpful.

### 5.4.5 Tool Support for Flow of Events

As mentioned in Section 5.3.6 we will need tool support for the use of flows.

### 5.4.6 Tool Support for Structured Types

As mentioned in Section 5.3.7 we will need tool support for the refinement of structured types. It should be easy for the developer to enter structured types such as records. Any internal overhead dealing with the refinement of records, i.e., witnesses with accessor functions should be hidden from the user of RODIN.

### 5.4.7 Summary

The used tools are already quite good, but further development is needed. Especially extended documentation is important. A deployment may fail because of method problems, it may also fail because of tool problems, but it will definitely fail if there is no concept and no material for teching the method.

# Chapter 6

# Open Research Issues

This chapter reports open research issues arising from the pilot deployment as well as possible solutions developed by the University of Newcastle in the first 18 months of DEPLOY (mainly in the context of Task 8.1) and is written from the perspective of the academic partner.

## 6.1 Introduction

This chapter describes the major open research strands we identified so far for pilot deployment in the Automotive Sector, i.e. *Linking Problem Frames and Event-B* and the *HJJ Approach* [JHJ07].

The first research strand is described in Section 6.2. Currently two different solutions to this research strand are investigated. The first solution - the "Bosch approach" - has been described in Chapter 4 and in Appendix B. In Section 6.2.1 open issues of the Bosch approach are discussed. Two alternative proposed solutions which are currently being investigated by the University of Newcastle are described in Section 6.2.2 - *Jackson's diagrams extensions* and Section 6.2.3 - *The UML-B way.*

The second research strand is about the applicability of the HJJ approach to safety-critical embedded automotive systems such as the cruise control. The application of this approach to the cruise control system has also been investigated by the University of Newcastle. First results and open issues of this application are described in Section 6.3.

For both of these strands, the collaboration with Bosch has resulted in an ideal scenario for "in vitro" experimentations. Besides reporting on first results and open research issues along the two research strands mentioned above this chapter also addresses an intended strategy for future deployment in Section 6.4.

The work on the two research strands reported in this chapter has been partly published before in a number of papers. The interested reader will find more technical details in [Maz09b] and [Maz09a]. The ideas related to *Jackson's diagrams extensions* are only sketched there and will have more space in future publications. In [Jon] the reader can find (in Chapter 7) a report of some of the current problems described in a preliminary form. Although the presented concepts have not yet been fully deployed by the Deployment Partners, the HJJ method *suggests* the use of Problem Frames in its first phase. Since Bosch has been working on a Cruise Control requirements specification in Problem Frames for the last 18 months, this represents an ideal application of our intuitions.

## 6.2 Linking Problem Frames and Event-B

Linking Problem Frames and Event-B is not an easy task since both methods represent substantially different views of the system. There are indeed two different underlying philosophies behind the two approaches. In the original Problem Frames Approach [Jac01] user requirements are seen as being about relationships in the operational context and not about functions the software system must perform and the focus is not immediately on the functional behaviour. It is someway a change of perspective with respect to other requirements analysis techniques. Consider, for example, the Use Case approach [Bit02]. Here what we do is specifying the interface and the focus is on the interaction user/machine. With PFs we are pushing our attention beyond the machine interface, we are looking into the real world. The problem is there and it is worth to start there. Some of the ideas derived from [JHJ07] (not specifying the digital system in isolation and deriving the specification starting from a wider system in which physical phenomena are measurable) can be indeed tracked back, with some further evolution, to [Jac01]. If we want to use PFs to develop a method for specification of systems, i.e. a description of the machine behaviour, we have to start understanding the problem.

The entire PFs software specification goal is modifying the world (the problem environment) through the creation of a dedicated machine which will be then put into operation in this world. The machine will then operate bringing the desired effects. The overall philosophy is that the problem is located in the world and the solution in the machine. The most important difference with respect to other requirements methodologies is the emphasis on describing the environment and not the machine or its interfaces. The focus of Event-B is, instead, different since its purpose is describing reactive

systems with the help of events, and refinement is used to move from one level of abstraction to the next one, using mathematical proof to verify consistency between the two levels. It is not difficult to see how the emphasis here is more on the machine and its interfaces, an approach very different from PFs.

Thus, bridging Problem Frames and Event-B is a difficult task. Since this mapping is not straightforward it has been separately investigated by Bosch and the University of Newcastle. Although the "Bosch Approach" (see Section 4.3 and Section 4.5) already represents a significant step forward towards linking problem frames and Event-B there are still some open research issues which are described in Section 6.2.1. Two alternative approaches intended to link problem frames and Event-B, namely *"Jackson's diagrams extensions"* (not to confuse with the Bosch extensions to Problem Frames described in Section 4.3) and *"the UML-B way"* are described in Section 6.2.2 and Section 6.2.3.

### 6.2.1 The Bosch Approach - Open Issues

At Bosch a significant step forward towards linking problem frames and Event-B has been taken. This progress is witnessed by the work reported in this report, especially in Chapter 4 and Appendix B about Pilot Deployment.

In Section 4.5, Table 4.3 shows the mapping between Problem Frames and Event-B. It is worth noting how the semi-formal specification generated by the process described above is still not expressive enough to link Problem Frames to Event-B since it lacks proper behavioural information. Indeed, it is not clear how a general requirement should be mapped into one or more events in a repeatable way without such a precious description or representation in a proper formalism (or formal language). So far the requirements, although clearly structured, are still expressed in natural language and the overall system behaviour has not been described anywhere. This consideration shows the need for a formalism able to express the dynamic behaviour and here is where Bosch opted for Finite State Machines as a way to augment the mainly static information given by PDs. Although this part of the work regarding behaviour is still in progress, it is already clearly recognizable how the overall Bosch methodology is a significant step toward full deployment. Anyway, it is worth emphasizing the need for behavioural "extension" of PDs (and a more precise process to get there) when considering the future directions. This will become even clearer in the following section when presenting alternative solutions to the problem.

Other open research issues of the Bosch approach are briefly described in Section 5.3 - Feedback to Methods.

### 6.2.2 Jackson's diagrams extensions

It is clearly understood that the objective of a PF analysis [Jac01] is the decomposition of a problem into a set of subproblems, where each of these matches a Problem Frame. A Problem Frame is a problem pattern, i.e the description of a simple and generic problem for which the solution is already known. Our perception is that, when describing the behaviour of interfering processes - especially when faults are considered as a special case of interference - the diagrams and the patterns provided are not powerful enough. For this reason, Jackson's diagrams extensions have been introduced in [Maz09b]. In our current understanding *Interference Diagrams* and *Process Diagrams* would be able to show how a Problem Diagram is linked to the dynamic set of existing processes, how they belong to different domains and how the requirements are related to this:

1. **Interface Diagram**: it represents an external, static view of the system. It is able to identify the operations of the system and its domains, and the input/output data of these operations (with their types). The relationship of these with the requirements identified in the Problem Diagram has to be represented at this stage.

2. **Process Diagram**: the whole system is represented as a sequential process and each of its domains as a sequential process. Concurrency within the system or within its domains is modelled by representing these as two or more subcomponents plus their rely and guarantee conditions [Jon81, Jon83b, Jon83a]. This is an external, dynamic view of the system and its domains.

In Figure 6.1 the process from Context to Behaviour is represented. Although there has been close engagement with Bosch on the mini-Pilot and pilot study requirements (especially regarding the use of Jackson's Problem Frames in conjunction with formal modelling in Event-B), and a number of on site meetings have been successfully performed, so far this problem is still open and the use of Jackson's diagrams extensions has not yet been fully investigated.

### 6.2.3 The UML-B way

UML-B [SB06], is a graphical formal modelling notation combining UML and Event-B. Although similar to UML, its main advantage is the fact that it provides tool support including a drawing tool and a translator to generate Event-B models. The tool support is provided in the form of a plug-in for the

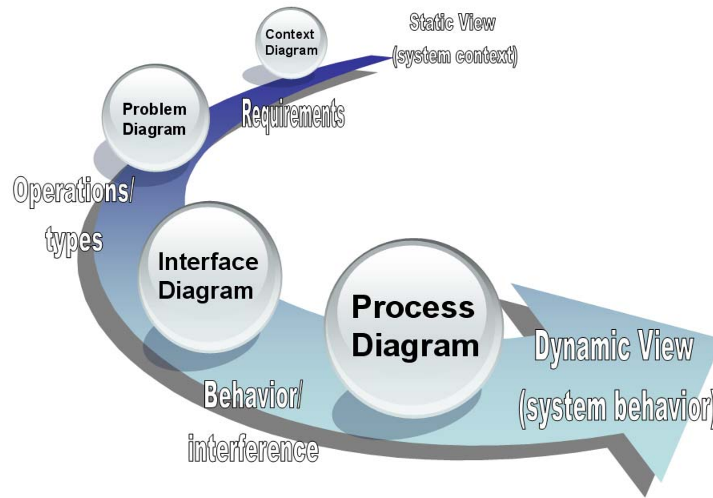


Figure 6.1: From Context to Behaviour

RODIN platform which is an Eclipse-based formal development framework for Event-B. By using UML-B one can graphically model various aspects of a system using class diagrams and finite state machines. One can also attach, graphically, formal constructs like invariants and theorems to the two diagram types. There are two very appealing things about UML-B. The first is its use of the "precise" mathematical semantics of Event-B combined with the graphical representation that can be automatically translated into Event-B. The second interesting aspect is the possibility of defining finite state machines to "enrich" the "static system representation" (for example class diagrams) in such a way to include behavioural details.

While Jackson's diagrams extensions represent a Problem Frames extension to get closer to Event-B, UML-B is an effort in the other direction. Both the ideas presented here aim at establishing an "intermediate" notation able to fill the Problem Frames/Event-B gap but starting from different points. Using class diagrams, UML-B provides a way of extending Problem Diagrams with "operations" like Interface Diagrams. Furthermore when depicting state machines UML-B is describing the system in its behavioural aspect in a similar way to what Process diagrams intend to do. This may appear surprising since Jackson's diagrams extensions have been devised with (almost) no knowledge of UML-B. Anyway, this seems as a revealing natural hint suggesting how, whatever side one decides to start with, filling this gap requires a "extension" in terms of types and behaviour.

Although UML-B seems to be promising for a description of the dynamic

behavior of the system which is missing in the original problem frames approach there are still open research issues that should be investigated. Especially it is not yet clear how to model implementation specific requirements in state diagrams in UML-B.

### 6.2.4 Comparsion of the Approaches

All the approaches are certainly appropriate for reaching the desired formal specification, but we believe there is a subtle difference that it is worth considering. The Bosch approach, with respect to the others, is delaying the moment in which the behavioral information is added. Obviously, both the attitudes present a bill with pros and cons. For example, adding behavioural information at an early stage means having to cope with further details that you could abstract over while you want to concentrate on decomposition or refinement. However, ignoring the behavioural nature of Event-B at the beginning can have negative consequences and possible additional complications at the time of the encoding. A mixed solution might be considered to maximize the positive impact of the different approaches.

## 6.3 The HJJ Approach

The HJJ research strand inside DEPLOY is about investigating and exploiting an approach for deriving specifications of systems not running in isolation but in the physical world and interacting though sensors and actuators with physical phenomena. This work has to be intended as a development of the ideas presented in the original Hayes/Jones/Jackson paper [JHJ07]. During the first 18 months of the project our methodological work focused mainly on the definition of a HJJ method for formally deriving such specifications. The method was indeed only initially sketched in the original paper. In [Maz09b] and [Maz09a] the difference between methods and languages is of main importance and it is precisely stated, the problems arising when confusing the two terms are well explained. Furthermore, the need for a method is identified (what it was missing in [JHJ07] was indeed the method not the formal approach). A more precise formalization of the Hayes/Jones/Jackson method is given and the idea of Layered Fault Tolerant Specification (LFTS) is proposed to make the method extensible to fault tolerant systems. The LFTS principle consists in layering the specification in different levels, the first one for the normal behaviour and the others for the abnormal. The abnormal behaviour is described in terms of an Error Injector (EI) which represents a model of the erroneous interference coming from the environ-

ment. This structure has been inspired by the notion of idealized fault tolerant component [LA90] but the combination of LFTS and EI using contracts (rely/guarantee, for example, but other solutions might be considered) to describe their interaction can be considered as a novel contribution of the project. More details about the method and its approach can be also found in [Jon], in the following we just provide an overview focusing on the automotive case study which represents an interesting challenge and gives us the possibility of exploiting all the main ideas discussed there.

### 6.3.1 The HJJ method and its Steps

In [Maz09a] we analyzed the method introduced in [JHJ07] according to some properties that a method should have. Three macroscopic steps are recognizable in the method:

1. Define boundaries of the systems
2. Expose and record assumptions
3. Derive the specification

Our idea is not committing to a single language/notation - we want a formal method, not a formal language - so we will define a general high level approach following these guidelines and we will suggest *reference tools* to cope with these steps. It is worth noting that these are only reference tools that are *suggested* to the designers. A formal notation can be the final product of the method but it still needs to be distinguishable from the method itself. In figure 6.2 these steps are presented and it is shown how different tools could fit the method at different stages. We call these notations the plug-ins since they can be plugged into the different steps.

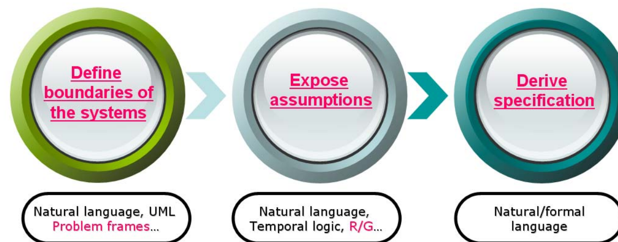


Figure 6.2: Steps and Reference Tools

Figure 6.2 is a generic representation of the method where we want to emphasize the different steps that were not clearly defined previously in [JHJ07].

The reader will understand that this is still a simplification of the process. We use the word "steps" instead of "phases" since we do not want to suggest a sort of linear process which is not always applicable, in the average case (especially when coping with fault tolerance as we will discuss later). We imagine, in the general case, many iterations between the different steps. The idea of the method is to ground the view of the silicon package in the external physical world. This is the problem world where assumptions about the physical components *outside* the computer itself have to be recorded. Only after this can we derive the specification for the software that will run *inside* the computer. The more precise formalization of the method and the features it has to exhibit is one of the main contributions of [Maz09a].

### 6.3.2 LFTS and the Automotive Case Study

One of the requirements of the cruise control, that we will analyze here, is to be switched off if an error in engine speed sensor is detected. This has to be taken into account in the specification. When the engine power is not adjusted properly, the absolute difference between the actual speed and the desired speed will not be decreased. A monolithic specification gathering together these aspects and the more ordinary ones would of course result in something complete but certainly not easily readable and well organized. Pragmatically, following the LFTS principle, we have to organize the specification in two layers: normal/abnormal (i.e. speed acquisition fails) adding a weaker layer of rely conditions for the second one [Jon81, Jon83b, Jon83a]. We use the CrCt to show how the idea of LFTS can be applied in (semi)realistic systems, i.e simplifications of real system for the sake of experimenting with new ideas but still not mere toy examples. Let us consider the following ideal piece of CrCt code:

```
while (target <> current){
   delta := smooth(target, current);
   result := set_eng(delta);
}
```

The car speed is acquired in `smooth(target, current)` and then a delta is calculated for the car to have a smooth acceleration (smoothness has to be determined by experience). The specification of this code in terms of P,Q,R,G is the following (it is expressed in natural language since we are not giving a mathematical model of the car here):

- P: target has to be in a given range

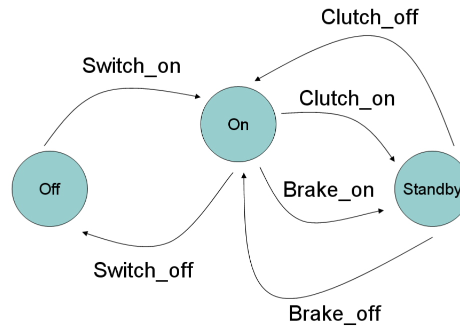


Figure 6.3: Simplified State Machine (no tip up/down)

- Q: delta is zero and the driver has been comfortable with the acceleration
- R: the engine is adjusted (smoothly) according to delta
- G: the absolute value of delta is decreasing

The requirement shown in Figure 6.4 is not taken into account in this ideal piece of code, so in case the speed acquisition goes wrong the guarantee will not hold and the absolute value of delta will not be decreased. Indeed, following the LFTS principle we should organize it in two layers: a normal mode and an abnormal one (speed acquisition goes wrong):

```
while (target <> current){
   delta := smooth(target, current);
   result := set_eng(delta);
   if result <> OK then
      switch_off
}
```

This means adding a weaker layer of conditions for the "abnormal case" that will still be able to provide certain (weaker) guarantees. If speed acquisition goes wrong we do not want to force the engine to follow the delta since it would imply asking for more power when, for example, the car speed is actually decreasing (maybe an accident is happening or it is just out of fuel). By switching the engine off we avoid an expensive engine damage. Figure 6.4 summarizes the layering and the entire structure of this simple example.

So far Jackson's diagrams extensions like Interference Diagrams and Process Diagrams have been not fully developed and exploited in this context,

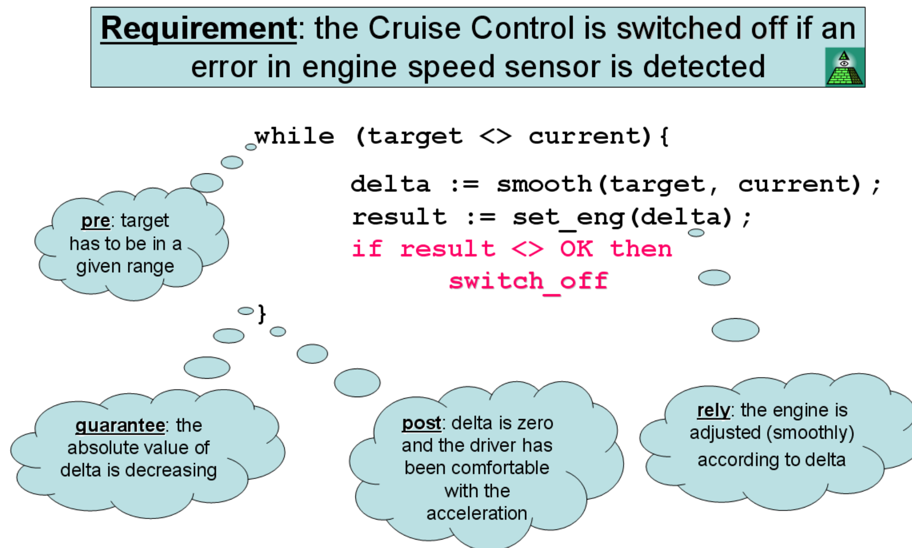


Figure 6.4: Layered CrCt Specification

however an ideal Process Diagram for the Cruise Control is depicted in Figure 6.5. For the diagram to be complete we should be able to describe the rely/guarantee conditions of each subcomponent represented. In its current state this diagram is able to show how a Problem Diagram is linked to the dynamic set of existing processes, how they belongs to different domains and how the requirements are related to this.

## 6.4 Future Strategy for Deployment

The future strategy for deployment will certainly depend on the appropriateness of the different approaches on linking problem frames and Event-B. Although we do not know yet which of the approaches is suited best, at least we identified the following main goals for future deployment:

1. Exploring the issues around Finite State Diagrams (and their translation to Event-B)

2. Exploring link between requirements notations (especially PFDs) and Event-B models

Both these investigations need to gain more confidence with the different ways proposed to select the most approachable for the Cruise Control. Other goals are related to different issues that may result, if solved, in a major advance in formal methods deployment in the Automotive Sector:

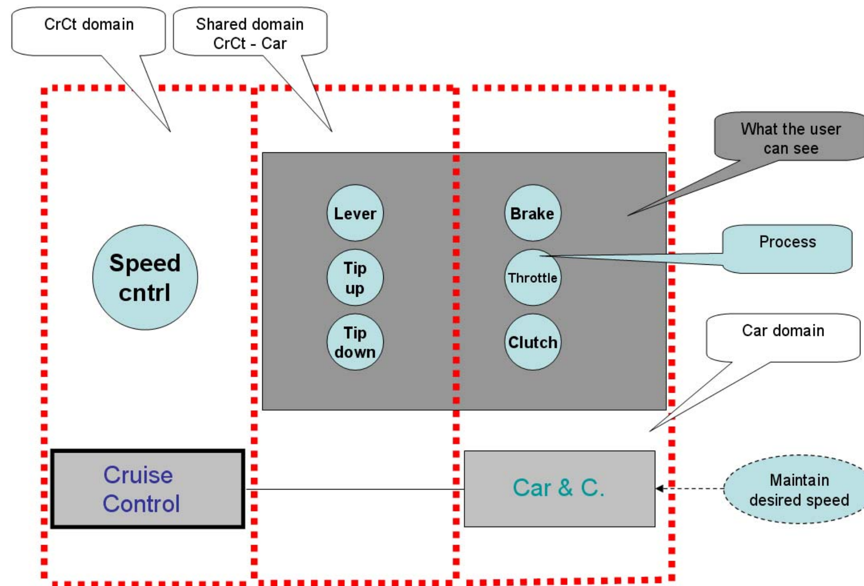


Figure 6.5: Static and Dynamic views combined

1. Formally defining the abnormal behaviour on the basis of a dedicated formal language with well defined semantics. Timebands have been investigated in this regard but other more specific solutions may result interesting as well.

2. Looking at issues around "discretising" continuous requirements

3. As mentioned in section 5.3.4, timing issues and continuous behavior have not been included and need further investigation.

As far as we are aware (1) is only an issue for Bosch and SSF while, in the broader interpretation, (2) and (3) could be of interest for all deployment partners.

# Chapter 7

# Conclusions

We would like repeat the goals for the task T1.3 Pilot Deployment as described in the DoW and comment the degree of fullfilling these goals in this Chapter. The goals for task T1.3 were:

- Amend the requirement specification of selected subproblem for formal modelling and validate the obtained formal requirement specification
- Use the DEPLOY methodology to construct designs and prototype implementations and validate the implementation
- Identify and evaluate benefits/drawbacks of the methodology, appraise cost-effectiveness
- Identify and evaluate consequences for development process

The original idea - during the time the DoW was written - was to first spend some work on amending the requirements before starting with the Event-B model. In WP1 we decided very early that this approach was not feasible due to several reasons. First of all, the requirements we obtained from the traditional requirements engineering process were not the optimal starting point for formal modelling. The gap between requirements in natural language and a formal model e.g. described in Event- B were simply too big. From our point of view, the validation problem in such a one-step approach is not economically solveable. Such a development process might work very well in small scaled applications but not in industrial size applications. Therefore, we developed an approach to overcome this problem which has been described in Section 4.3. This approach uses a semi-formal requirements engineering method with several extensions to bridge the gap between

informal specification and Event-B. Using this approach we were able to divide the big validation problem into two smaller ones. As a side effect we also raised the quality of the requirements.

Our first experiences in modelling in Event-B have been very encouraging. Nevertheless there are some challenges left, namely expressing responsiveness and time in Event-B. These two issues need further research. We are currently aware of the fact that Event-B does not provide means to model closed loop controllers and continous behavior. Therefore, we decided not to include the development of closed loop controllers in the Event-B development process. It is an open question to what extent simplified models of a controller could be included in the Event-B model. Again more research is needed here.

We have gathered evidence (see Section 5.2) that the use of Problem Frames is very helpful. We reduced the number of requirements and increased the quality by the same means. From an overall perspective the refinment strategy in Event-B naturally follows the refinment strategy of our extended Problem Frames Approach. However, some questions still remain open. For example, the concrete mapping of extended Problem Frames to Event-B model elements (see Section 6) is not yet fully solved. At the moment more experience with Event-B is needed before we are able to report evidence about Event-B.

We also have identified some consequences (Section 3.1) for the development process (more consequences may be necessary if we have gathered more experience and the remaining open questions are solved). The biggest change at the moment is the introduction of an intermediate phase, i.e., the semiformal requirements engineering phase using Problem Frames.

# Appendix A

# Cruise Control - Semiformal Specification

The following sections describes the application of the semi-formal requirements engineering process on the cruise control system.

## A.1 Identification of Abstract Context Diagram

We start with the abstract context diagram (Context 0) which is shown in Figure A.1.

The driver interacts with the environment *Env* at interface *a* using controlled phenomena of the environment, e.g., the lever. The cruise control machine interacts with the domain *Env* at interface *b*. At interface *b* the machine shares part of the phenomena controlled by itself with the *environment*, e.g., the desired acceleration *P_CrCtl__Acceleration*, and it sees phenomena that are controlled by the environment and shared with the machine, e.g., the vehicle speed *P_Env__VehicleSpeed*. At interface *c* the machine shares phenomena controlled by itself with the designed domain *Speed Memory*, i.e., the target speed *P_CrCtl__TargetSpeed* and it sees phenomena controlled by the speed memory which are shared with the machine, i.e., the stored target speed *P_SpeedMemory__StoredTargetSpeed*.

The requirement *R500* shown in Figure A.1 states:

| R500.1: The target speed (P_SpeedMemory__StoredTargetSpeed) must relate to the control signals (P_Env__ControlSignals) the environment generates. This relation must guarantee safety (P_Para, P_Env) and ensure comfort if possible. |
|---|

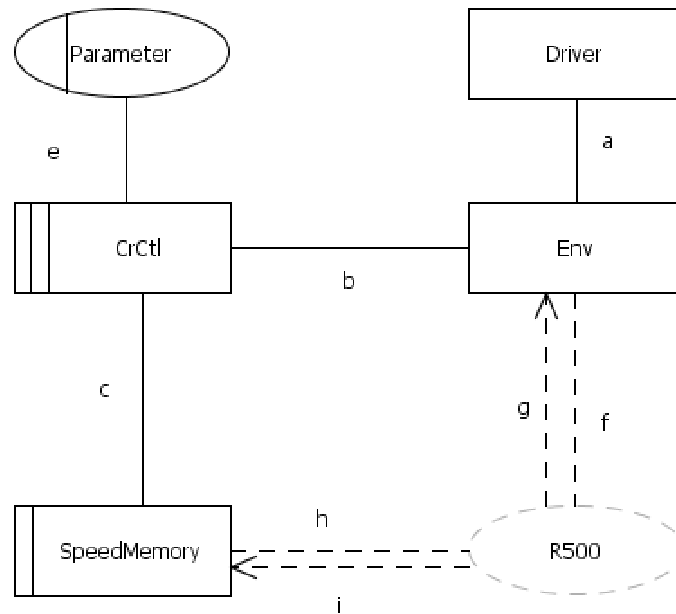


Figure A.1: Cruise Control - Abstract context diagram (Context_0).

R500.2: If the corresponding control signals (P_Env__ControlSignals) are received and if it is safe and possible, the vehicle speed (P_Env__VehicleSpeed) must be influenced comfortably in one of the following three ways:

- The vehicle speed must stay to the target speed (P_SpeedMemory__StoredTargetSpeed) as close as possible or reach the target speed (P_SpeedMemory__StoredTargetSpeed) as soon as possible.
- The vehicle speed must increase/decrease.
- The vehicle speed must not be influenced.

R500.3: At any given time, the output phenomena (P_Env__Output) must correctly reflect the target speed (P_SpeedMemory__StoredTargetSpeed) and the current state (P_StateModel__StoredState) of the cruise control system.

The requirement R500 refers to phenomena of the *environment domain*, i.e., control signals of the cruise control interface, and of the *speed memory*, i.e., the stored target speed and constrains phenomena of the *environment domain*, i.e., the vehicle speed. This is illustrated in Figure A.1 with the dashed lines labelled with *f, g, h, i.*

## A.2 Elaboration of Context Diagram

In the second step of the process, the abstract context diagram (Context 0) is elaborated into a more concrete context diagram (Context 1) which is shown in Figure A.2.

As one can see from Figure A.2 the cruise control system interacts with the domains *Input HMI*, *Vehicle&Env*, and *Output HMI*. The domain *Input HMI* hereby describes input devices which are used by the driver to influence the behaviour of the cruise control system. The domain *Vehicle&Env* represents the parts of the vehicle which are influenced by the cruise control system. The domain *Output HMI* describes output devices which are used to display information about the cruise control system to the driver. The designed domains *Speed Memory* and *State Model*, and the designed description *Parameters* describe parts of the cruise control machine which are needed for implementation.

Table A.1 shows the phenomena which are shared between the different domains.

| Label | Controlled by | Phenomena |
|---|---|---|
| a | Driver | P_DriverInteraction |
| b | Input HMI | P_Env_ControlSignals |
| c | CrCtl | P_CrCtl_State |
| c | State Model | P_StateModel_StoredState |
| d | CrCtl | P_CrCtl_TargetSpeed |
| d | Speed Memory | P_SpeedMemory_StoredTargetSpeed |
| e | Cruise Control | P_CrCtl_Acceleration |
| e | Vehicle&Env | P_Vehicle, P_Env_VehicleSpeed |
| f | CrCtl | P_Env__Vehicle_CrCtlState |
| g | Output HMI | P_Env__OutputHMI__Output |
| h | Parameter | P_Parameter__Para |
| i | Timer | P_Timer__Time |

Table A.1: Elaborated context diagram - Shared Phenomena

As you can see from Figure A.2 and from Table A.1, the given domain *Driver* is now connected to the domains *Input HMI* (Interface *a*) and *Output HMI* (Interface *g*). Thus, the domains *Input HMI* and *Output HMI* are used as connection domains to communicate the *interactions of the driver P_DriverInteraction* to the CrCtl and to communicate the *status of the CrCtl* (P_Env__Vehicle_CrCtlState) to the driver.

Furthermore, you can see from Figure A.2 that requirement R500 has been decomposed into requirements R501, R502, and R503. The using and

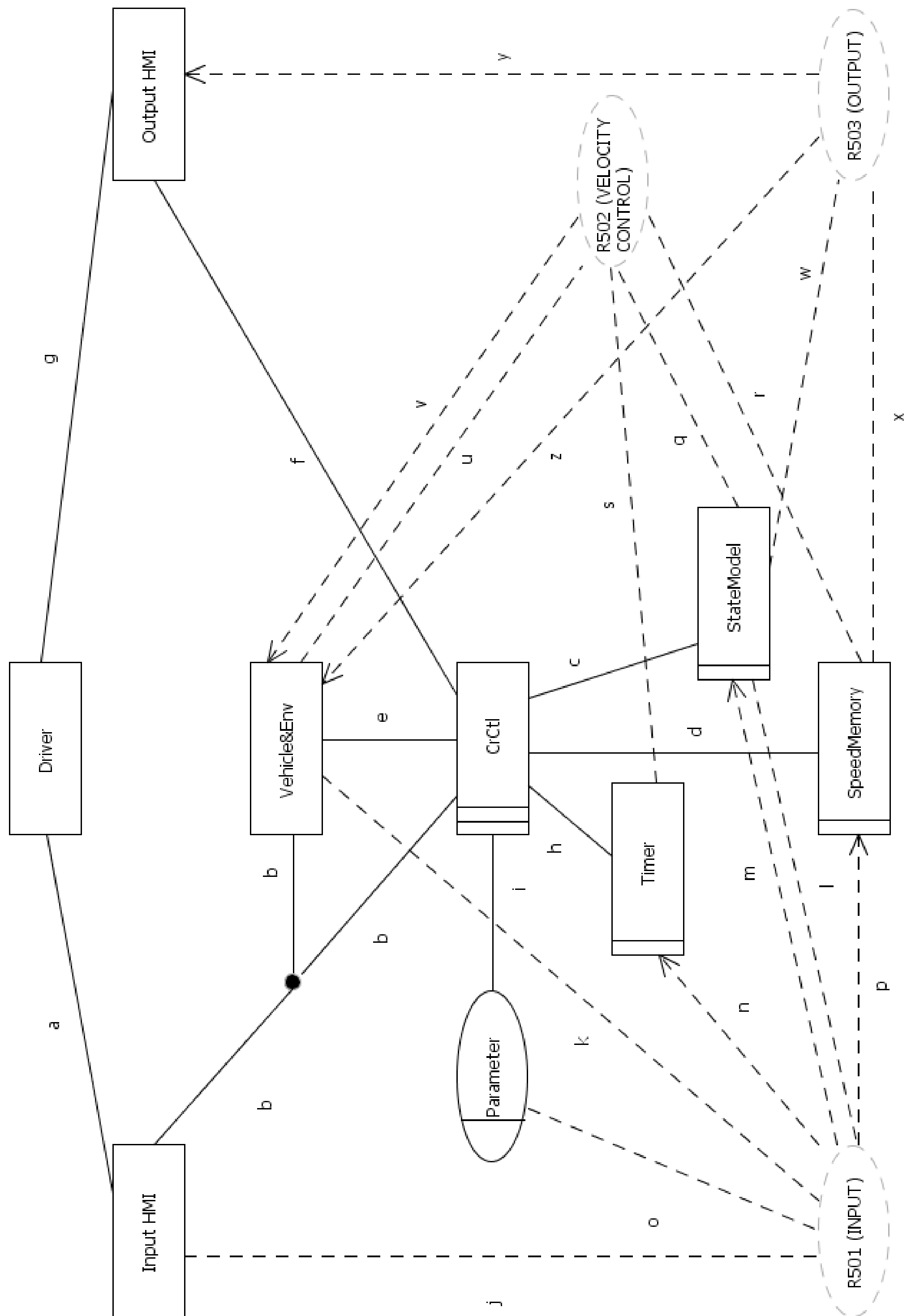


Figure A.2: Cruise Control - Elaborated Context Diagram (Context_1).

constraining requirement references of the original requirement R500 shown in Figure A.1 have been adapted in Figure A.2. They now point to the domains *Input HMI*, *Output HMI*, and *Vehicle&Env* which represent the decomposition of the given domain *Env*. The requirements R501, R502, and R503 state:

R501: Depending on the control signals (P_Env__ControlSignals) and/or switch-off conditions (P_Vehicle), the stored state (P_StateModel__StoredState) must change safely and comfortably to UBAT_OFF, INIT, OFF, STANDBY, ERROR, CRUISE, RESUME, RAMP_DONW, ACC or DEC and/or the stored target speed (P_SpeedMemory__StoredTargetSpeed) must be set or deleted and/or the timer must be set (P_Timer_Time).

R502: Depending on the stored state (P_StateModel__StoredState), the vehicle speed (P_Env__VehicleSpeed) must be influenced comfortably and safely in one of the following ways:

- CRUISE/RESUME: The vehicle speed must stay to the target speed (P_SpeedMemory__StoredTargetSpeed) as close as possible or must reach the target speed (P_SpeedMemory__StoredTargetSpeed) as soon as possible.
- ACC/DEC/RAMP_DOWN: The vehicle speed must increase or decrease.
- UBAT_OFF, INIT, OFF, ERROR, R_ERROR, STANDBY: The vehicle speed must not be influenced.

R503: Information on the current state of the cruise control system (P_StateModel__StoredState) must be provided to the driver and other ECUs (P_Env_OutputHMI__Output). As long as the cruise control system is capable of actively influencing the target speed, the stored target speed (P_SpeedMemory__StoredTargetSpeed) must be provided to the driver (P_Env_OutputHMI__Output). At any given time at which state and speed information is provided to other ECUs or the driver, the output phenomena (P_Env_OutputHMI__Output) must correctly reflect the stored state (P_StateModel__StoredState) and the stored target speed (P_SpeedMemory__StoredTargetSpeed).

## A.3 Identification of Subproblems

As you can see from Figure 4.2, the next step after the elaboration of the system context is the *identification of subproblems*. During pilot deployment we identified the following subproblems *signal evaluation*, *velocity control*, and *display*.

## A.4 Projection into Subproblems

After the relevant subproblems have been identified, the elaborated context diagram is projected into the identified subproblems (see Step 4 in Figure 4.2). The result of this step is a set of problem diagrams which describe the different subproblems. Each subproblem is a *projection* of the elaborated system context shown in Figure A.2. In the following we will focus on the signal evaluation subproblem. Figure A.3 shows the problem diagram for the signal evaluation aspect of the cruise control system. It is a projection of the elaborated context diagram shown in Figure A.2. It contains the domains *Input HMI* and *Vehicle* which are taken over as is from the elaborated context diagram. As you can see from Figure A.3, the name of the machine domain has been changed to *CrCtl SignalEval* in order to indicate that the machine only describes the signal evaluation aspect of the cruise control system.

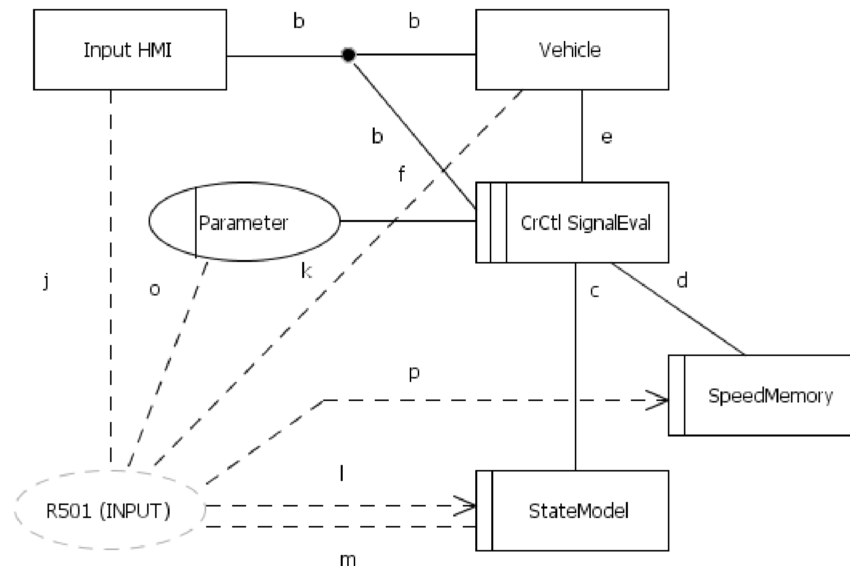


Figure A.3: Cruise Control - Signal Evaluation (SignalEval_0).

The requirement *R501 (INPUT)* shown in the elaborated context diagram (Figure A.2) has been taken over as is:

| R501: Depending on the control signals (P_Env__ControlSignals) and/or switch-off conditions (P_Vehicle), the stored state (P_StateModel__StoredState) must change safely and comfortably to UBAT_OFF, INIT, OFF, STANDBY, ERROR, CRUISE, RESUME, RAMP_DONW, ACC or DEC and/or the stored target speed (P_SpeedMemory__StoredTargetSpeed) must be set or deleted and/or the timer must be set (P_Timer_Time). |
|---|

The set of shared phenomena between domains is exactly the same as in the elaborated context diagram.

## A.5 Elaboration of Subproblems

The last step of the process depicted in Figure 4.2 is to elaborate the subproblems identified in step 3 and projected in step 4. Figure A.4 shows an elaboration of the signal evaluation aspect shown in Figure A.3. As you can see from Figure A.4, the domain *Input HMI* has been decomposed into the domains *Pedals*, *Control Interface* and *Ignition*. The domain *Vehicle&Env*, the designed domains *Timer*, *State Model*, and *Speed Memory* have been taken over as is.

The original requirement R501 shown in Figure A.3 has been decomposed into the requirements R504 (PEDALS), R505 (CONTROL INTERFACE), R506 (VEHICLE), R534 (INIT REQ), and R78/R508 (IGNITION). These requirements state:

| R504 (PEDALS): If a pedal signal (P_Env__ControlSignals__PedalSignals) is set, the stored state is CRUISE, RESUME, ACC, DEC,RAMP_DOWN or STANDBY (P_StateModel__StoredState), and the clamp15 signal is set (P_Env__ControlSignals__Cl15), and no error conditions are set (P_Vehicle), the stored state (P_StateModel__StoredState) must be switched to R_ERROR. |
|---|

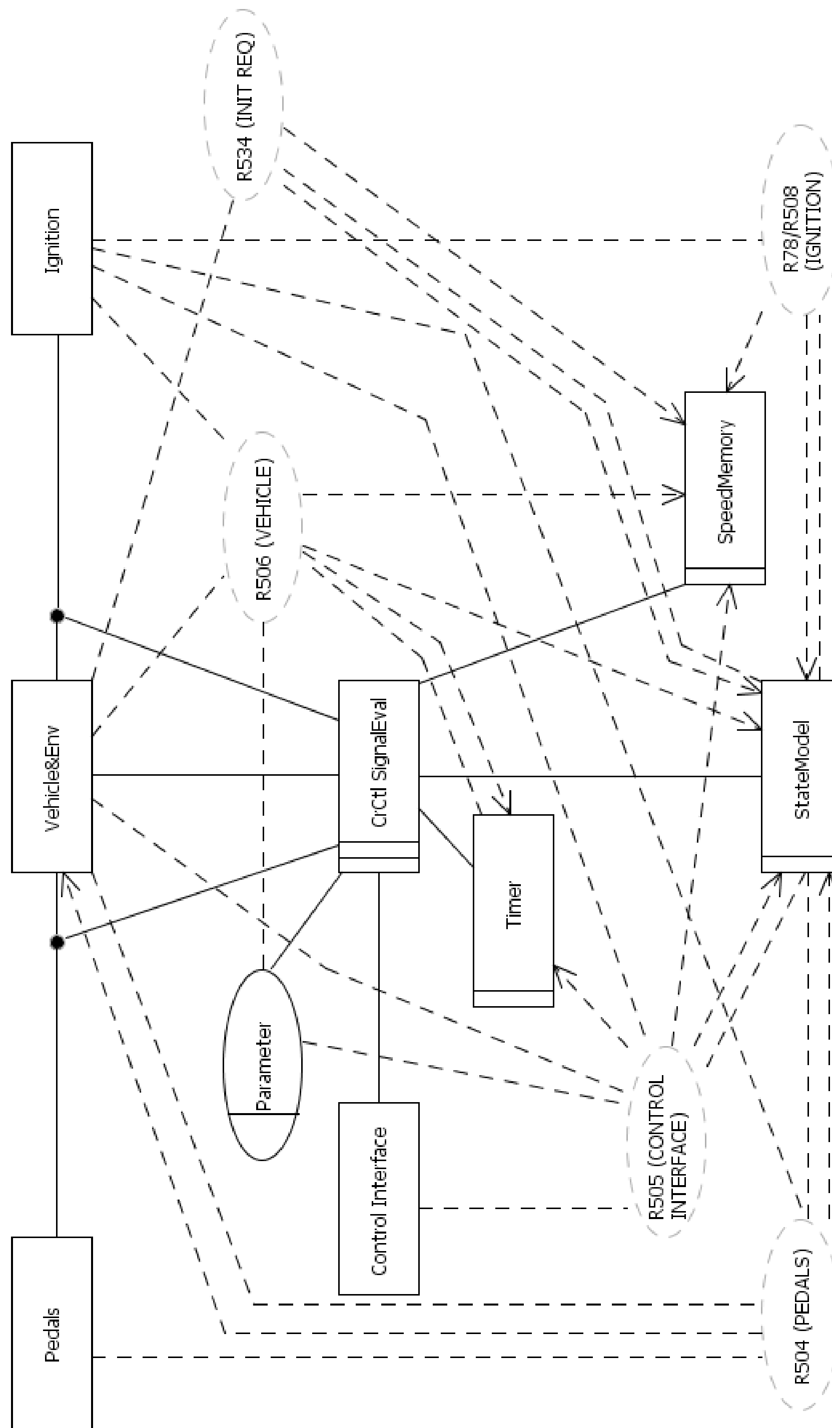


Figure A.4: Cruise Control - Signal Evaluation (SignalEval_1).

R505 (CONTROL INTERFACE): If the appropriate control interface signals (P_Env__ControlSignals__ControlInterfaceSignals) are set, the clamp15 signal is set (P_Env__ControlSignals__Cl15), no switch-off or error conditions are set (P_Vehicle) and no pedal signals are set (P_Env__ControlSignals__PedalSignals), and the current stored state (P_StateModel__StoredState) is OFF, STANDBY, CRUISE, RESUME, ACC or DEC, the stored state (P_StateModel__StoredState) must be changed to either OFF, STANDBY, CRUISE , RESUME, ACC or DEC and/or the stored target speed (P_SpeedMemory__StoredTargetSpeed) must be set, changed or deleted, and/or the timer (P_Timer_Time) must be set.

R506 (VEHICLE): If any of the error, switch-off, comfort-switch-off, or vehicle conditions are set (P_Vehicle) and the clamp 15 signal is set (P_Env__ControlSignals__Clamp15), the stored state (P_StateModel__StoredState) must be switched to state R_ERROR, RAMP_DOWN or ERROR. Depending on the switch-off or error condition, the stored target speed (P_SpeedMemory__StoredTargetSpeed) must or must not be deleted. Furthermore, the timer (P_Timer_Time) must be refreshed.

R534 (INIT REQ): If the Init_request signal is set (P_Env__Vehicle__InitRequest), the stored state must be switched to INIT (P_StateModel__StoredState). In this case, the target speed must be deleted (P_SpeedMemory__StoredTargetSpeed).

R78 (IGNITION): If the clamp 15 signal is set (P_Env__ControlSignals__Cl15) and the stored state (P_StateModel__StoredState) is UBAT_OFF, the stored state must be switched to INIT (P_StateModel__StoredState) and the target speed must be deleted (P_SpeedMemory__StoredTargetSpeed).

R508 (IGNITION): If the clamp 15 signal is not set (P_Env__ControlSignals__Cl15), the stored state must be switched to UBAT_OFF (P_StateModel__StoredState) regardless of the current state. In this case the target speed must be deleted (P_SpeedMemory__StoredTargetSpeed).

We also have applied the process steps described in Section A.4 and A.5 to the other subproblems, i.e., velocity control and display. Furthermore, we applied the process steps 3 to 5 iteratively to all elaborated subproblems, i.e., SignalEval_1, VelocityControl_1, and Display_1, until the cruise control system had been fully specified.

# Appendix B

# Cruise Control - Formal Specification

In the following sections, we describe the application of the formal specification process on parts of the cruise control. The technical steps of the process are described in Section 4.5

## B.1 Abstract Model

Reconsider the abstract context diagram shown in Figure A.1. This context diagram is mapped to a machine called *CrCtl_Context_0* and an associated context called *c0* in Event-B. In the context *c0* we define types which are later used in the associated machine *CrCtl_Context_0*. These types are either defined by using *constants* or *carrier sets*. Furthermore, the context is used to define abstract functions which are later used within *events*. These functions are defined using *axioms*.

For example, in the context *c0* we have defined the type *T_Speed*, and the function *F_VehicleSpeed* as follows:

**CONTEXT** c0

**SETS**

`T_Speed` type for vehicle/target speed

**CONSTANTS**

`F_VehicleSpeed` function for calculating the vehicle speed

`CONTROL` constant for CONTROL mode of CrCtl

`ACONTROL` constant for ACONTROL mode of CrCtl

`NOCONTROL` constant for NOCONTROL mode of CrCtl

`T_Mode` type for CrCtl mode

**AXIOMS**

`axm1` : $F_VehicleSpeed \in T_Acceleration \times T_Env_Vehicle \times T_Env_ControlSignals \times T_Speed \rightarrow T_Speed$

`axm2` : $partition(T_State, CONTROL, ACONTROL, NOCONTROL)$

`axm10` : $T_Mode = \{CONTROL, ACONTROL, NOCONTROL\}$

**END**

We now show how we modelled the requirement R500 shown in Figure A.1 in the machine *CrCtl_Context_0*. The requirement *R500* shown in Figure A.1 states:

R500.1: The target speed (P_SpeedMemory__StoredTargetSpeed) must relate to the control signals (P_Env__ControlSignals) the environment generates. This relation must guarantee safety (P_Para, P_Env) and ensure comfort if possible.

R500.2: Depending on the control signals (P_Env__ControlSignals) the stored state (P_StateModel__StoredState, P_StateModel_Timer) must change safely and comfortably (P_Env_Vehicle, P_Para) to CONTROL —— NOCONTROL —— ACONTROL and/or the stored target speed (P_SpeedMemory__StoredTargetSpeed) must be adjusted.

- CONTROL: The vehicle speed must stay to the target speed (P_SpeedMemory__StoredTargetSpeed) as close as possible or reach the target speed (P_Speed_Memory__StoredTargetSpeed) as soon as possible.
- ACONTROL: The vehicle speed must increase/decrease.
- NOCONTROL: The vehicle speed must not be influenced.

R500.3: At any given time, the output phenomena (P_Env__Output) must correctly reflect the target speed (P_SpeedMemory__StoredTargetSpeed) and the current state (P_StateModel__StoredState) of the cruise control system.

The first step in formally specifying this requirement in Event-B is to define the phenomena used and constrained by the requirement in the machine

*CrCtl_Context_0* as *variables* or *constants*. For each *variable* we need to define its type using a *type invariant*. In case of requirement R500 we made the decision to model the used and constrained phenomena as follows:

**MACHINE** CrCtl_Context0

**SEES** c0

**VARIABLES**

`P_Env_Output` variable for environment output

`P_Env_Vehicle` variable for vehicle signals

`P_Env_ControlSignals` variable for control signals

`P_Env_VehicleSpeed` variable for current vehicle speed

`P_Para` variable for external parameters

`P_CrCtl_Display` variable for display signals generated by CrCtl

`P_CrCtl_Acceleration` variable for demanded acceleration of CrCtl

`P_CrCtl_TargetSpeed` variable for stored target speed (corresponds to P_SpeedMemory_StoredTargetSpeed)

`P_CrCtl_Mode` variable for CrCtl mode

`P_CrCtl_Output` variable for CrCtl output

`P_CrCtl_Timer` variable for CrCtl timer

**INVARIANTS**

`inv2` : $P_CrCtl_TargetSpeed \in T_Speed$

`inv3` : $P_CrCtl_Acceleration \in T_Acceleration$

`inv4` : $P_CrCtl_Display \in T_Display$

`inv5` : $P_Env_Vehicle \in T_Env_Vehicle$

`inv6` : $P_Env_ControlSignals \in T_Env_ControlSignals$

`inv7` : $P_Env_VehicleSpeed \in T_Speed$

`inv8` : $P_Env_Output \in T_Env_Output$

`inv9` : $P_Para \in T_Para$

`inv12` : $P_CrCtl_Mode \in T_Mode$

`inv13` : $P_CrCtl_Output \in T_Env_Output$

`inv14 :` $P_CrCtl_Timer \in T_TimeState$

**END**

After having defined variables for the phenomena used and constrained by requirement R500 we now define the following events which *implement* the requirement R500:

**Event** $CrCtl_Change_Mode \;\widehat{=}$
Machine: CrCtl (R500.1)

**begin**

`act1 :` $P_CrCtl_Mode :\in T_Mode$
`act3 :` $P_CrCtl_Timer :\in T_TimeState$

**end**

The event *CrCtl_Change_Mode* partly fulfills requirement R500.1. It refers to the second part of requirement R500.1 in which the change of *P_CrCtl_Mode* and the change of the timer is stated. Since the requirement does not state the new value of *P_CrCtl_Mode* we nondeterminstically assign a value of type *T_Mode*.

**Event** $CrCtl_Change_TargetSpeed \;\widehat{=}$
Machine: CrCtl (R500.1)

**begin**

`act1 :` $P_CrCtl_TargetSpeed :\in T_Speed$

**end**

The event *CrCtl_Change_TargetSpeed* partly fulfills requirements R500.1 It refers to the first part of requirement R500.1 in which the change of *P_CrCtl_TargetSpeed*. Since the requirement does not state the new value of *P_CrCtl_TargetSpeed* we nondeterminstically assign a value of type *T_Speed*.

The following three events refer to requirement R500.2. They deal with the three different behaviours of the cruise control based on the current mode, i.e., CONTROL, ACONTROL, or NOCONTROL. As you can see from these events, they are all guarded by the *P_CrCtl_Mode* variable. For each case the acceleration demanded by the cruise control (P_CrCtl_Acceleration) is calculated by a function which depends on the current vehicle speed (P_Env_VehicleSpeed) and the stored target speed (P_CrCtl_TargetSpeed).

**Event** *CrCtl_Change_Acceleration_Control* $\widehat{=}$
Machine: CrCtl (R500.2)

**when**

grd1 : $P_CrCtl_Mode = CONTROL$

**then**

act1 : $P_CrCtl_Acceleration :=$
$F_CruiseResume(P_Env_VehicleSpeed \mapsto P_CrCtl_TargetSpeed)$

**end**

**Event** *CrCtl_Change_Acceleration_AControl* $\widehat{=}$
Machine: CrCtl (R500.2)

**when**

grd1 : $P_CrCtl_Mode = ACONTROL$

**then**

act1 : $P_CrCtl_Acceleration :=$
$F_AccDecRampDown(P_Env_VehicleSpeed \mapsto P_CrCtl_TargetSpeed)$

**end**

**Event** *CrCtl_Change_Acceleration_NoControl* $\widehat{=}$
Machine: CrCtl (R500.2)

**when**

grd1 : $P_CrCtl_Mode = NOCONTROL$

**then**

act1 : $P_CrCtl_Acceleration := NO_ACCELERATION$

**end**

The last cruise control event, namely *CrCtl_Change_Output* implements requirement R500.3:

**Event** *CrCtl_Change_Output* $\widehat{=}$
Machine: CrCtl (R500.3)

**begin**

act1 : $P_CrCtl_Output :\in T_Env_Output$

**end**

In order to have a full specification we also need to define events for changing phenomena of the environment. These events simulate the environment, i.e., reactions of the environment on button presses of the driver and reactions of the environment on outputs of the cruise control software.

**Event** $Env_Change_ControlSignals \mathrel{\widehat{=}}$
Domain: Env (simulates control signals generated by the environment)

**begin**
act1 : $P_Env_ControlSignals :\in T_Env_ControlSignals$
**end**

**Event** $Env_Change_Vehicle \mathrel{\widehat{=}}$
Domain: Env (simulates vehicle internal signals)

**begin**
act1 : $P_Env_Vehicle :\in T_Env_Vehicle$
**end**

**Event** $Env_Change_VehicleSpeed \mathrel{\widehat{=}}$
Domain: Env (simulates engine control)

**begin**
act1 : $P_Env_VehicleSpeed := F_VehicleSpeed(P_CrCtl_Acceleration \mapsto P_Env_Vehicle \mapsto P_Env_ControlSignals \mapsto P_Env_VehicleSpeed)$
**end**

**Event** $Env_Change_Output \mathrel{\widehat{=}}$
Domain: Env (simulates output)

**begin**
act1 : $P_Env_Output := F_VehicleOutput(P_CrCtl_Output)$
**end**

We also need to define an initialisation event in order to initialize the variables to well defined values.

**Initialisation**

**begin**

act1 : $P_Env_ControlSignals :\in T_Env_ControlSignals$

act2 : $P_Env_VehicleSpeed :\in T_Speed$

act3 : $P_Env_Vehicle :\in T_Env_Vehicle$

act4 : $P_Env_Output :\in T_Env_Output$

act6 : $P_CrCtl_TargetSpeed :\in T_Speed$

act7 : $P_CrCtl_Acceleration :\in T_Acceleration$

act8 : $P_CrCtl_Display :\in T_Display$

act9 : $P_Para :\in T_Para$

act13 : $P_CrCtl_Mode := NOCONTROL$

act14 : $P_CrCtl_Output :\in T_Env_Output$

act15 : $P_CrCtl_Timer :\in T_TimeState$

**end**

## B.2 Refinement of Abstract Model

After having specified a first abstract model of the cruise control system we will now refine this model. We start with mapping the elaborated context diagram *Context_1* shown in Figure A.2 to an Event-B context called *c1* and an Event-B machine called *CrCtl_Context1*. The context *c1* hereby refines the context *c0* and the machine *CrCtl_Context1* refines the abstract machine *CrCtl_Context0*.

Reconsider the requirements R501, R502, and R503:

| R501: Depending on the control signals (P_Env__ControlSignals) and/or switch-off conditions (P_Vehicle), the stored state (P_StateModel__StoredState) must change safely and comfortably to UBAT_OFF, INIT, OFF, STANDBY, ERROR, CRUISE, RESUME, RAMP_DOWN, ACC or DEC and/or the stored target speed (P_SpeedMemory__StoredTargetSpeed) must be set or deleted and/or the timer must be set (P_Timer_Time). |
|---|

R502: Depending on the stored state (P_StateModel__StoredState), the vehicle speed (P_Env__VehicleSpeed) must be influenced comfortably and safely in one of the following ways:

- CRUISE/RESUME: The vehicle speed must stay to the target speed (P_SpeedMemory__StoredTargetSpeed) as close as possible or must reach the target speed (P_SpeedMemory__StoredTargetSpeed) as soon as possible.
- ACC/DEC/RAMP_DOWN: The vehicle speed must increase or decrease.
- UBAT_OFF, INIT, OFF, ERROR, R_ERROR, STANDBY: The vehicle speed must not be influenced.

R503: Information on the current state of the cruise control system (P_StateModel__StoredState) must be provided to the driver and other ECUs (P_Env_OutputHMI__Output). As long as the cruise control system is capable of actively influencing the target speed, the stored target speed (P_SpeedMemory__StoredTargetSpeed) must be provided to the driver (P_Env_OutputHMI__Output). At any given time at which state and speed information is provided to other ECUs or the driver, the output phenomena (P_Env_OutputHMI__Output) must correctly reflect the stored state (P_StateModel__StoredState) and the stored target speed (P_SpeedMemory__StoredTargetSpeed).

If you compare the requirements R501, R502, R503 of the elaborated context diagram with the requirement R500 you see that the instead of the P__mode the requirements now refer to the more concrete state (P_StateModel__StoredState). Table B.1 shows the mapping of cruise control modes to cruise control states again.

In order to achieve this mapping of modes to concrete states in Event-B we need to do a *data refinement* of the type of the variable *P_CrCtl_Mode*. First of all, we need to define the values of the concrete cruise control state as constants in context *c1*. Furthermore we need to relate the values of the cruise control mode to the corresponding values of the cruise control state. This can be achieved by the following axioms in context *c1*:

**AXIOMS**

axm1 : $partition(NOCONTROL, \{UBAT_OFF\}, \{OFF\}, \{ERROR\}, \{R_ERROR\}, \{STANDBY\}, \{INIT\})$

| Mode | State | Description |
|---|---|---|
| NO_CONTROL | UBAT_OFF | Ignition is off and engine not running |
| | INIT | Ignition is ON and cruise control is being intialized |
| | OFF | Ignition is ON, cruise control has been initialized and is switched OFF |
| | ERROR | An irreversible error has occurred |
| | STANDBY | Cruise control has been switched ON |
| | R_ERROR | A reversible error has occurred |
| CONTROL | CRUISE | Cruise control is maintaining the target speed |
| | RESUME | The target speed is approached from above or from below |
| ACONTROL | ACC | Cruise control is accelerating the car |
| | DEC | Cruise control is decelerating the car |
| | RAMP_DOWN | Cruise control is being switched off |

Table B.1: Modes and states of the cruise control system.

axm2 : $partition(CONTROL, \{CRUISE\}, \{RESUME\})$

axm3 : $partition(ACONTROL, \{ACC\}, \{DEC\}, \{RAMP_DOWN\})$

thm1 : $NOCONTROL = \{UBAT_OFF, OFF, ERROR, R_ERROR, STANDBY, INIT\}$

thm2 : $CONTROL = \{CRUISE, RESUME\}$

thm3 : $ACONTROL = \{ACC, DEC, RAMP_DOWN\}$

Furthermore we need to define the following invariants in the machine *CrCtl_Context1* to make sure that the mapping between the mode and the states is obeyed by the model:

inv1 : $P_CrCtl_Mode = NOCONTROL \Leftrightarrow P_CrCtl_State \in \{UBAT_OFF, OFF, ERROR, R_ERROR, STANDBY, INIT\}$

inv2 : $P_CrCtl_Mode = CONTROL \Leftrightarrow P_CrCtl_State \in \{CRUISE, RESUME\}$

inv3 : $P_CrCtl_Mode = ACONTROL \Leftrightarrow P_CrCtl_State \in \{ACC, DEC, RAMP_DOWN\}$

The the events *CrCtl_Change_Acceleration_Control*, *CrCtl_Change_Acceleration_AControl*, *CrCtl_Change_Acceleration_NControl* are refined as follows:

**Event** *CrCtl_Change_Acceleration_Control* $\widehat{=}$
Machine: CrCtl (R502)

**refines** *CrCtl_Change_Acceleration_Control*

**when**

grd1 : $P_CrCtl_State \in \{CRUISE, RESUME\}$

**then**

act1 : $P_CrCtl_Acceleration :=$
$F_CruiseResume_Controller(P_Env_VehicleSpeed$
$\mapsto P_CrCtl_TargetSpeed)$

**end**

**Event** *CrCtl_Change_Acceleration_AControl* $\widehat{=}$
Machine: CrCtl (R502)

**refines** *CrCtl_Change_Acceleration_AControl*

**when**

grd1 : $P_CrCtl_State \in \{ACC, DEC, RAMP_DOWN\}$

**then**

act1 : $P_CrCtl_Acceleration :=$
$F_AccDecRampDown_Controller(P_Env_VehicleSpeed$
$\mapsto P_CrCtl_TargetSpeed)$

**end**

**Event** *CrCtl_Change_Acceleration_NoControl* $\widehat{=}$
Machine: CrCtl (R502)

**refines** *CrCtl_Change_Acceleration_NoControl*

**when**

grd1 : $P_CrCtl_State$
$\in \{OFF, UBAT_OFF, INIT, ERROR, R_ERROR, STANDBY\}$

**then**

act1 : $P_CrCtl_Acceleration := NO_ACCELERATION$

**end**

If you compare the abstract events with the refined events above you notice that guards have been strengthened. They now refer to the variable *P_CrCtl_State* instead of referring to *P_CrCtl_Mode* in the abstract machine.

## B.3 Decomposition of Refined Model

After having refined the abstract model we now need to decompose the machine *CrCtl_Context1* into three different machines, namely *SignalEval_0*, *VelocityControl_0* and *Display_0* (see Figure 4.4). These machines will implement the three subproblems *SignalEval_0*, *VelocityControl_0*, and *Display_0* shown in Figure 4.3.

This decomposition is similar to *A-style decomposition* [Abr09a], i.e., decomposition of contexts and machines using shared variables. However, we do not restrict shared variables from being refined in the different parts later on. We only require that the refinements of a shared variable are the same in different parts. For example, the variable *P_CrCtl_State* is shared between the machine *SignalEval_0* and *VelocityControl_0*. If we now refine *P_CrCtl_State* in machine *SignalEval_0* we also need to refine it exactly in the same way in *VelocityControl_0*. In our application of the cruise control we tried to minimize the interface, i.e., the number of shared variables between the different parts of the machine.

As the decomposition of machines and contexts in Event-B follows the projection of domains and requirements into subproblems we need to look at the problem diagram shown in Figure A.3 to determine which parts of the context *c1* and which parts of the machine *CrCtl_Context1* need to be copied to the decomposed context *SignalEval_c0* and to the machine *SignalEval0*. The resulting machine *SignalEval0* after decomposition is the following:

**MACHINE** SignalEval0

**SEES** c1

**VARIABLES**

```
P_Env_Vehicle
P_Env_ControlSignals
P_Env_VehicleSpeed
P_Para
P_CrCtl_TargetSpeed
P_CrCtl_Timer
```

P_CrCtl_State

**INVARIANTS**

invc11 : $P_CrCtl_State \in T_State$

inv2 : $P_CrCtl_TargetSpeed \in T_Speed$

inv5 : $P_Env_Vehicle \in T_Env_Vehicle$

inv6 : $P_Env_ControlSignals \in T_Env_ControlSignals$

inv7 : $P_Env_VehicleSpeed \in T_Speed$

inv9 : $P_Para \in T_Para$

inv14 : $P_CrCtl_Timer \in T_Timer$

**EVENTS**

**Initialisation**

**begin**

act1 : $P_Env_ControlSignals :\in T_Env_ControlSignals$
act2 : $P_Env_VehicleSpeed :\in T_Speed$
act3 : $P_Env_Vehicle :\in T_Env_Vehicle$
act6 : $P_CrCtl_TargetSpeed :\in T_Speed$
act9 : $P_Para :\in T_Para$
act15 : $P_CrCtl_Timer :\in T_Timer$
actc1 : $P_CrCtl_State := UBAT_OFF$

**end**

**Event** $InputHMI_Change_ControlSignals \widehat{=}$
Domain: Input HMI (simulates driver related control signals)

**begin**

act1 : $P_Env_ControlSignals :\in T_Env_ControlSignals$

**end**

**Event** $VehEnv_Change_Vehicle \widehat{=}$
Domain: VehicleEnv (simulates vehicle internal signals)

**begin**

act1 : $P_Env_Vehicle :\in T_Env_Vehicle$

**end**

**Event** $VehEnv_Change_VehicleSpeed \widehat{=}$
Domain: VehicleEnv (simulates engine control)

**any**

$P_CrCtl_Acceleration$

**where**

grd2 : $P_CrCtl_Acceleration \in T_Acceleration$

**then**

act1 : $P_Env_VehicleSpeed := F_VehicleSpeed(P_CrCtl_Acceleration \mapsto P_Env_Vehicle \mapsto P_Env_ControlSignals \mapsto P_Env_VehicleSpeed)$

**end**

**Event** $CrCtl_Change_State \widehat{=}$
Machine: CrCtl (R500.1)

**begin**

act1 : $P_CrCtl_State :\in T_State$

**end**

**Event** $CrCtl_Change_Timer \widehat{=}$

**begin**

act3 : $P_CrCtl_Timer :\in T_Timer$

**end**

**Event** $CrCtl_Change_TargetSpeed \widehat{=}$
Machine: CrCtl (R500.1)

**begin**

act1 : $P_CrCtl_TargetSpeed :\in T_Speed$

**end**

**END**

If you compare the machine *SignalEval0* with the machine *CrCtl_Context1* you will notice that *SignalEval0* only contains those parts of the machine *CrCtl_Context1* which are necessary for implementing the problem diagram shown in Figure A.3.

## B.4 Refinement of Signal Evalaution Model

After having decomposed the model of the cruise control into individual subsystems we will now show how to refine these decomposed parts using the signal evaluation subsystem as an example. If you reconsider the elaboration of the *SignalEval_0* (see Figure A.3 into *SignalEval_1* (see Figure A.4), you will see that the domain *Input HMI* has been splitted into the domains *Pedals*, *Control Interface*, and *Ignition*. Furthermore, the phenomenon *P_Env_ControlSignals* has been decomposed into the phenomena *P_Env_PedalSignals*, *P_Env_ControlInterfaceSignals*, and *P_Env_IgnitionSignal*. Therefore, we have to refine the variable *P_Env_ControlSignals* into three variables representing the phenomena above. To achieve this define a new context called *c2* which extends *c1* and defines the following sets, constants, and axioms:

**CONTEXT** c2_SignalEval1

**EXTENDS** c1

**SETS**

T_Env_PedalSignals

T_Env_ControlInterfaceSignals

T_Env_IgnitionSignal

**CONSTANTS**

AF_PedalSignals

AF_ControlInterfaceSignals

AF_IgnitionSignal

**AXIOMS**

axm9 : $T_Env_IgnitionSignal = BOOL$

axm1 : $((AF_PedalSignals \otimes AF_ControlInterfaceSignals) \otimes AF_IgnitionSignal) \in T_Env_ControlSignals \rightarrowtail\!\!\!\!\twoheadrightarrow T_Env_PedalSignals \times T_Env_ControlInterfaceSignals \times T_Env_IgnitionSignal$

thm1 : $AF_PedalSignals \in T_Env_ControlSignals \rightarrow T_Env_PedalSignals$

thm2 : $AF_ControlInterfaceSignals \in T_Env_ControlSignals \rightarrow T_Env_ControlInterfaceSignals$

thm3 : $AF_IgnitionSignal \in T_Env_ControlSignals \rightarrow T_Env_IgnitionSignal$

**END**

As you can see from the definitions above the type *T_Env_ControlSignals* is now seen as a record with three tuples, namely *T_Env_PedalSignals*, *T_Env_ControlInterfaceSignals* and *T_Env_IgnitionSignal*. Thus, we are now able to use variables of these three types in machine *SignalEval1*, namely *P_Env_PedalSignals*, *P_Env_ControlInterfaceSignals*, and *P_Env_IgnitionSignals*. In order to establish a refinement relationship of the abstract variable *P_Env_ControlSignals* which is not part of the refined machine *SignalEval1* we need to add witnesses to all events which refine events of the abstract model in which *P_Env_ControlSignals* has been used. For example, in the following event we had to add a witness because this events refines an event of the abstract model in which *P_Env_ControlSignals* has been changed:

**Event** *InputHMI_Change_PedalSignals* $\widehat{=}$
Domain: Input HMI (simulates driver related control signals)

**refines** *InputHMI_Change_ControlSignals*

**begin**

**with**

P_Env_ControlSignals′ :
$P_Env_ControlSignals' = ((AF_PedalSignals \otimes AF_ControlInterfaceSignals) \otimes AF_IgnitionSignal)^{-1}(P_Env_PedalSignals' \mapsto P_Env_ControlInterfaceSignals \mapsto P_Env_IgnitionSignal)$

act1 : $P_Env_PedalSignals :\in T_Env_PedalSignals$

**end**

As you can see from looking at the witness it establishes the missing relationship between the use of P_Env_PedalSignals in the concrete event and P_Env_ControlSignals in the abstract event.

# Bibliography

[Abr09a] Jean-Raymond Abrial. Event model decomposition. Technical report, ETH Zürich, 2009.

[Abr09b] Jean-Raymond Abrial. Faultless systems: Yes we can! *Computer*, 42(9):30–36, 2009.

[Bit02] K. Bittner. *Use Case Modeling*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[But09] M. Butler. Towards a cookbook for modelling and refinement of control problems. Technical report, University of Southampton. School of Electronics and Computer Science, 2009.

[Jac95] Michael Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudices*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[Jac01] Michael Jackson. *Problem Frames: Analyzing and structuring software development problems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[JHJ07] C.B Jones, I.J Hayes, and M.A. Jackson. Deriving specifications for systems that are connected to the physical world. In *Formal Methods and Hybrid Real-Time Systems*, pages 364–390, 2007.

[Jon] Cliff Jones. DEPLOY Deliverable D15: Advances in Methodological WPs. http://www.deploy-project.eu/pdf/D15final.pdf.

[Jon81] C.B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Programming Research Group,University of Oxford, 1981.

[Jon83a] C.B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[Jon83b] C.B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[LA90] P.A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1990.

[LGR09] Felix Loesch, Rainer Gmehlich, and Christine Rossa. DEPLOY 091 - Requirements Engineering for Formal System Development. Technical report, Robert Bosch GmbH - Corporate Sector Research and Advance Engineering, August 2009.

[Maz09a] Manuel Mazzara. Deriving specifications of dependable systems: toward a method. In *Proceedings of the 12th European Workshop on Dependable Computing (EWDC 2009)*, 2009.

[Maz09b] Manuel Mazzara. Different perspectives for reasoning about problems and faults. Technical Report CS-TR No. 1151, School of Computing Science, University of Newcastle, April 2009.

[SB06] C. Snook and M. Butler. UML-B: Formal modelling and design aided by UML. 2006.