Project DEPLOY
Grant Agreement 214158

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

DEPLOY Deliverable D38

D1.2 Report on Enhanced Deployment in the Automotive Sector (WP1)

Public Document

31 October 2011

`http://www.deploy-project.eu`

Contributors:

Katrin Grau	Robert Bosch GmbH
Rainer Gmehlich	Robert Bosch GmbH
Felix Loesch	Robert Bosch GmbH
Jean-Christophe Deprez	CETIC
Renaud De Landtsheer	CETIC
Christophe Ponsard	CETIC

Reviewers:

Cliff Jones	University of Newcastle
Michael Jastram	University of Düsseldorf

Contents

Chapter 1

Introduction

The DEPLOY workpackage WP1 deals with the deployment of formal engineering methods (principally 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
- University of Düsseldorf
- ETH Zürich
- CETIC

Our main objectives in WP1 are 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.

The purpose of this deliverable is to describe how lessons learnt from the pilot deployment helped to improve the methodology for the enhanced deployment. The application of the methodology to the start stop system is presented through the different steps of the process and illustrated with examples.

The remainder of the deliverable is structured as follows: Chapter 2 introduces the enhanced deployment, a start stop system and gives an overview of its functionality. Chapter 3 presents the methodology used in the enhanced deployment. In Chapter 4 the requirements phase of the methodology is explained in detail, followed by Chapter 5, where the specification phase of the start stop system is illustrated. In Chapter 6 the plans for the Event-B model of the enhanced deployment are presented. Chapter 7 describes the contribution of the enhanced deployment to the workpackage WP11 Evidence. Chapter 8 concludes this deliverable.

Chapter 2

Enhanced Deployment

As mentioned in Chapter 1 the enhanced deployment's task is applying the methodology to a system with different characteristics than the pilot deployment to adapt and strengthen the developed methodology. During the enhanced deployment the methodology of the pilot deployment is not only applied to a different system, but adapted due to the experiences during the pilot deployment.

2.1 Overview

For the enhanced deployment in the automotive sector we chose the start stop system (SSE).
The SSE is a system that helps to save fuel and reduce emissions by turning the combustion engine off when it is not needed and turning it on again as soon as it is required.
A typical scenario is stopping at a red light. Usually the engine would keep running (consuming fuel and producing emissons) although it is not needed until the light changes to green and the driver continues the journey. If the driver stops at the red light, changes the gear to neutral and releases the clutch, the SSE turns the engine off. As soon as the driver presses the clutch again to change the gear, the SSE starts the engine and the driver can move forward as usual.
But it is not only the driver who influences the SSE. Revisiting the situation described before (driver stops at a red light, gear in neutral, clutch released) the SSE does not stop the engine if the state of charge of the battery is below a certain threshold.

2.2 Description of the Start Stop System (SSE)

During the enhanced deployment we concentrate only on the software part of the system. We do not model the whole system which would include e.g. wearing out. The use of a SSE requires a better starter, a better battery, a battery sensor, etc.. We assume that these requirements are addressed elsewhere. But we have to include the fact that we do not want to constantly start and stop the engine, but only stop it if it is reasonable to assume that it will be off for a certain amount of time. This amount of time is not only influenced by the energy needed to restart the engine but also by the wearing out phenomena.

The SSE is embedded in surrounding systems (see simplified Figure 2.1). In Section 2.1 the basic functionality of the SSE was explained. With this description it is obvious the SSE needs information about the clutch pedal, the gearbox and the electical energy management (including the battery).

There are three different kinds of systems that interact with the SSE:

1. Input system - a system that provides inputs to the SSE
2. Output system - a system that uses output of the SSE
3. Input/output system - a system that provides input to the SSE as well as uses outputs of it

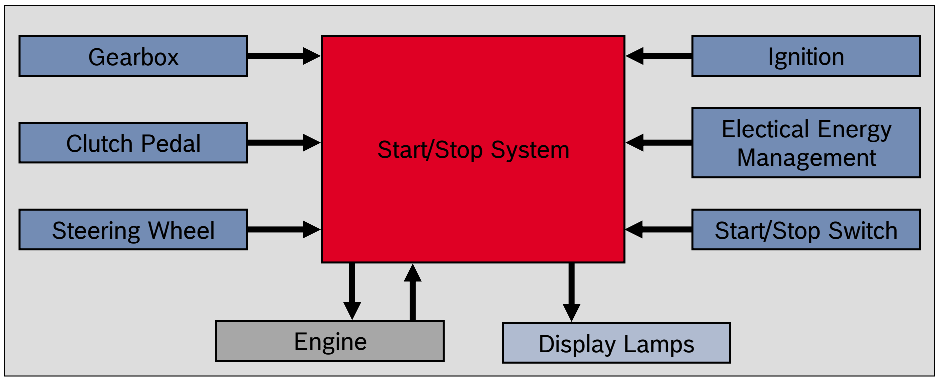


Figure 2.1: System Overview

These systems are highlighted in different colours: Dark blue for input systems, light blue for output systems, grey for input/output systems.

The SSE itself (from now on referring only to the software part) consists of three different parts:

1. A number of subsystems that provide information concerning a single aspect (e.g. electrical energy management) to the start stop coordinator (SSE coordinator)

2. The SSE coordinator which evaluates all this information and decides when to stop or start the engine

3. A Human-Machine Interaction (HMI) Display subsystem, which is responsible to provide information to the driver

Further details will be discussed in the subsequent sections, e.g. in Section 4.2, where requirements for the SSE are discussed.

Chapter 3

Methodology

The Deliverable D19 describes the methodology of using Event-B for the pilot deployment [D19] (see also [D15]). We do not want to repeat this description, but only emphasize two main issues we address there:

- The development of an Event-B model is only part of the overall development process, see the box "DEPLOY WP1" in Figure 3.1
- The gap between the informal (natural language) world and the formal world (Event-B) is too big to be easily taken in one step, see Figure 3.2

The goal of the methodology we used in the pilot deployment was to ease the development of an Event-B model and to facilitate the integration of formal methods in the overall development process.

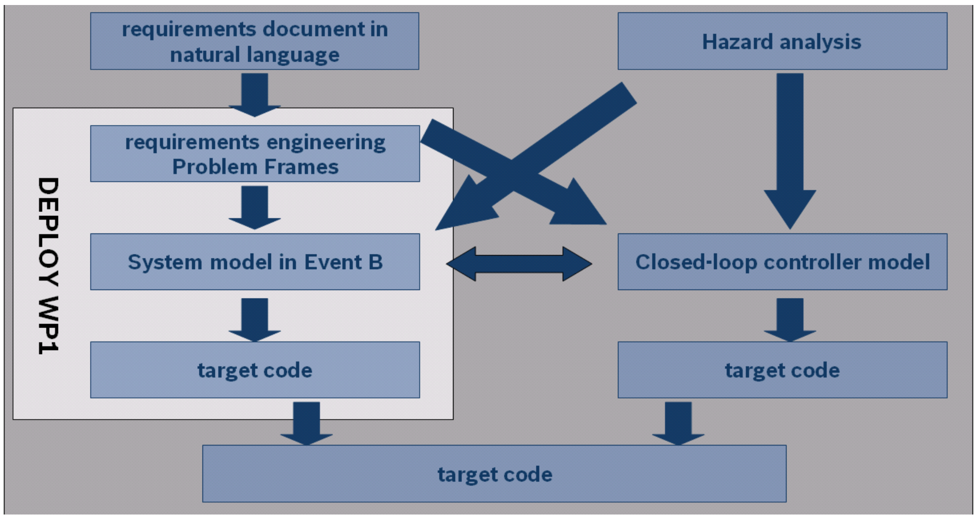

requirements document in natural language
Hazard analysis
DEPLOY WP1
requirements engineering Problem Frames
System model in Event B
Closed-loop controller model
target code
target code
target code

Figure 3.1: Overall development process - Pilot Deployment.

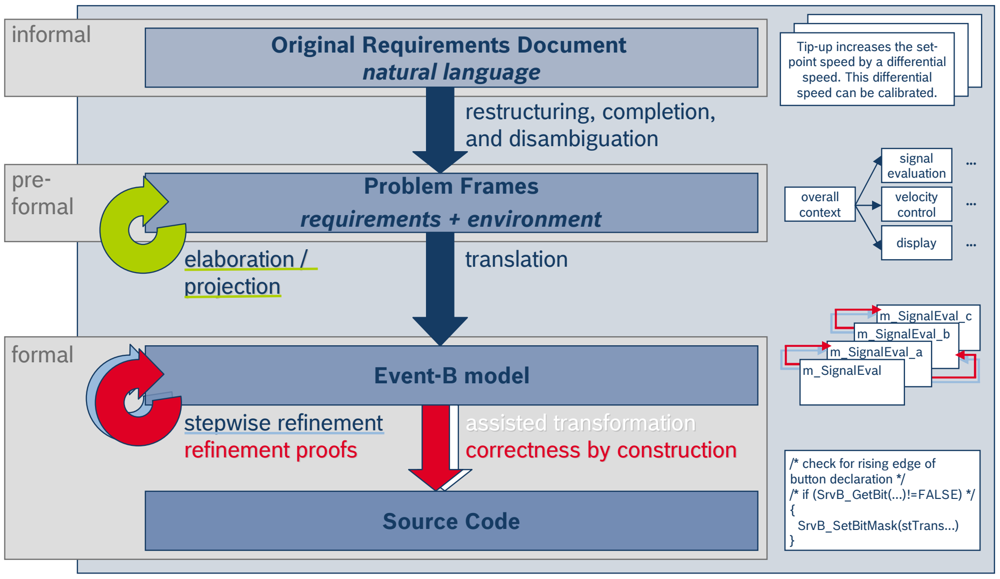

informal
Original Requirements Document
natural language
restructuring, completion, and disambiguation
pre-formal
Problem Frames
requirements + environment
elaboration / projection
translation
formal
Event-B model
stepwise refinement
refinement proofs
assisted transformation
correctness by construction
Source Code
Tip-up increases the set-point speed by a differential speed. This differential speed can be calibrated.
overall context
signal evaluation
velocity control
display
m_SignalEval_c
m_SignalEval_b
m_SignalEval_a
m_SignalEval
/* check for rising edge of button declaration */
/* if (SrvB_GetBit(...)!=FALSE) */
{
SrvB_SetBitMask(stTrans...)
}

Figure 3.2: The deployment approach of WP1 - Pilot Deployment.

3.1 Lessons Learnt from Pilot Deployment

There are several lessons learnt from the pilot deployment we tried to respect in the enhanced deployment. We will seperate them into three subsections: application, method/process and tools.

3.1.1 Application

The application for the pilot deployment was the cruise control which is an embedded real time system including a closed loop controller as an essential part. With the lessons learnt from the cruise control we chose the SSE as the second pilot application. Reasons for that are:

First of all the SSE does not include a closed loop controller. During the pilot deployment we decided to separate the development and modelling of the closed loop controller from the Event-B modelling/development process. This was done because it is obvious that Event-B does not support the development and/or modelling of closed loop controllers. The separation between the discrete and the continuous parts (closed loop controller) was successfull, nevertheless it complicated the requirement engineering (in which the separation was done) and later on the synthesis between the two development strands. This separation was done in an ad hoc manner which needs (if deployed in industry) a deeper investigation how this could/should be done and therefore a more mature method and tool support. Such a support is beyond the scope and abilities of the DEPLOY project and might be a bigger research issue itself.

The second difference between the cruise control and the SSE is the size. We underestimated the effort (during requirement engineering as well as during modelling and proving) we had to spend for the cruise control system. Due to this and the limited time for the enhanced deployment we chose a smaller application. Additonally (see tools and method subsection) the size itself (beyond the restricted resources) introduces problems into the Event-B modelling and verification.

In summary there are similarities between the two applications:

- big interface with other parts of the engine control
- calculates the driver demands from direct and indirect information
- contains a statemachine with a moderate number of states but complicated conditions
- information to the driver by lamps/display about the current state

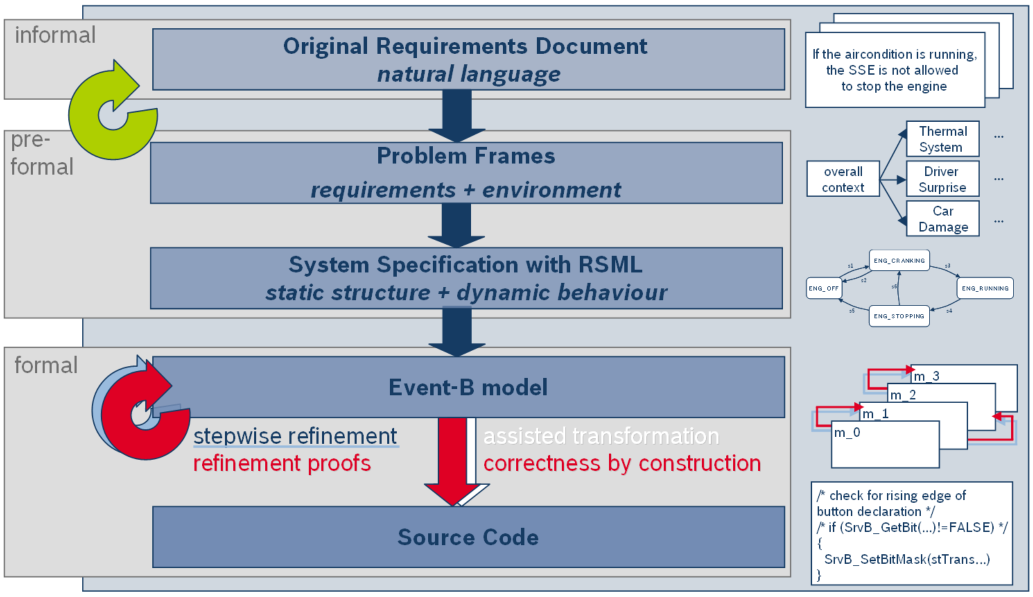
and differences:

- SSE is smaller that the Cruise Control
- The main output of the Cruise Control is a torque demand (calculated by a closed loop controller) compared to two boolean variables of the SSE.
- SSE contains no closed loop controller

3.1.2 Method/Development Process

In the pilot deployment we used extended Problem Frames [D19] for analysing the system to produce detailed requirements and a design specification. Following the requirements engineering phase, we started modelling the system in Event-B, preserving the structure we introduced during the requirements engineering step. This was done to achieve direct traceabiltiy between the requirements document and the Event-B model. For the approach taken during pilot deployment see also Figure 3.2.

For the enhanced deployment we changed this development process slighty taking into account the lessons learnt. In Figure 3.3 the general approach for the enhanced deployment is shown. The biggest change is that we introduced an additional step between requirement engineering with Problem



```
/* check for rising edge of
button declaration */
/* if (SrvB_GetBit(...)!=FALSE) */
{
  SrvB_SetBitMask(stTrans...)
}
```

Figure 3.3: Development Process

Frames and modelling in Event-B: the specification document. With the approach taken during the pilot deployment our major issue was to achieve direct traceabilty between the natural language requirements and the final Event-B model. We are successful in doing this, but we have to pay a price. There is a trade-off between traceabiltiy and architecture. In following the pilot deployment approach you will end up with a model which is traceable but - from a modelling, proving and maintainability point of view - poor. Our decision for the enhanced deployment was to use a development process which is more balanced concerning these different qualities. The solution we chose is to produce an intermediate document between the Problem Frames anaysis and the Event-B model. The purpose of this document is to produce a more solution oriented structure of the system. With the experience of the pilot deployment in mind, we chose RSML (with some adjustments, see Section 5.2) as the language for the specification document. The reason is that a natural language description of finite state machines is cumbersome.

In Figure 3.4 the artifacts we are producing during the development of the SSE are shown. We start with inital requirements written in natural language. This is the starting point for requirements analysis, done with the Problem Frames method. The result of this step is a Problem Frames model. From the Problem Frames model we produce a dedicated requirements document (a detailed description of these steps are included in Chapter 4). The requirements document and the Probelm frames model are the starting points

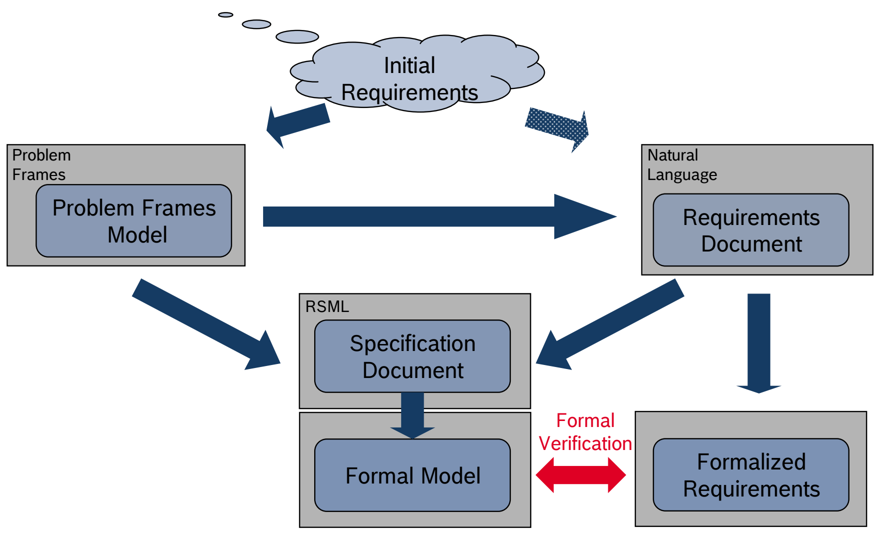


Figure 3.4: Detailed Development Process

for the system specification (see Chapter 5). The requirements document is used for the formalization of the requirements. In this step we try as early as possible to produce Event-B invariants which we want to prove later on during the Event-B modelling. Thinking about Event-B invariants in this early step is done, because we do not want to include information in the invariants which is related to the implementation. Invariants should be model (or implementation) independant as far as possible. Based on the specification document the Event-B modelling is done (see Chapter 6).

3.1.3 Tools

As mentioned in the section about the enhanced deployment application (see Section 3.1.1), the size of the application itself is a problem. During the pilot deployment the performance of RODIN was not satisfactory in dealing which such big models. There were performance issues related to the pure editing process (very slow reaction time in editing) and also performance issues related to the proving interface. As a consequence of this feedback there was a significant progress in supporting big models. Nevertheless there are still some missing features related to parallel team development of big models (e.g. version control) which prevents us from trying again modelling a application of compareable size.

Chapter 4

Requirements

To bridge the gap between the informal natural language requirements and the formal Event-B model we used an extended version of Problem Frames during the pilot deployment. For the original Problem Frames approach see [Jac01]. The extended version and its application to the pilot deployment are explained in [D19].

4.1 Lessons Learnt from Pilot Deployment

During the pilot deployment there were a lot of lessons learnt (see Section 3.1). These lessons learnt are used to adapt and improve the methodology in the enhanced deployment. In this section we will concentrate on the lessons that concern the requirements and affect this part of the development process.

- **Traceability:** The use of Problem Frames during the pilot deployment helped us to bridge the gap between informal natural language requirements and a formal Event-B model. Especially the traceability of the requirements in the Event-B model was a challenge and had to be ensured. The developed Problem Frames model was structured in different levels of abstraction which were sustained in the Event-B model. This structure eased the traceability.
- **Teamwork:** The Problem Frames model was not only stuctured in different levels but also in different subproblems within one level of abstraction. This subproblems could be modelled by different members of the team and merged afterwards.
- **Scalability:** In addition to this possiblity for teamwork the handling of the large model of the pilot deployment became easier and the tool

was better suited for these smaller models than for the complete model which merges all subproblems.

- **Structure/Architecture:** The fact that we used one structure during the whole development process also has some drawbacks. The structure is fixed early in the development and it is probably not optimal for every phase of the development process. If a perfect structure for the problem analysis has been found (which is not at all certain) it might not lead to an easy Event-B model or a convenient implementation. One has to judge if the benefit of having easy traceability is worth the strict structure. One side-effect of restricting the structure of the Event-B model to the structure of the Problems Frames is that there is no dedicated phase to think about design and architecture again.

- **Requirements and Specification:** The Problem Frames model was used in the pilot deployment not only for the development and documentation of the requirements. It was also a specification which was directly translated into an Event-B model.

4.1.1 Consequences for the new approach:

For the enhanced deployment we decided to have a requirements document (see Section 4.2.2), a Problem Frames model (see Section 4.2.1) and a specificiation document (see Chapter 5) in comparison to just a Problem Frames model during the pilot deployment (see also Figure 3.4 in Chapter 3).
We use Problem Frames for the problem analysis. In addition to the Problem Frames model a requirments document is produced. The requirements document is the basis to obtain invariants that can be used in Event-B by formalizing these requirements (see Section 4.2.3). These invariants are later used in the Event-B model (see Chapter 6). The Problem Frames model contains already information which is used in the specification document. A description of the whole development process can be found in Chapter 3.
During the enhanced deployment we decided to use a suitable structure during the Problem Frames analysis without constraining the structure for the following phases. In the requirements document we refuse to assume any design at all and describe the requirements without using internal variables.

4.2 Requirements & Problem Frames Model of SSE

To develop the requirements of the SSE the limitations and borders of the system have to be clear. We concentrate only on the software part of the system. A real SSE affects not only the software part of the car. We illustrate this with an example:
The engine of a car with a SSE is started and stopped more often than an engine of a car without a SSE. The starter has to be capable of this additional requirement. Therefore the development of a complete SSE must include the requirements for the surrounding systems (see Section 2.2). We assume for the enhanced development that these requirements are treated elsewhere and concentrate only on the software part.
For both the Problem Frames model and the requirements document assumptions have to be made. An example of such an assumption is the following

> Every error concerning the vehicle (accident, sensor error,. . .) is detected by the error handling of the car.

This assumption ensures that it is enough to include the error handling in the model of the SSE to address all kinds of errors.

Time and Timing Constraints

Before we illustrate the requirements document, the Problem Frames model and the invariants, we address one important aspect for all of them. Time and timing constraints play an important role, but they have to be adjusted to the different system borders.
In the requirements document and the Problem Frames model time and timing constraints address real physical phenomena. A (simplified) example would be

> The engine must change the status from off to running within time t if the battery charging state is below a defined limit $Para\_Battery$.

To fullfill this requirement a sensor has to check the battery charging state, pass the information to the SSE, the SSE has to check if it is below the defined limit $Para\_Battery$ and in this case send an order to start the engine to the engine control, which has to start the engine, which again needs some time. For the requirements it is convenient to address the real world phenomena, but when translating these requirements into invariants for Event-B timing constraints have to be carefully considered. The Event-B model only considers the time between the information reaches the SSE and the output leaves the SSE. Therefore timing constraints have to be adjusted. Within the SSE time is needed to produce an output according to the input. The events in the Event-B model are ordered and when formulating the invariant one has to consider this order which depends on the chosen architecture. This cannot be done before modelling the system in Event-B but has to be addressed there for further adjustment by including the order of the events.

4.2.1 Problem Frames Model of the SSE

As mentioned in Section 4.1.1 the development of the requirements is done within the Problem Frames model. The Problem Frames model is then used as basis for the requirements document (see Section 4.2.2), but contains more information than what is needed there.
The main part of the SSE is the coordinator itself. The task of the coordinator is to judge if the SSE should send a command to the engine coordinator to start or stop the engine. The foundation of these commands is information that is provided to the coordinator by different subsystems only concerning themselves. So the coordinator has to take into account the different priorities of the subsystems. In fact there are four different types of information the different systems send to the coordinator:

- Start_Enable
- Start_Request
- Stop_Enable
- Stop_Request

Start_Enable and Stop_Enable of a subsystem declare that this subsystem has no objection to switching the engine on or off respectively.
Start_Request and Stop_Request of a subsystem declare that this subsystem wants the engine to be switched on or off respectively.
Please note that not every subsystem must provide all four types of information.
In general the SSE coordinator only sends a command to stop or start the engine if all subsystems have send their permission (Start_Enable/Stop_Enable) and there is at least one subsystem that has requested this change (Start_Request/

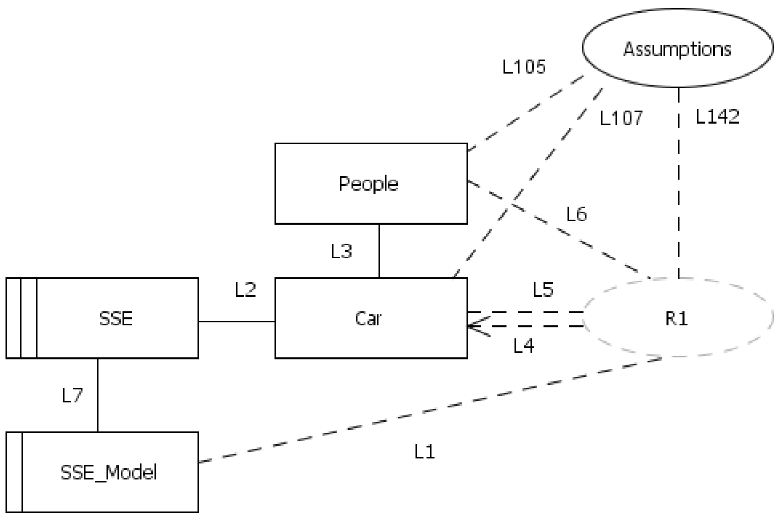


Figure 4.1: SSE_Context

Stop_Request). But of course there are exceptions.

The most abstract form of the requirement for the SSE is described in the SSE_Context diagram (see Figure 4.1). There is stated what the main function of the SSE is: To start and stop the engine with certain constraints. The assumptions the analysis of the problem is based on are documented in this most abstract diagram.
To further describe these constraints several subproblems are considered in more detail. These subproblems are Problem Frames projections. All of the subproblems only describe the situation from their point of view. To be able to combine them later we decided that each of them refers to a designed domain which records the information of this subproblem, i.e. if the subproblem wants to start or stop the engine or if it gives the permission to do so.
One of the projections is about the explicit actions of the driver (see Figure 4.2) like turning the steering wheel, pressing the clutch pedal or changing the gear. With this information it is determined if the driver is stopping the car for a certain amount of time (which will result in a stop request for the engine) or if he is preparing to move the car (which will cause a start request for the engine). The requirement that will be presented later in Section 4.2.2 is a part of this subproblem.

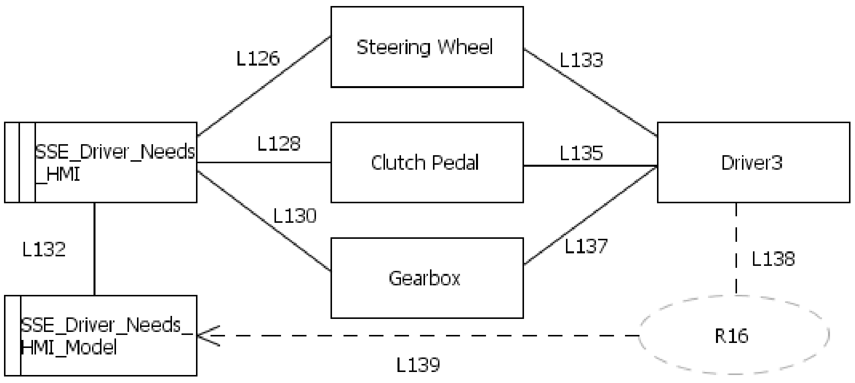


Figure 4.2: SSE_Driver_Needs_HMI

After having considered all subproblems in isolation, they have to be combined again. The SSE coordinator manages all requests and permissions for the start or the stop of the engine as well as the *INIT* status of the ECU. The *INIT* status prevents the SSE from changing the engine status. It evaluates all information provided by the subproblems, prioritises them and decides when to send a request for the start or the stop of the engine.
To summarize the structure of the SSE Problem Frames model three different kinds of diagrams can be identified:

1. Abstract diagram
2. Diagrams to extract information and to concentrate it in a designed domain
3. Diagrams to interpret and combine information

These three different kinds of subproblems play different roles in the model. One of the diagrams of the third kind is the coordinator itself. The diagrams of the second kind only consider their own needs and problems and do not care if they are in conflict with other subproblems.

Where is interesting information located?

During the modelling of the SSE the question arose where in the problem frame diagram interesting information is located. The reason was that for all the subproblem which determine how the designed domains are structured the requirements are very similar. Roughly they state something like "if

the driver wants to move the car, the variable in the designed domain should capture that". The important information about what exactly the driver does when he or she wants to move the car (which is an abstraction of the reality) is stated in the domain description of the driver. It was surprising that the requirements of the SSE itself do not reveal the interesting information about the system. To understand, build and maintain the SSE the information within the different domains is crucial. This separation is reflected in the requirements (see Section 4.2.2).
Another issue was where assumptions in the problem frames diagram would fit in. We decided to formulate our assumptions in the most abstract diagram and did not further investigate what happens, if these assumptions are not true.

4.2.2 Requirements for the SSE

To illustrate the requirements we start with the following abstract description of the SSE (see also 2.2):

- The driver is allowed to switch the SSE on and off.
- If the SSE is switched on, the system achieves fuel economy by automatically turning the engine on and off.
- It does so without adversely affecting the behaviour of the car as expected by the driver.
- It does this safely.
- The system informs the driver about the current status of the SSE by lamps.

Some parts are already quite clear, like "... informs the driver about the current status of the SSE by lamps", some parts are vague, e.g. "It does this safely". The first example is straightforward to refine: The detailed requirement has to state which lamps exist and how the status of the SSE influences these lamps. Not all internal states expose helpful information to the driver, but it is a well-defined task. The second example is much more complicated, as it does not point to a certain part of the system but has to be included in every aspect.

The next step is a detailed description of the requirements. This step is combined with the development of a Problem Frames model, which was explained in Section 4.2.1. The requirements from the Problem Frames model

(see Figure 4.2, R16) are the basis for the requirements in the requirements document, but with the difference that in the Problem Frames model the combination problem is solved in a separate diagram and internal variables (designed domains) are used.
An example of a more detailed requirement is the following about the needs of the driver and its effects for the SSE:

> (Driver_Needs_HMI) The SSE is not allowed to change the engine status from running to off if the driver wants to move the car (see 1 and 2 for "driver wants to move the car").

The obvious question is how to know the wishes of the driver, in this case, when does the driver want to move the car. This question is deliberatly addressed separately. The requirement is not referencing sensors, as it is indeed about the needs of the driver and not about how the system could know them. To emphazise this even more: If it was possible to have link to the brain of the driver, one would not bother to use information of e.g. the pedals. The problem is separated to make the intention of the requirement clear. The last step is of course to determine how the system detects the wishes of the driver, which is included in the requirements document as well. In the Problem Frames model this would be the description of a given domain (see also Section 4.2.1, Where is the interesting information located). In the requirements document this would be listed as additional information like the following example shows:

The requirement refers to the wish of the driver to move the standing car.

1. If the engine is running and the driver does not want to move the car, then the steering wheel is not used, the clutch is released and the gearbox is in neutral.

2. If the engine is running and the driver does want to move the car, then the steering wheel is used, the clutch is used or the gearbox is not in neutral.

4.2.3 Invariants for the SSE

The requirements of the SSE are translated to invariants that will be proven in the Event-B model. The idea is to have invariants before starting modelling the SSE in Event-B. They are captured in a separate document. We start with an example of such an invariant:

1	**Driver_Needs_HMI** The SSE is not allowed to change the engine status from running to off if the driver wants to move the car (see 1 and 2 for "driver wants to move the car").
	$Engine\_Status = Running \land$ $DriverNeedsMoveRunning = \mathbf{T} \land$ $Time = Para\_Response\_Time$ $\Rightarrow$ $SSE\_Stop\_Order = \mathbf{F}$

The variable $DriverNeedsMoveRunning$ is defined as follows:

$$\begin{aligned}&Engine\_Status = Running \Rightarrow \\ &\Big(DriverNeedsMoveRunning = \mathbf{T} \\ &\Leftrightarrow \\ &(Steering\_Wheel = USED \ \lor \ Clutch\_Pedal \neq RELEASED \ \lor \\ &Gearbox \neq NEUTRAL)\Big)\end{aligned} \tag{4.1}$$

When formulating the invariants we tried to avoid using internal variables. Internal variables, as the name suggests, contain internal information. Invariants should describe properties independent of the internal strutcture, similar to the requirements themselves. This is the illustrated best with an example. A phrase used in the requirements is "if the engine was switched off by the SSE". This phrase is very difficult to translate to an invariant in Event-B as it contains information about the past. The current external information would be *engine is off*, but not the reason for it. Of course it is possible to formulate this in terms of external information, e.g. there has been a request of the SSE coordinator to stop the engine *in the past*, but it is very cumbersome, as there is no direct access to this kind of information and *in the past* has to be further defined. It is easier to assume that the SSE keeps track of this information in an internal variable and we use this variable in the invariant. Please note that we always try to use external information and only allow the use of internal information if it is justified by the reduction of effort.
The formulation of the invariants is based on some assumptions. One assumption is the following:

> The SSE influences the engine only with the two variables SSE_Start_Order and SSE_Stop_Order. If they are false, the SSE does not influence the engine.

The majority of the requirements refer to the status of the engine which is not part of the SSE. The assumption provides the necessary connection.

Another assumption concerns time and timing constraints and states that the phrase $Time = Para\_Response\_Time$ used in the invariants is a macro which has to be adapted in the further development as mentioned earlier in Section 4.2 (Time and Timing Constraints).
It is obvious that it won't be possible to simply copy the invariants in the Event-B model and prove them. Some adjustment is needed to include necessary information of Event-B model of the SSE. It is important to be aware of the fact that the invariants do not describe a complete model, e.g. not all variables are described. They are only part of a formal model and validation is very important when they are included in the Event-B model.

Chapter 5

Specification

5.1 Lessons Learnt from Pilot Deployment

We learnt many lessons from the pilot deployment of Event-B to the cruise control system [D19]. During pilot deployment we directly mapped elements of the Problem Frames Model to Event-B elements. Although this direct mapping provides good traceability it has several drawbacks which are described below:

- **Requirements Analysis and Specification** During pilot deployment the Problem Frames model was used as a documentation of both the requirements and the specification. The problem was that the Problem Frames approach is better suited for requirements development than for specification. The Problem Frames approach does not provide means of describing a specification other than natural language descriptions of machine domains. Therefore our specification during pilot deployment had to be written in terms of machine descriptions that are mainly based on natural language.

- **Structure/Architecture** The decision we made during pilot deployment to use the Problem Frames model as a documentation for both requirements and specification clearly constrained us in the structure, i.e., the structure of the Problem Frames model dictated the structure of the specification because the specification was contained within the model itself. This caused problems when we mapped the structure of the Problem Frames model to Event-B. The hierarchical structure of the Problem Frames model we used during pilot deployment was in fact a stepwise refinement from an abstract design to a concrete design, i.e., a specification of the system.

- **Description of State Machines** During pilot deployment we had to specify a state machine for the cruise control system. Each transition of the state machine was described as a separate requirement in the Problem Frames model. Sometimes the conditions for a transition in the state machine were split to more than one requirement due to the subproblem structure of the Problem Frames model. When we tried to map that structure to Event-B it became very difficult to understand the resulting Event-B model because the conditions for a single transition in the state machine were scattered to multiple events in the Event-B model. Since the Problem Frames approach did not provide additional means for the specification of state machines we used an external drawing tool for drawing the states and the transitions. This meant additional work for synchronizing the conditions on the transitions in the drawing tool with the requirements text in the Problem Frames model.

- **Recombination of Subproblems** During pilot deployment we used the Problem Frames model for solving the recombination problem of subproblems. Although we managed to recombine the subproblems after several iterations of constructing different Problem Frames models we learnt that Problem Frames is in fact not the optimal method for solving the recombination problem. Problem Frames is very good for analyzing the requirements but not optimal for writing a specification.

Consequences for Enhanced Deployment
As a consequence of the lessons learnt described above we decided to introduce another process phase for the enhanced deployment, namely the specification phase (see also Chapter 3). The input to this phase is the Problem Frames model and the requirements document (as described in Chapter 4). The output of this phase is a detailed specification document describing the static structure and the dynamic behaviour of the SSE.

In addition to the introduction of the specification phase we decided to use the Problem Frames model for the description of requirements and assumptions of the environment and not for the description of the specification. We believe that this strict separation of requirements analysis and specification is a better way of dealing with these two important development tasks before the formal modelling in Event-B.

In the following we will describe the specification of the SSE in more detail.

5.2 Specification of SSE

This section describes the specification approach we used during enhanced deployment. The approach is illustrated by examples taken from the SSE specification. The specification approach we used has been inspired by the specification languages RSML [LHHR94] and SCR [HPSK78, Hen80, HBGL98] which are dedicated languages for the specification of reactive process control systems, i.e., embedded systems. These languages are especially suited for the description of state machines which is why we have selected them for the specification of the SSE.

We mainly used concepts of RSML [LHHR94] for the specification of the SSE. However, some of the concepts have been used in different ways than suggested in the original RSML approach or have been simplified to increase the understandability of the specification and make the process of writing it easier.

The specification language RSML provides concepts for the description of the static structure of the controller and concepts for the description of its dynamic behaviour. In the following we will explain the concepts in more detail using examples of the specification of the SSE.

5.2.1 Static Structure

In RSML the static structure of the software system is described using so called *components*, i.e., separate encapsulated blocks of the *software system*. Each *component* has an *interface definition* that describes the *inputs* and *outputs* of the component. *Variables* with defined *types* are used for the description of inputs and outputs.

In order to distinguish these inputs and outputs from the system inputs and outputs they are referred to as *component inputs* and *component outputs*. Please note, that *component outputs* of one component can be *component inputs* of another component or *system outputs*. Likewise, *component inputs* of a component can either be *component outputs* of another component or *system inputs*.

Figure 5.1 illustrates the relationship of *system inputs*, *system outputs*, *component inputs*, and *component outputs* using a fictious software control system with two components A and B. As you can see from Figure 5.1 the system inputs I_C are at the same time *component inputs* of component A (I_A) and the component outputs of B (O_B) are system outputs O_C. Furthermore, the outputs of component A (O_A) are at the same time inputs of component B (I_B).

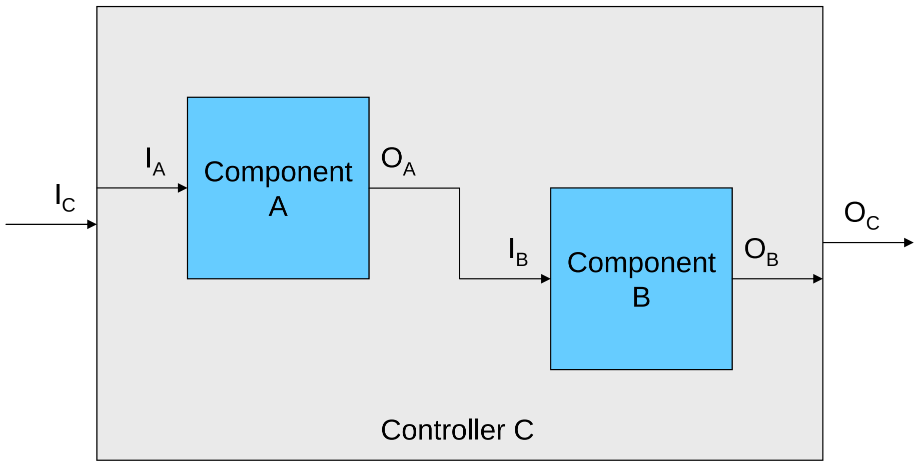


Figure 5.1: Controller and its components with inputs and outputs.

Figure 5.2 shows the static structure of the SSE. We also used this structure in order to structure the specification document.

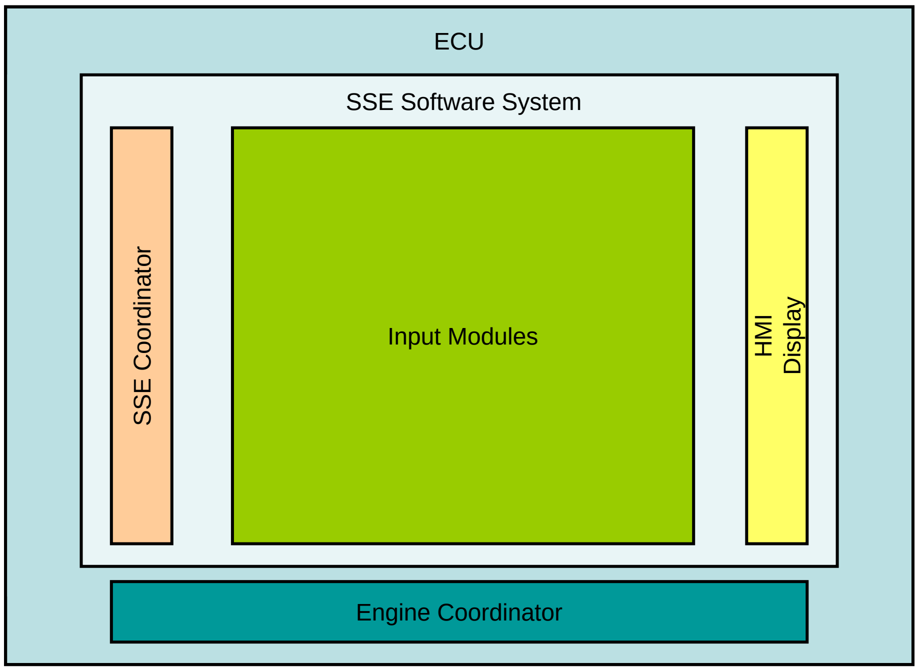


Figure 5.2: Static Structure of SSE

As you can see from Figure 5.2 the SSE consists of three different classes of components (input modules, SSE coordinator, and HMI display). The `input modules` are responsible for generating stop requests, stop enable, start requests and start enable signals for the SSE coordinator. The `SSE coordinator` is responsible for calculating the overall stop or start order taking the request and enables as an input. The `HMI Display` is responsible for producing information for the driver, signaling the current status of the SSE. The `engine coordinator` is not part of the SSE. It provides information about the current status of the pysical engine for the SSE coordinator and is responsible for taking care of start and stop order produced by the SSE coordinator.

Example: In order to illustrate the specification of component interfaces we will use the input component dealing with conditions when to start or stop the car such that the driver will not be surprised. The purpose of this component is to compute the two outputs *Driver_Surprise_Strt_Ena* and *Driver_Surprise_Stop_Ena* based on several inputs (for description of the input signals see Section 5.2.2):

Driver Surprise

Inputs
The component Driver Surprise has the following inputs:

- Clutch_Pedal : [RELEASED | PRESSED_10 | PRESSED_90]
- Gearbox : [NEUTRAL | NOT_NEUTRAL]

Outputs
The Driver Surprise has the following two outputs:

- Driver_Surprise_Strt_Ena : BOOL
- Driver_Surprise_Stop_Ena : BOOL

Internal Variables
none

Sometimes it may be necessary for a component to define so called *internal variables*. An *internal variable* of a component is only used inside the component and cannot be used as an input for other components. An *internal variable* is an auxiliary function defined on input variables, states or other internal variables. Internal variables thus help to make the specification concise. Instead of writing the conditions on input variables in many places inside the component (e.g. for the specification of transition conditions) one can use the internal variable instead. In SCR [HPSK78] an internal variable is referred to as a *term* whereas in RSML [LHHR94] internal variables are called *macros* or *functions*. Internal variables are modelled in the same way as output variables, i.e., one needs to write an *assignment specification* for internal variables (see Section 5.2.2).

The specification is not yet complete since we have not stated the dynamic behaviour of the component, i.e. the relationship between the values of the outputs and the values of the inputs.

5.2.2 Dynamic Behaviour

RSML provides several concepts for the description of the desired dynamic behaviour of the software system (respectively its components). We have used and extended these concepts. In the following we describe the basic concepts and our extensions in more detail.

AND/OR Tables

In RSML the values of controller outputs are described by *conditions* on the controller inputs (respectively the values of component outputs are described by conditions on the component inputs). A *condition* is a predicate logic statement over one or more system elements, i.e., a *variable* or a *state* of a state machine.

One possibibility to specify these conditions is to simply use a propositional logic notation with $\wedge$ and $\vee$ (e.g. $((x > 0) \wedge (y < 1)))$ However, the inventors of RSML discovered that the conditions for specifying the relationship between inputs and outputs are often complex not in the logical operators used but in the number of different conditions. To overcome this problem RSML uses a tabular representation in disjunctive normal form (DNF) that is called AND/OR tables for the specification of the conditions. This concept is easier to handle than a long list of conditions in propositional logic and has been used successfully for the specification of the Transition Collision Avoidance System (TCAS).

Example: Table 5.1 shows an AND/OR table. The far-left column of the AND/OR table lists the logical phrases. Each of the other columns is a conjunction (logical AND) of those phrases and contains the logical values of the expressions. If one of the columns is true, then the table evaluates to true. A column evaluates to true if all of its elements match the truth values of the associated predicates. A dot denotes "don't care".

X > Y	**T**	**F**	•
A < B	**T**	**F**	•
S = PRESSED	•	**T**	**T**
Y = ON	•	•	**T**

Table 5.1: AND/OR Table

This table evaluates to true in the following cases:

1. $((X > Y) \wedge (A < B)) \vee$
2. $((X \leq Y) \wedge (A \geq B) \wedge (S = PRESSED)) \vee$
3. $(S = PRESSED) \wedge (Y = ON)$

Assignment Specifications

An *assignment specification* describes the assignment of values to an output variable or an internal variable of a component based on certain conditions on the input variables or other internal variables. In order to describe these conditions an AND/OR table is used (see above for a detailed explanation of AND/OR tables).

Example: The following table shows the assignment specification for the output variable *Driver_Surprise_Strt_Ena* of the component handling driver surprise. The conditions of the input variables are described by an AND/OR table.

Assignment: **Driver_Surprise_Strt_Ena**

Condition: k

Clutch_Pedal = PRESSED_90	•	**T**
Clutch_Pedal = PRESSED_10	**T**	•
Gearbox = NEUTRAL	**T**	•

Action(s): Driver_Surprise_Strt_Ena := TRUE

Condition: $\neg k$

Action(s): Driver_Surprise_Strt_Ena := FALSE

The assignment specification states that the output variable *Driver_-Surprise_Strt_Ena* is assigned the value TRUE iff the input variable *Clutch_Pedal* is pressed by more than 90 percent (PRESSED_90) and the input variable *Gearbox* is NOT_NEUTRAL or iff the *Clutch_Pedal* is pressed by more than 10 percent and less than 90 percent (PRESSED_10) and the *Gearbox* is NEUTRAL. The output variable *Driver_Surprise_Strt_Ena* is assigned the value FALSE iff the conditions that were specified for the TRUE case do not hold.

State Machines

The desired dynamic behaviour of a component can be described by a *state machine* if a direct *assignment specification* of the outputs of the component is too complicated. For the specification of the SSE we used *state machines* in a similar way but in a simpler form than defined by the RSML specification language [LHHR94]. For our purposes a *state machine* consists of *states* connected by *transitions. States* have a unique *name* and a descriptive *comment* that describes the *state. Transitions* define how to get from one state to another. A *transition* is guarded by one or more *conditions* and may or may not have an associated *transition action.*

A *state machine* of a component is modelled as an internal or output variable with an enumerated set as a type. Each *state* of the *state machine* is represented as a member of this enumerated set.

Example: In the following we show the state machine for the *engine coordinator* an external component of the SSE which coordinates the engine. The state machine of the engine coordinator has the following states:

- ENG_OFF
- ENG_CRANKING
- ENG_RUNNING
- ENG_STOPPING

We modelled the *states* as an enumerated set with the values *ENG_OFF*, *ENG_CRANKING*, *ENG_RUNNING*, and *ENG_STOPPING*. A state variable *Engine_State* with the enumerated set as a type has been used to model the state machine itself.

Transition Specification

As with all state-machine models, transitions between states are governed by *conditions* on inputs and the current state of the modelled system. A *transition* usually has some *actions*, i.e., assignments to values. In RSML [LHHR94], a transition is also guarded by a *triggering event* which can be used to order the execution of transitions in parallel state machines. However, in order to keep our specification as simple as possible we have decided not to use *triggering events*. Instead of *triggering events* we use an explicit separate description of the execution order of assignments and transitions which is described below.

In many state-machine models the state machine is depicted graphically. Typically the states are shown as *boxes* and *arrows* represent the possible transitions between the states. This is also the standard representation for Mealy automatons. The *guarding conditions* under which the transitions are taken and the *transition actions* are usually written as labels on the transition arrows. However, if the *guarding conditions* and *transition actions* are complicated and very long, the graphical state machine diagram will get easily cluttered and it will become more difficult to understand the state machine.

In order to solve this problem, RSML [LHHR94] and SCR [HPSK78] use tabular notations for describing the *guarding conditions* and *transition actions* in addition to the graphical depiction of the state machine. In SCR so called *mode transition tables* are used to specify the transition conditions. RSML [LHHR94] uses AND/OR tables for the specification of transition conditions.

Since we have made similar experiences with complicated *conditions* and *actions* in our case studies we decided to use the RSML approach with AND/OR tables for the specification of transition conditions for the SSE.

Example: Figure 5.3 shows the state diagram for the engine coordinator. Instead of writing the conditions directly as labels on the transition, each transition is labelled with a unique *transition identifier* (s0 to s6 in our example). The *conditions* and *actions* for each transition are specified using an AND/OR table.

The following transition specification shows the *guarding conditions* and *transition actions* for transition *s1* from state ENG_OFF to ENG_CRANKING and the self-transition *ss1*.

Statemachine: **Engine Coordinator**
State: **ENG_OFF**

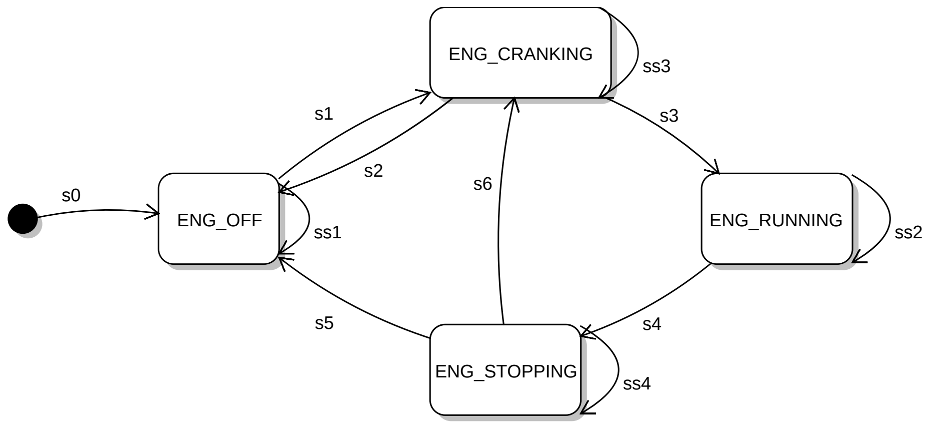


Figure 5.3: State Diagram for Engine Coordinator.

Transition: **s1: ENG_OFF → ENG_CRANKING**

Condition: $c2$

Ignition	**T**	•
SSE_Start_Order	•	**T**

Action(s): None

Transition: **ss1: ENG_OFF → ENG_OFF**

Condition: $\neg c2$

Action(s): None

The transition specification states that the engine state changes from *ENG_OFF* to *ENG_CRANKING* iff the *Ignition* is TRUE or *SSE_Start_Order* is TRUE. These conditions are denoted by $c2$. In all other cases denoted by $\neg c2$ the engine state stays *ENG_OFF*, i.e., iff *Ignition* is FALSE and *SSE_Start_Order* is FALSE.

Execution Order

Whereas RSML [LHHR94] uses *triggering events* to explicitly specify the evaluation order of parallel state machines, we used a separate specification of the execution order in which the assignments and state machines are executed. In RSML, an assignemnt or transition of a state machine is not executed until an explicit event is generated. When the assignment or transition is executed, additional events may be generated as actions. In this way, the events propagate through the system triggering assignments and transitions in different components.

In our specification of the SSE we did not use *events* for ordering assignments or transitions of state machines. Instead of specifiying *events*, the execution order of assignments and transitions of state machines is determined by an *execution order specification* which is a mandatory specification element for each component. Each *assignment* and *state machine* of the component must be ordered with regard to the other *assignments* and *state machines* of the component using the following *syntactic ordering constructs*

- Parallel Operator (||): the assignments/state machines conjoined by this operator can be executed in parallel or in any possible sequential order
- Sequential Operator (.): the assignments/state machines conjoined by this operator must be executed sequentially (left first)

Execution of a state machine means that one of the associated transitions of the state machine is taken.

Example: The component *driver surprise* of the SSE calculates start enables and stop enables based on conditions that will not surprise the driver, i.e., a start enable or stop enable is only issued if the driver will not be surprised. The component has assignment specifications for the following five variables *SSE_Max_Veh_Speed_Since_Strt*, *SSE_Max_Eng_Speed_Since_Strt*, *SSE_Mode_Prev*, *Driver_Surprise_Strt_Ena*, and *Driver_Surprise_Stop_Ena*. The execution order for these assignments is specified as follows.

Order: $[SSE\_Max\_Veh\_Speed\_Since\_Strt \; ||$
$SSE\_Max\_Eng\_Speed\_Since\_Strt]$.
SSE_Mode_Prev .
$[Driver\_Surprise\_Strt\_Ena \; ||$
$Driver\_Surprise\_Stop\_Ena]$

This order states, that the assignments to SSE_Max_Veh_Speed_Since_Strt and SSE_Max_Eng_Speed_Since_Strt are executed in parallel, before the assignment to SSE_Mode_Prev, before the parallel assignments to Driver_-Surprise_Strt_Ena and Driver_Surprise_Stop_Ena.

Similarly to the specification of the execution order of assignments and state machines we can also specify the execution order of *components*. We simply use the same syntax as for the specification of the execution order of assignments and state machines.

Example: The SSE consists of a number of different components (see Figure 5.2). The input modules can all be executed in parallel but must be executed before the SSE coordinator. The execution order for the components of the SSE can thus be specified as follows:

Order: [Input Modules || HMI Display] . SSE_Coordinator

Time

In the specification of the SSE we also had some conditions that were based on the time that had elapsed since a specific state of a state machine had been entered. We decided not to use the concept of *timers* for specifying time. Instead we used a concept of RSML [LHHR94]. In RSML, an implicit global time is available which can be referred to by the greek letter τ. In order to measure the time that has been elapsed since a specific state has been entered, we record the point in time at which the state has been entered. This is referred to as $\tau(\mathit{entered\_state}(\mathit{state}))$. One can now compare the current global time τ with the point in time at which a specific state has been entered.

Example: In the SSE we had to specify conditions under which a stopping of the car does not surprise the driver. One of these conditions was that the start stop system is only allowed to stop the car if the engine has been running since a minimum time *Para_Min_Time_Since_SSE_Start*. We modelled this condition as follows:

Assignment: **Driver_Surprise_Stop_Ena**

Condition: l

$\tau \geq (\tau(\text{entered\_state(SSE\_OPERATION)}) + \mathit{Para\_Min\_Time\_Since\_SSE\_Start})$	**T**

Action(s): Driver_Surprise_Stop_Ena := TRUE

Condition: $\neg l$

Action(s): Driver_Surprise_Stop_Ena := FALSE

As you can see from this assignment specification we compare the global time τ against the point in time at which the state SSE_OPERATION (engine running) has been entered plus *Para_Min_Time_Since_SSE_Start*, i.e., the variable *Driver_Surprise_Stop_Ena* is assigned the value TRUE iff more than *Para_Min_Time_Since_SSE_Start* has elapsed since the state SSE_OPERATION (engine running) has been entered.

Chapter 6

Event-B

6.1 Lessons Learnt from Pilot Deployment

During pilot deployment we went from the Problem Frames Model to an Event-B model without an intermediate specification. We directly mapped Problem Frames elements to Event-B elements and used data refinement in Event-B. Details of our Event-B modelling approach for pilot deployment are described in the DEPLOY deliverable D19 [D19]. We learnt the following lessons with regard to Event-B. Some of these lessons are also described in a paper that will be presented at ICFEM 2011 [GGH+11]:

- **Refinement Strategy** Event-B supports horizontal (superposition) and vertical (data) refinement. Both refinement strategies can be used in an Event-B model. Typically, horizontal (superposition) refinement is used for augmenting the model with additional events and variables whereas vertical (data) refinement is used to refine abstract data types. We started with an abstract model containing abstract data types and abstract events corresponding to the abstract context diagram in our Problem Frames model. We then applied vertical refinement to make those abstract data structures more concrete. Although Event-B in principle supports vertical refinement, we had difficulties in finding the best way to do the vertical refinement. In the end we used a special form of instantiation of abstract data types for modelling the relationship between abstract and concrete types. Since this special form of instantiation was not directly supported by the RODIN tool we had to manually add axioms to the context that were both difficult to understand and cumbersome to maintain. The lesson we learnt during pilot deployment was that RODIN is better suited for horizontal refinement than for vertical refinement.

- **Modularization** The Problem Frames model we built for the case study for pilot deployment contained several subproblems on different abstraction layers (see deliverable D19 for details [D19]). The subproblems in our Problem Frames model were linked by elaboration and projection operations. We wanted to map this structure to the Event-B model. Although we were able to model the different abstraction levels of our Problem Frames model using Event-B refinement it was very difficult to model projections of subproblems with Event-B means. Besides refinement Event-B and the RODIN tool provide very little additional structuring mechanisms. Especially a clear modularization concept found in many programming languages and even in classical B is missing in Event-B. The decomposition approaches and the modularization approach provided by RODIN plugins were also not suited for our needs. Finally, we had to manually split Event-B machines into separate machines, independently refine these machines and later combine them to achieve what we had modelled in the Problem Frames model. Again this was only possible with a lot of manual and hard work. The lesson we learnt from this was that we need to carefully investigate ways of structuring our Event-B model for enhanced deployment.

- **State Machines** The Problem Frames model of the case study used for pilot deployment contained a large state machine. The transitions of this state machine were stated as requirements in the Problem Frames model. Due to the structure of our Problem Frames model which consisted of different subproblems the requirements of the state machine were not located in a single place. Our Event-B model closely followed the structure of our Problem Frames model. Each requirement describing a required transition of the state machine was modelled as a separate Event-B event. Sometimes a requirement only described part of the conditions required for a transition because the other conditions were described in a different subproblem or several transitions. This resulted in consistency problems between the different machines in our Event-B model. In the end consistency between machines was only made possible by a lot of manual and hard work. The lesson we learnt was that modelling state machines in Event-B is not an easy task and requires careful thinking of how the transitions of the state machine should be modelled. Especially one needs to think about how refinement and modelling of state machines play effectively together. Ideally there should be dedicated tool support for modelling state machines in Event-B.

- **Flow of Events** The execution semantics of Event-B does not prescribe a strict order in which the events are executed. The execution semantics of Event-B defines that events whose guards are true can be taken. If the guards of more than one event are true one event is chosen non-deterministically for execution. However, the system we modelled for pilot deployment required a strict ordering of events. In oder to make the model deterministic we added auxiliary variables in the last refinement step to the events that define a strict order. This step was necessary because we wanted to prove reactive properties of the system, i.e. properties that involve a sequence of events. The flow plugin [wik] provided by the RODIN tool helped us to graphically specify a desired order of the events and later prove that this desired order is possible. Counter examples generated by the ProB plugin helped us to fix the model in cases in which proof obligations generated by the flow plugin could not be proven because of bugs in the model. One of the problems we faced during pilot deployment was that the flow POs got very big since we introduced them in the last refinement step. The lesson we learnt was that the flow of events should be introduced earlier in the process.

6.2 Plans for Enhanced Deployment

Our plans for enhanced deployment are based on our experiences with Event-B during pilot deployment and take the lessons learnt described in the previous section into account. In the following we will describe these plans in more detail and discuss the advantages and disadvantages.

6.2.1 Refinement Strategy

Description

For enhanced deployment we plan to use the traditional Event-B horizontal refinement strategy in which the development starts with a very abstract Event-B model and details are added gradually by refining this abstract model. For the SSE we plan to start the Event-B modelling with an abstract version of the SSE coordinator. This abstract model contains only events that assign values to the outputs of the SSE. The following listing shows this abstract Event-B model which contains two events without guards that nondeterministically assign values to the two main outputs of the SSE.

An Event-B Specification of m0
Creation Date: 10 Aug 2011 @ 03:23:32 PM

MACHINE m0
VARIABLES
  SSE_Start_Order
  SSE_Stop_Order
INVARIANTS
  inv1 : $SSE\_Start\_Order \in BOOL$
  inv2 : $SSE\_Stop\_Order \in BOOL$
EVENTS
Initialisation
  begin
    act1 : $SSE\_Start\_Order := FALSE$
    act2 : $SSE\_Stop\_Order := FALSE$
  end
Event $Change\_SSE\_Start\_Stop\_Order \mathrel{\widehat{=}}$
  begin
    act1 : $SSE\_Start\_Order :\in BOOL$
    act2 : $SSE\_Stop\_Order :\in BOOL$
  end
END

In further refinements more and more details are added to the model. These refinements include the introduction of new variables, guard strengthening of existing events, and the introduction of new events. Figure 6.1 shows an exemplary refinement chain using horizontal refinement and four Event-B machines named m0 to m3.

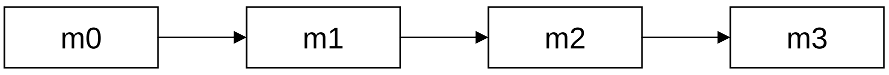


Figure 6.1: Horizontal Refinement Chain

For subsequent refinement levels two different strategies are possible:

1. Add abstract events for all other modules and then gradually refine these abstract events

2. Add one module per refinement level

In the first strategy one would add abstract events for all input modules (see Figure 5.2) and then refine these events by making them more concrete. In the second strategy one would start with one component shown in the component diagram of the specification and then add the other modules in subsequent refinement steps. Since these modules are independent there is no strict order in which each component must be introduced during refinement.

Discussion

Our refinement strategy closely follows the horizontal refinement strategy provided by Event-B. The main advantage of this refinement strategy is that the method and the RODIN tool support that strategy very well. However, the horizontal refinement strategy does not address the problem of modularizing the Event-B model. In each refinement level more and more variables and events are added to the model. In the end the model will become very big and difficult to understand and handle. Another problem with horizontal refinement is that it is not obvious how team-work can be supported, i.e., multiple persons working on the same model. For example, it is not obvious how to prepare later refinements independent of the more abstract machines. This is a clear disadvantage of the horizontal refinement strategy.

6.2.2 Modularization

Overview

Another Event-B modelling strategy besides horizontal refinement is to independently model components of the specification as separate Event-B machines and later combine them. As we have described in Chapter 5 the components of the SSE communicate via shared variables, i.e., components have inputs and outputs and the outputs of some components are inputs of other components. This type of communication should also be used in the Event-B model, i.e., a main model that communicates with other Event-B modules using shared variables.

We analyzed whether we could use the modularization plugin to support our planned modularization strategy. However, the modularization plugin for RODIN uses a different concept for communicating between modules and the main model. Instead of using shared variables the modularization plugin provides so called interfaces that can be used to model pre- and post-conditions of functions. The main model is allowed to call functions of the module and receives return values that can be used in assignments to variables.

Discussion

The modularization strategy has the advantage that each component described in the specification can be modelled independently of the other modules. This is a clear advantage over the horizontal refinement strategy in which later refinements need to have the abstract model as a reference, i.e., it is very cumbersome to prepare later refinements in isolation and later integrate them into the refinement chain. In other words, the modularization strategy is clearly better suited for team development than the horizontal refinement strategy.

However, a big disadvantage of the modularization strategy is that currently there does not exist tool support for our strategy. The modularization plugin provided for the RODIN tool currently only supports function calls as a mechanism for communicating between modules and the main Event-B model.

6.2.3 State Machines

Overview

For enhanced deployment, i.e., the modelling of the SSE, we plan to use the new Event-B statemachine plugin (see [wik]). This plugin allows the modeller to graphically draw a state machine using boxes representing states and arrows representing transitions between these states. The plugin also supports hierarchical state machines and Event-B refinement. The graphical state machine can be translated into Event-B. Each state is hereby modelled as a separate boolean variable named by the state which is set to TRUE when the state is entered and set to FALSE when the state is exited.

Discussion

The advantage of using the Event-B statemachine plugin over manually modelling state machines in Event-B is that it provides a graphical visualization of the state machine. During pilot deployment we often had difficulties validating whether the modelled events corresponded to the requirements because both were only available in textual form. For example, it was very difficult to check whether all transitions for a given state had been correctly modelled in Event-B. Another advantage of using the Event-B statemachine plugin is that it also supports animation of state machines using ProB. We believe that the use of the Event-B statemachine plugin helps to solve the problems we encountered in manually modelling state machines during pilot deployment.

6.2.4 Flow of Events

As described in Chapter 4 the invariants we want to prove of our SSE model involve reactive properties, i.e., a sequence of events. In order to model that in Event-B we require a strict ordering of events. We plan to use the new version of the flow plugin [wik] for enhanced deployment. This new version supports a graphical specification of a desired order of events and automatically generates the required auxiliary variables for the Event-B model as well as proof obligations that show that the model fulfills the desired order of events.

Discussion

The advantage of using the new version of the flow plugin over the old version is that it automatically generates the auxiliary variables. This alleviates the tedious task of validating whether the manually added auxiliary variables correctly model the graphically specified flow of events. We believe that the new version of the flow plugin will make the process of defining a strict ordering of events for our SSE model easier.

Chapter 7

Evidence Consolidation

During the enhanced deployment new contributions were identified to enrich the evidence repository and the associated industrial FAQ (Frequently Asked Questions). These were jointly discussed between CETIC and Bosch. This chapter summarised the main contributions which are related to first, reuse concerns, second, impact on the system development process and third, tool support.

7.1 Reuse issues

R-EA-1: When using a formal method efficiently, does it become more natural to design generic, reusable components than when using non formal methods?

The second pilot being studied during the enhanced deployment phase is a start and stop system. About reuse, there were no model artefact reused from the first pilot (cruise control) given the systems are quite different (closed loop controller vs input/output controller). The reuse considered here is at the process level.

Consequently, there is currently no evidence of model artefact reuse. However model reuse could be considered in the automotive domain between more similar systems for example the cruise controller and the speed limiter.

R-PQAM-1: Does the potential of reuse increase when formalism is used efficiently?

At the process level, the answer is yes: Bosch reused the methodology that was developed for the first pilot with some additional improvements to cope with identified limitations.

The methodology is based on a progressive formalisation approach from requirements expressed in natural language to requirements expressed as problems frames and then to formal models in Event-B. In order to improve the transition to Event-B, an additional formalism based on RSML (Requirements State Machine Language) was introduced. This allows analysts to better specify state and transition-based systems. Previously finite state machines had to be directly traced from natural requirements to Event-B resulting in poor maintainability. With RSML finite state machines can now be specified in an explicit formalism before being translated in Event-B.

7.2 Impact on the System Development Process

CIF-HM-2: How do organizational procedures used in various system development life cycle processes need to be adapted when formal methods are introduced?

When deploying formal methods in the automotive sector, it was really important to design a method enabling a progressive formalisation process. A large effort was devoted to designing an approach to a gradual introduction of formalism during the first pilot. This approach could extensively be reused during the second pilot with some improvement (see R-PQAM-1 on reuse).

The main drivers for adapting the development procedure are the following:

- Introduce formalism progressively using an intermediate semi-formal notation not requiring an expertise in formal methods.
- Identify key properties early. Those will typically results in invariants and generate proof obligation at the most formal level.
- Preserve traceability between the informal, the semi-formal and the formal models.
- Enable teamwork on large models

EM-PQAM-2: Does the use of a formal engineering model have any beneficial effect on requirements and design traceability?

In the case of the automotive pilot and enhanced pilot, the answer is a clear yes.

About requirements quality, the development process deployed at Bosch is converging towards a development process using increasingly formal notation throughout the development process described in Figure 3.3.

The process described is however not a cascade process but can result in iterations. The benefit of each formalisation step is to discover potential ambiguities, incompleteness, inconsistency and trigger corrections of the previous models, resulting in better requirements. Evidence of quality improvement was already documented presented during the first pilot and is also expected at the end of the enhanced deployment. At this point the enhanced deployment is not yet finished, the final assessment and integration deliverable (D43/JD3) will report about the benefits of the methodology developed in the automotive sector.

About requirements traceability, being able to trace requirements down to the Event-B model was a challenge and had to be ensured. The developed Problem Frames model was structured in different levels of abstraction which were sustained in the Event-B model. This structure eased the traceability.

During the enhanced deployment, the introduction of RSML as domain specific language resulted in a richer and more structured traceability with respect to finite state machine requirements. However this extra level of specification is also increasing the effort required to build and maintain the global traceability.

7.3 Known Strengths and Weaknesses of Tools

TOOL-EA-1: How reliable are the tools supporting a particular formalism? Are they able to carry their tasks without crashing and causing additional delays? Do they often corrupt their data, incurring additional hidden costs?

As mentioned in the chapter about the enhanced deployment application (see Chapter 3.1.1), the size of the application developed by Bosch was initial a significant problem. During the pilot deployment the performance of RODIN was not satisfactory in dealing with large models. There were performance issues related to the mere editing process (very slow reaction time in editing) and also performance issues related to the proving interface. As a consequence of this feedback there was a significant progress in supporting large models, although improvement is still needed. The tool development process is now adapted to integrate performance testing during the qualification phase of the release process.

Nevertheless a number of important features not directly related to verification but still very important for industrial adoption are not yet addressed well enough; those are related to: configuration management, version management, variant management, and team development.

7.4 General Modelling Language and System Concepts

G-EA-1 - What important system concepts (such as real time, concurrency, fault tolerance, probabilities, continuous behaviour, finite/infinite instances) can be handled "elegantly" with a selected formal method?

Considering Event-B, this formalism has shown limitations to model continuous behaviour (e.g. closed loop controller) and real-time constraints which are needed in the automotive sector. Other complementary formalisms/ tools should be considered to reason on those aspects. The complexity and overhead of managing extra formalisms and tools must also be taken into account.

Chapter 8

Conclusions

During pilot deployment we were focused on finding a solution for bridging the gap between the informal and formal world. We used (an extended version of) the Problem Frames method for requirments engineering to solve this problem. Our main issue was to preserve the traceability between natural language requirements and the final formal model in Event-B. By doing this we also increased the quality of our requirements. Beside the success we had with this approach, we also identified some drawbacks or open issues. During the enhanced deployment we try not only to repeat the previous developed method (this would have been an option if everything was perfect). Our goal was to reuse steps or methods which work well and try out alternatives in the cases we identified problems.

In the enhanced deployment we changed our development process by introducing a dedicated specification document written in a tailored version of RSML. Also we introduced an identification and (Event-B) specification of invariants. The results of these steps are a better support of good design and systematically deduced invariants which are independant of the concrete model. To achieve this we move our focus away from traceability towards design.

There are several overall conclusions (taking into account both the pilot and enhanced deployment) about Event-B and the WP1 work itself.

Cloosed loop controller There is no practible way of modelling closed loop controller in Event-B. This is a complete open field with lot of research. One of the main questions is, is it useful to try this and if yes how could this be done.

Time Time was from the very beginning an open issue and is still. In the embedded systems market there is a strong need to include at some point during the development a notion of time.

Gap between informal and formal world We had made encouraging progress in the WP1, which gives us confidence that this problem is (or could be) solved. What is needed here is more experience in applying the developed methods and strong tool support.

Formal modelling We made significant progress in formal modelling industrial size applications. Beside the progress especially the cruise control application exposed the borders of formal modelling and the tools support for formal modelling in RODIN. There is room for improvement.

Industrial development process There is an open issue left in supporting state of the art industrial development processes. Especially the supporting processes (configuration managment, variant managment, team development, version managment) are not (or not good enough) supported by RODIN.

To conlcude the last deliverable in the responsability of WP1, we (the main authors of Bosch) want to use the opportunity to thank all the people which made contributions to the work and success of WP1 and the DEPLOY project itself.

Bibliography

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

[D19] DEPLOY Deliverable D19: D1.1 Pilot Deployment in the Automotive Sector WP1. http://www.deploy-project.eu/pdf/D19-pilot-deployment-in-the-automotive-sector.pdf.

[GGH$^+$11] Katrin Grau, Rainer Gmehlich, Stefan Hallerstede, Michael Leuschel, Felix Loesch, and Daniel Plagge. On the difficulty of fitting a formal method in practice. In *13th International Conference on Formal Engineering Methods (ICFEM 2011)*, 2011.

[HBGL98] Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Computer-Aided Verification, 10th Annual Conf.*, pages 109–122, 1998.

[Hen80] Kathryn Heninger. Specifying Software Requirements for Complex Systems: New Techniques and Their Application. *IEEE Transactions on Software Engineering*, 6(1):2–13, January 1980.

[HPSK78] Kathryn L. Heninger, David L. Parnas, John E. Shore, and John W. Kallander. Software requirements for the A-7e aircraft. Technical Report 3876, Naval Research Lab., Washington, D.C., 1978.

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

[LHHR94] Nancy G. Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon D. Reese. Requirements specification for process-control systems. *IEEE Trans. Softw. Eng.*, 20:684–707, September 1994.

[wik] Event-B Wiki. http://wiki.event-b.org/.