Newcastle
University

COMPUTING SCIENCE

Towards a Formalism-Based Toolkit for Automotive Applications

Rainer Gmehlich, Katrin Grau, Michael Jackson, Cliff Jones, Felix Loesch and Manuel Mazzara

TECHNICAL REPORT SERIES

No. CS-TR-1317 March 2012

TECHNICAL REPORT SERIES

No. CS-TR-1317 March, 2012

Towards a Formalism-Based Toolkit for Automotive Applications

R. Gmehlich, K. Grau, M. Jackson, C. Jones, F. Loesch and M. Mazzara

Abstract

The success of a number of projects has been shown to be significantly improved by the use of formalism, both conceputal (methods) and software (tools). However, most of the approaches described in the literature so far leave an open issue: to what extent can the development process be built around strict formal notations from the very beginning. The majority of approaches demonstrate a low level of flexibility by attempting to use a single notation to express all of the different aspects encountered in software development. Often, these approaches leave a number of scalability issues open. We prefer a more eclectic approach. In our experience, the use of a formalism-based toolkit with adequate notations for each development phase is a viable solution. Following this principle, any specific notation is used only where and when it is really suitable and not necessarily over the entire software lifecycle. The approach explored in this article is perhaps slowly emerging in practice — we hope to accelerate its adoption. However, the major challenge is still finding the best way to instantiate it for each specific application scenario. In this work, we describe a development process and method for automotive applications which consists of five phases. The process recognises the need for having adequate (and tailored) notations (Problem Frames, Requirements State Machine Language, and Event-B) for each development phase as well as direct traceability between the documents produced during each phase. This allows for a step-wise verification/validation of the system under development. The ideas for the formal development method have evolved over two significant case studies carried out in the DEPLOY project.

Printed and published by Newcastle University,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

GMEHLICH, R., GRAU, K., JACKSON, M., JONES, C., LOESCH, F., MAZZARA, M.

Towards a Formalism-Based Toolkit for Automotive Applications
[By] R. Gmehlich, K. Grau, M. Jackson, C. Jones, F. Loesch, M. Mazzara
Newcastle upon Tyne: Newcastle University: Computing Science, 2012.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1317)

Added entries

NEWCASTLE UNIVERSITY
Computing Science. Technical Report Series. CS-TR-1317

Abstract

The success of a number of projects has been shown to be significantly improved by the use of formalism, both conceptual (methods) and software (tools). However, most of the approaches described in the literature so far leave an open issue: to what extent can the development process be built around strict formal notations from the very beginning. The majority of approaches demonstrate a low level of flexibility by attempting to use a single notation to express all of the different aspects encountered in software development. Often, these approaches leave a number of scalability issues open. We prefer a more eclectic approach. In our experience, the use of a formalism-based toolkit with adequate notations for each development phase is a viable solution. Following this principle, any specific notation is used only where and when it is really suitable and not necessarily over the entire software lifecycle. The approach explored in this article is perhaps slowly emerging in practice — we hope to accelerate its adoption. However, the major challenge is still finding the best way to instantiate it for each specific application scenario. In this work, we describe a development process and method for automotive applications which consists of five phases. The process recognises the need for having adequate (and tailored) notations (Problem Frames, Requirements State Machine Language, and Event-B) for each development phase as well as direct traceability between the documents produced during each phase. This allows for a step-wise verification/validation of the system under development. The ideas for the formal development method have evolved over two significant case studies carried out in the DEPLOY project.

About the authors

Rainer Gmehlich received the diploma degree in computer science from the University of Karlsruhe, Germany in 1994 and the doctoral degree in computer science also from the University of Karlsruhe, Germany in 2000. Since 1998 he has been working for the Corporate Research and Advance Engineering Center of Robert Bosch GmbH in Schwieberdingen, Germany as a senior scientist. His area of research is software and systems engineering. Especially verification and validation methods for embedded automotive systems.

Katrin Grau received the diploma degree in mathematics from University of Stuttgart, Germany, in 2008. She has been working for Robert Bosch GmbH since September 2008. Until March 2012, she worked in Corporate Research and Advance Engineering in Schwieberdingen in the field of software and systems engineering with the main focus on formal methods.

Michael Jackson is a visiting professor to the School of Computing Science . Since 1990 he has worked as an independent consultant and researcher in software development method, holding visiting posts at several universities and participating DIRC and in several other research projects. Recent work has focused on the analysis and structure of software development problems, using an approach based on the idea of problem frames. He has described his work in four books: Principles of Program Design (1974); System Development (1983); Software Requirements & Specifications (1995); and Problem Frames (2001).

Cliff B. Jones is currently Professor of Computing Science at Newcastle University. As well as his academic career, Cliff has spent over 20 years in industry. His 15 years in IBM saw among other things the creation –with colleagues in Vienna– of VDM which is one of the better known “formal methods”. Under Tony Hoare, Cliff wrote his doctoral thesis in two years. From Oxford, he moved directly to a chair at Manchester University where he built a world-class Formal Methods group which –among other projects– was the academic lead in the largest Software Engineering project funded by the Alvey programme (IPSE 2.5 created the “mural” (Formal Method) Support Systems theorem proving assistant). He is now applying research on formal methods to wider issues of dependability. Until 2007 his major research involvement was the five university IRC on “Dependability of Computer-Based Systems” of which he was overall Project Director. He is also PI on an EPSRC-funded project “AI4FM” and coordinates the “Methodology” strand of the EU-funded DEPLOY project. He also leads the ICT research in the ITRC Program Grant. Cliff is a Fellow of the Royal Academy of Engineering (FREng), ACM, BCS, and IET. He has been a member of IFIP Working Group 2.3 (Programming Methodology) since 1973 (and was Chair from 1987-96).

Felix Loesch received the MS degree in computer science from the Georgia Institute of Technology, Atlanta, in August 2004 and the doctoral degree in computer science from University of Stuttgart, Germany, in 2008. Since April 2005, he has been working for the Corporate Research and Advance Engineering Center of Robert Bosch GmbH in Schwieberdingen, first as a doctoral researcher, then as a post-doctoral researcher. His area of research is software engineering with an emphasis on formal methods for the development of safety-critical embedded automotive systems.

Manuel Mazzara achieved his Masters in 2002 and his Ph.D in 2006 at the University of Bologna. His thesis was based on Formal Methods for Web Services Composition. During 2000 he was a Technical Assistant at Computer Science Laboratories (Bologna, Italy). In 2003 he worked as Software Engineer at Microsoft (Redmond, USA). In 2004 and 2005 he worked as a freelance consultant and teacher in Italy. In 2006 he was an assistant professor at the University of Bolzano (Italy) and in 2007 a researcher and project manager at the Technical University of Vienna (Austria). Currently he is a Research Associate at the Newcastle University (UK) working on the DEPLOY project.

Suggested keywords

REQUIREMENTS ENGINEERING
FORMAL METHODS
EVENT-B
PROBLEM FRAMES
RSML

Towards a Formalism-Based Toolkit for Automotive Applications

Rainer Gmehlich, Katrin Grau, Michael Jackson, Cliff Jones, Felix Loesch, Manuel Mazzara

Abstract—The success of a number of projects has been shown to be significantly improved by the use of formalism, both conceptual (methods) and software (tools). However, most of the approaches described in the literature so far leave an open issue: to what extent can the development process be built around strict formal notations from the very beginning. The majority of approaches demonstrate a low level of flexibility by attempting to use a single notation to express all of the different aspects encountered in software development. Often, these approaches leave a number of scalability issues open. We prefer a more eclectic approach. In our experience, the use of a formalism-based toolkit with adequate notations for each development phase is a viable solution. Following this principle, any specific notation is used only where and when it is really suitable and not necessarily over the entire software lifecycle. The approach explored in this article is perhaps slowly emerging in practice — we hope to accelerate its adoption. However, the major challenge is still finding the best way to instantiate it for each specific application scenario. In this work, we describe a development process and method for automotive applications which consists of five phases. The process recognises the need for having adequate (and tailored) notations (Problem Frames, Requirements State Machine Language, and Event-B) for each development phase as well as direct traceability between the documents produced during each phase. This allows for a step-wise verification/validation of the system under development. The ideas for the formal development method have evolved over two significant case studies carried out in the DEPLOY project.

Index Terms—Requirements Engineering, Formal Methods, Event-B, Problem Frames, RSML

"Aut inveniam viam aut faciam"

1 Introduction

Requirements Engineering is a complex discipline where “human skills” cannot be relegated to a secondary role. The process of elicitation, analysis and specification relies heavily on human ability to collect, analyse, communicate and interpret information in a context. That is why aiming at a completely mechanizable procedure leading from requirements to code in one go is not just an ambitious goal, it is not reasonable and it is simply not doable. That is also why it is not the goal of this paper. However, although in the earliest phases of a project dependable communication between parties may still be considered the most important aspect to take care of, the success of the following steps has been proven to be significantly affected by the use of formal tools, both conceptual (methods) and software (applications).

It is generally difficult to produce a final artefact which entirely satisfies customer requirements, but Object Oriented Design [1] and Component Computing [2] are two well known examples of how rigor and discipline can improve the final quality besides any human communication implication. The success of languages like Java or C# has to be interpreted from this perspective, i.e., they are natural target languages to (formally) organize and structure problems from the earliest phases of development. It is worth remembering that such a success is due both to the existence of software and rigorous methodological tools. Semi-formal notations like UML [3] also helped in creating a language that can be understood by both specialists and non specialists, providing different views of the system that can be negotiated between stakeholders with different backgrounds. The power (and not the limitation) of UML is the absence of a formal semantics (although many attempts can be found in literature) and its strong commitment to a way of reasoning and structuring problems which is the one disciplined by object orientation.

However, in Object Oriented Design or other traditional formally structured development methodologies, the first formal object created is still code. Static analysis and testing are very effective ways to identify a number of flaws, but it is still very hard to verify that code satisfies the original customer requirements. With this approach, verification is carried out very late in the development process and, as a consequence, fixing errors has a higher cost. With these premises, verification appears to be impractical due to the large gap between code and requirements. To bridge this gap, several other approches based on formal methods have emerged over the decades. Formal/mathematical notations have existed for a long time and have been used to specify and verify systems. Examples are process algebras (a short history by Jos Baeten in [4]), specification languages like Z (early description in [5]), B [6] and Event-B [7].

- *Cliff Jones, Michael Jackson, and Manuel Mazzara are with the School of Computing Science, Newcastle University, Newcastle upon Tyne, United Kingdom*
- *Rainer Gmehlich, Katrin Grau and Felix Loesch are with Corporate Research, Robert Bosch GmbH, Stuttgart, Germany*

This work has been funded by the EU FP7 DEPLOY Project (Industrial deployment of system engineering methods providing high dependability and productivity). More details at www.deploy-project.eu.

The Vienna Development Method (VDM) is one of the earliest attempts to establish a formal method for the development of computer systems [8], [9], [10]. A survey of these (and others) formalisms can be found in [11] while a discussion on the methodological issues of a number of formal methods is presented in [12], [13], [14].

All these approaches (and others described in literature) still leave an open issue, i.e., they are built around strict formal notations which affect the development process from the very beginning. These approaches demonstrate a low level of flexibility. It is indeed not reasonable to expect that a single notation can express all the different aspects encountered during the software development cycle. Therefore, these methods seem to work only for small problems, leaving a number of scalability issues open.

In this paper, following the experience accumulated during the FP7 DEPLOY project [15], we reject the notion that any specific notation can solve all the software development problems. In our research, we instead made progress by moving to a very different position where having a toolkit of notations or formalisms, and using those we felt could assist us best in each phase (from requirements to code), shown to be the most viable solution. The remainder of this paper is structured as follows: Section 2 describes the context in which this research has been done and the related issues; Section 3 gives a short introduction to the different methods on which the approach proposed is based; Section 4 contains a description of the application area, i.e., a description of the important characteristics of automotive systems; Section 5 describes phases and relative decisions that have been made; Section 6 describes the formal development method in detail; related works and conclusions complete the paper in Section 7 and Section 8.

2 The DEPLOY Project

After having experimented over two significant case studies in Bosch Research (Cruise Control and Start/Stop System) for the duration of the DEPLOY project [15], the ideas and thoughts which have evolved are presented in this paper. DEPLOY was an ambitious project addressing diverse major industrial areas: automotive, train transportation, business and aerospace software. Putting all of them under the same umbrella was a difficult (if not impossible) task for both instrinsic and extrinsic reasons. We recognized from the very beginning that each deployment scenario was different and needed (at least partially) different approaches, concepts and tools. Industries were different, development processes differed, internal organizations varied and, to some extent, business models were different as well as politics. Most importantly, target applications and in-house engineering tools/standards showed little similarity. Therefore, integration needs were different. Ample documentation and a strong publication track have been created during the project, the interested reader can refer to the project website [15] to find papers describing the train transportation, business and aerospace applications of formal methods. In this paper we focus on the automotive scenario and the collaboration with Bosch.

Over the duration of the DEPLOY project, the collaboration between Newcastle University and Bosch Research has been close and brought a broader understanding of the problems to both sides. Before the beginning of the project, Bosch project members had no previous experience with Problem Frames [16], Requirements State Machine Language (RSML) [17] and Event-B [7], neither at the theoretical nor at the application level, which made this deployment scenario a particularly interesting one. More specifically, the absence of any previous experience with the formalisms and tools on which the project was focusing immediately brought a major requirement to our attention, i.e., the formalisms applied had to be understandable by engineers. The actual meaning of this statement can only be instantiated empirically and working in the relevant field and it cannot be in any way the result of intellectual or academic speculation. This was one of the major barriers to be overcome in achieving the project objectives. Hard work, close collaboration, will to communicate and common sense offered a solution to the problem. Thus we avoided falling into "Maslow's hammer thinking trap" [18].[1]

Traceability is another major issue and the use of a single notation, even when it comes together with a rigorous refinement methodology like Event-B, does not seem to offer a complete solution by itself. Unfortunately, software development does not offer a panacea to cover every phase from requirements to code. The use of a formalism-based toolkit with a varied portfolio showed to be a viable solution instead. Following this principle, a specific notation is used only where and when it is really suitable and not necessarily over the entire lifecycle. In this work, Jackson's Problem Frames Approach (PFA) [16] and the Requirements State Machine Language (RSML) [17] have been applied before the application of Event-B in order to progress incrementally from an informal to a formal description of the system. This method generates several documents during the development process (e.g., requirements, specification, formal model, code), but traceability is not lost since the formality of the approach permits establishment of links between requirements and subsequent documents. Errors can be isolated early thanks to step-wise verification/validation of results (i.e., the specification is validated against the requirements; the formal model is validated against the specification; and, finally, the code can be automatically generated by the Event-B refinement chain).

3 Background

This section gives a short introduction to the different methods on which the approach proposed is based. It is

[1] Maslow, in his book, noted that "It is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail".

not intended to be exhaustive, but it provides the reader with relevant pointers for further investigation.

3.1 Problem Frames

PFA [16] focuses on systems in which the computer interacts with the physical world to achieve a required behaviour there. Stakeholders in the system –users, sponsors, operators, regulators and others– want this behaviour to satisfy certain properties. These desired properties may be expressed in various forms and with various degrees of exactness: for an industrial press a vital desire is the operator's safety; for an electronic purse system it is conservation of money in any transaction between two purses even if the transaction fails or is aborted. The requirements engineer must understand these desires and design a feasible joint behaviour of the computer and the world that will satisfy them.

In PFA this task is understood in terms of three principal parts. First, the machine: this is the computer executing the software that will eventually be developed. Second, the problem world, seen as an assemblage of distinct domains interacting with each other and with the computer. Third, the system requirement, initially seen as the set of desired properties of the system behaviour. The system is represented in a problem diagram. The diagram shows the computer, the problem domains, and the interfaces of shared phenomea at which they interact; the requirement is represented by a distinguished block linked to the problem domains to whose phenomena it refers. The requirements engineering task is to specify the given properties W of the problem world domains, the behaviour M of the computer, and the required joint behaviour R resulting from their interactions. The entailment $M, W \models R$ must hold, and the behaviour R must exhibit the properties desired by the stakeholders.

For a realistic system M, W, R and the desired properties will be complex. The problem is therefore decomposed into subproblems, each represented by a problem diagram. A subproblem is a closed independent projection of the original problem, ignoring all interactions with other subproblems. Recombination is deferred until each subproblem is well enough understood in isolation. A further task is then to design the temporal composition of the subproblem behaviours and to resolve any interference and conflict arising in their resulting interactions.

This specification of system behaviour does not map directly either to an Event-B specification or to a software architecture: refactoring is a further step in the path to implementation. It is a fundamental claim of PFA that the cost of this refactoring is amply compensated by the clarity that can be achieved in the requirements engineering task itself and the consequent improvement in system quality and dependability.

3.2 Requirements State Machine Language (RSML)

The Requirements State Machine Language (RSML) [17] is a formal black-box specification language invented by Nancy Leveson and has been widely applied in the avionic industry for the specification of complex state-based embedded systems like the transition collision avoidance system (TCAS II). RSML was developed in order to have precise description of the functional behaviour of state-based systems which is formal enough to reason about general aspects like completeness and consistency of state machines [19] but still easy enough to be understandable by engineers. The language itself consists of concepts for structuring a large specification, i.e., the language supports modules with defined interfaces as well as formal concepts for describing state machines based on statecharts [20] extending state diagrams with state hierarchies and broadcast communications.

An important concept introduced by RSML is the concept of AND/OR tables which are used to describe conditions for state transitions and conditions for the assignment of variables. Table 1 shows an example for 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 phases 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”.

TABLE 1
Example of an AND/OR table

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

3.3 Event-B

The Event-B Modelling Language [7] was developed by J.-R. Abrial and his team at ETHZ as a specialization of the B-Method [6] and it is used to describe formally systems and reason mathematically about their properties. Event-B belongs to the category of model-based formalisms. There is a long tradition of methods in this category which appeared before Event-B, e.g., VDM [8], [9], [10], Z [5] and the B-method itself [6]. The mathematics underlying all these formalisms is set theory and first order logic. The approach consists of modeling the system's state in terms of sets and functions, and modelling state transformation using operations (or events in the case of Event-B). Predicates are used to express invariant conditions on the state. In Z, the emphasis is on formal specification, whilst the B-method emphasizes the ”method” itself. Both B and Event-B focus on the application of stepwise refinement (reification in VDM), that is, the verifiable transformation of a high-level formal specification into an executable program. Model-based formalisms are mature, and they have extensive tool support, for example, Overture for VDM [21] and Rodin for Event-B [22].

In more detail, an Event-B model consists of Machines (modelling the dynamic behaviour: variables, invariants, events) and Contexts (static information: constant identifiers, values and their properties). Variables store the machine's state while invariants constrain types and state logical properties over the variables. Events define the actual behaviour of the system (state transitions) and may include a guard, defining the states from which the transition is allowed to occur. Events also include a set of actions defining how variables should be updated, i.e., state transformation. During the refinement process, new events are introduced at every refinement step adding further details to the model and allowing the developer to get closer and closer to the concrete implementation. At each refinement step, existing events can be kept as they are or they can be refined into another event (for example changing state variables). They can also be split. These refinement steps come together with specific proof obligations (which are essentially predicates on states) and tool support providing help to discharge them. This means that, in Event-B, refinement steps are verified with respect to their proof obligations in such a way that transitivity of refinements guarantees the final system description being a refinement of the initial one. Further and deeper explanation of the method, paired with concrete examples of modelling coming from real case studies, is presented in Section 6.3.

Event-B offers a framework which claims to be flexible enough to support system development through stepwise refinement. In this paper we are partly challenging this claim. It is worth noting that we are certainly not undervaluing the Event-B contribution to software engineering, we want only to point out that formal methods, in general, does not offer a panacea to cover every phase of the software lifecycle, from requirements to code (and beyond). The remainder of this paper shows some of the Event-B limitations.

4 Application Area

Automotive applications are embedded systems. This section outlines some major technical and non-technical characteristics necessary to understand the bigger picture in which our results emerged. Technical characteristics include:

- Most automotive applications are designed as embedded systems to implement one or a few dedicated functions. The coordination and prioritization of requests is an important task of embedded systems within the automotive environment. As a consequence of this the concept of finite state machines is often used.
- On the other hand, automotive systems control/influence the behaviour of a car which is intrinsically continuous. The core functionality of many automotive applications is a closed loop controller.
- Reacting as quick as possible to driver requests or changing environmental conditions is another important task of most applications implemented in the automotive domain. Thus, an appropriate way to model time is also an important aspect when applying formal methods to sytems typical of the automotive domain.
- Automotive applications are cyclically executed, with a predefined static priority of the different tasks. The basis execution scheme is gathering input, calculating and producing the output.

Non-technical characteristics include:

- Typical software applications in the automotive domain may never behave in a way that may endager the driver, the vehicle, or other vehicles nearby. For that reason, safety requirements [23] are defined and the fullfillment of these safety requirements has to be justified.
- The current development process is best described as a tailored version of the V-Model. Therefore and for regulatory reasons (fullfilling of standards [23]) some basic documents have to be produced during the development.

5 Our Approach

Our approach for a formal development process for automotive applications [2] evolved after having experimented over two significant case studies in the DEPLOY project [15]. During these case studies we found that the gap between informal descriptions (i.e., requirements in natural language and formal descriptions of the system in Event-B) is very large. Verifying that the formal descriptions are consistent with the informal descriptions turned out to be a very difficult task because of the inevitable vagueness of informal descriptions and missing traceability links between the informal and formal descriptions.

In order to bridge this gap and to progress incrementally from an informal to a formal description of the system, our approach consists of five phases (requirements, specification, formal modelling, formal verification, and code generation) in which carefully selected and appropriate (formal) notations are used. A specific formalism is used only where and when it is really suitable and not over the complete development cycle. The outcome of each phase during the development process is an adequate document which describes the results of each phase and which can be used to communicate easily with other stakeholders like managers, customers, and other developers during the development process. Traceability between the documents produced in each phase is not lost since the formality of our approach permits the establishment of links between requirements and subsequent documents. Step-wise verification/validation of results allows an early detection and correction of errors

[2] Automotive applications contain discrete and continuous parts (closed loop controllers). In the case studies we concentrated on the discrete part of the system. We decided not to model the continuous part and only used an abstract notion of time.

during the development process (i.e., the specification is validated against the requirements; the formal model is validated against the specification; and, finally, the code can be automatically generated by the Event-B refinement chain).

Figure 1 graphically depicts our development process. Table 2 summarizes the five phases of our approach, the applied (formal) notations, the main activities as well as the outcomes of each phase.

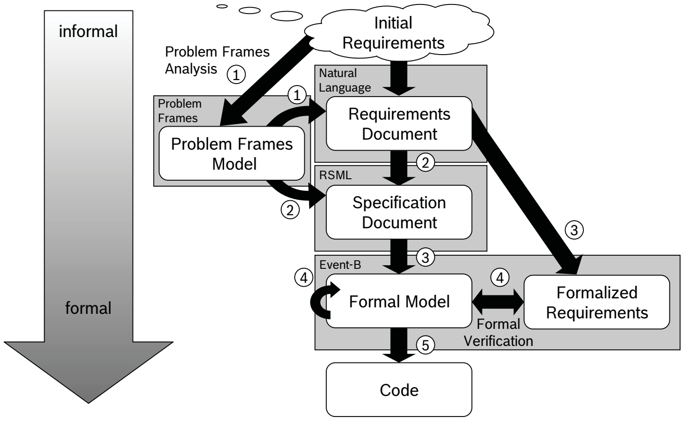

Fig. 1. Formal development process

The starting point for our process is an abstract idea of the system and/or some vague initial requirements (see step 1 in Figure 1). In order to produce a requirements document for further development these vague initial requirements have to be analyzed and concrete requirements have to be developed. For this analysis and development PFA [16] is applied. The main outcome of this phase is a requirements document in natural language which also contains assumptions on the system to be developed. We deliberately chose to describe requirements in natural language in order to make it as easy as possible to discuss the requirements with different stakeholders in the development process (e.g., customers and managers). Section 6.1 describes this phase.

The next phase in the process is the specification phase in which the desired functional behaviour of the system and the architecture is described in a precise way using RSML [17] (see step 2 in Figure 1). Inputs to this phase are the requirements document as well as the Problem Frames model. The outcome of the specification phase is a specification document which contains a description of the architecture of the solution as well as a detailed description of the functional behaviour of each component. A detailed description of this phase is presented in Section 6.2.

After the specification phase the formal modelling phase follows (see step 3 in Figure 1). During this phase the specification is translated into a formal model written in a formal language (e.g., Event-B). The main activities in this phase are the formalization of the functional behaviour, i.e., how the system is achieving it, as well as the formalization of requirements, i.e., what the system should do. Section 6.3 contains a detailed description of the formal modelling phase.

The next phase in our approach is the formal verification phase (see step 4 in Figure 1). In this phase the refinements of the formal model as well as the formalized requirements are verified on the model using formal verification techniques such as theorem proving and model checking. The outcome of the verification phase is a verified formal model with regard to the formalized requirements. Details of this phase are described in Section 6.4. For a better understanding the formal verification and the formal modelling phase are described as two separate sequential tasks in this paper. However, in practice the formal verification phase goes hand in hand with the formal modelling phase. Good integeration of these two tasks helps to find errors as early as possible and is therefore an important aspect of the development.

Having obtained a verified formal model of the system, the last phase in our approach is code generation (see step 5 in Figure 1). During this phase the verified formal model is translated into a programming language which can then be compiled and run on embedded devices. The main outcome of this phase is thus the code which serves as an input to the compiler. A description of this phase can be found in Section 6.5.

6 Formal Development Method

This section presents a detailed description of the five phases of our approach. For each phase of our approach the specific requirements and constraints for chosing an adequate (formal) notation are discussed before we present the arguments for how the chosen notation fulfils the requirements of each phase. The description of each phase is illustrated using an example from our second case study in the DEPLOY project [15].

The system we analyzed in our second case study was a Start/Stop System which automatically stops the engine, e.g., at traffic lights, to save fuel (see also [26]). The engine will be automatically restarted when the driver wants to move the car again. The system is an embedded real time system. However, contrary to other software functions in the automotive domain, the Start/Stop System only consists of discrete functionality containing a complicated state machine for determining when to stop and when to start the engine.

6.1 Requirements Development

Constraints The starting point for a new product or a new feature for a product is usually an abstract idea, some vague initial requirements. To produce a requirements document which can be used for further development these initial requirements have to be refined. We used Problem Frames [16] (see Section 3.1) for the problem analysis and the central idea of this first part of the development process is to concentrate on the problem that has to be solved, not on possible solutions.

TABLE 2
Overview of development phases, formalisms, main activities and outcomes

Phase	Formalism/Method	Main Activities	Outcomes
Requirements Analysis / Development	Problem Frames Approach (PFA) [16]	• Problem decomposition • Development of requirements	• Requirements Document
Specification	Requirements State Machine Language (RSML) [17]	• Description of desired solution structure • Description of desired functional bevahiour	• Specification Document
Formal Modelling	Event-B [7]	• Formalization of functional behaviour • Formalization of requirements	• Formal Model
Formal Verification	Theorem Provers (Atelier-B)/ Model Checking (ProB) [24]	• Formal proof of desired properties • Formal proof of consistency of the model	• Proven Formal Model
Code Generation	EB2C [25]	• Generation of executable code	• Code

During this analysis, having some structure helps to find a systematic way to analyse the problem. On the other hand, a completetly formal notation would restrict the freedom needed during this early phase.
The outcome of the requirements phase of the development process is a requirements document, not a Problem Frames model. Although problem frames are easy to read some basic knowledge is still needed to understand the diagrams. The requirements document is written only in natural language to be as easy to read as possible. This document is the basis for discussions with all stakeholders, including customers and engineers. In the requirements document no information about the solution structure of the system is used. The system is described as a black box as this starting point for further development should not restrict possible solutions for the system being analyzed.

Description of method In PFA we start with an abstract diagram, an overview of the world of which the system to be built is a part. An abstract requirement describes the effect the system has on the world. Note that the requirement does not refer to the system itself (which would be a restriciton of the solution). After this abstract examination more concrete subproblems are considered. In these subproblems one aspect of the overall problem is developed in detail with requirements that refer only to this specific aspect. The problem of how to recombine these different aspects is postponed and adressed after the development of all subproblems. In every subproblem there is at least one requirement. This requirement refers only to the subproblem. After the development of every subproblem in isolation the recombination must address the priorization of the single subproblems.

An interesting question is where to find relevant information in the Problem Frames model which has to be included in the requirements document. Of course the requirements in the model are themselves part of the requirements document. But this is not enough to build a solid basis for further development. First of all the assumptions the model is based on have to be included. The domain descriptions illustrating the domains surrounding the system are also a very important part of the requirements document as they state necessary information for the system. In the requirements document the combination problem has to be solved. In this document only a black-box description is allowed and therefore the complete system has to be addressed. All these different information sources together build the basis for the requirements document.

Example 1: The Start/Stop System is not allowed to prevent the driver from moving the car whenever he or she wishes to do so. Therefore there is a requirement in the natural language requirements document stating *The Start/Stop System is not allowed to change the engine status from running to off if the driver wants to move the car*. There is the obvious question of how the Start/Stop System is supposed to judge when the driver wishes to move the car. The Start/Stop System does not have access to the wishes of the driver, but it has access to the steering wheel, the clutch pedal and the gearbox. The wish of the driver to move a standing car is modelled as follows:

1) If the engine is running and the driver does not want to move the car, then the steering wheel is not being

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 being used, the clutch is pressed or the gearbox is not in neutral.

In the requirements document not only the requirement itself has to be included but also this additional information.

Example 2: The same aspect of the Start/Stop System is treated in the Problem Frames subproblem shown in Figure 2. The machine, i.e., the box with the double vertical stripe, is called SSE_Driver_Needs_HMI, referring to the fact that this subproblem concentrates of the needs of the driver, which are deduced by the HMI (Human-Machine Interface). To be able to solve the recombination problem the engine is not part of the subproblem. Instead a designed domain called SSE_Driver_Needs_HMI_Model is used and therefore the requirement does not refer to the engine (as in the requirements document) but to this designed domain, i.e., the box with the single vertical stripe. The designed domain has a phenomenon named HMI_Stop_Ena (there is another phenomenon called HMI_Strt_Req, which is not relevant for this example but will be used in Example 4). The phenomenon stores the information of this subproblem related to the stopping of the engine, i.e., of whether this subproblem enables the Start/Stop System to stop the car or not. For more details please see [26]. In the domain Driver a model of the driver is defined, which states the connection mentioned in example 1 in 1.) and 2.) between the wishes of the driver and the steering wheel, the clutch and the gearbox. The steering wheel can be used or not used, the clutch pedal can be pressed or released, the gearbox can be in neutral or not in neutral.

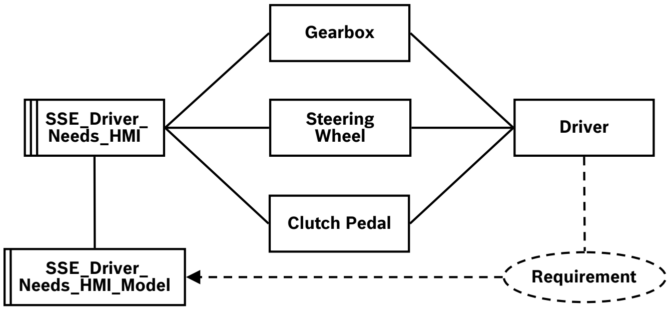

Fig. 2. Problem Frames subproblem

Please note that although the two examples above were presented the other way around PFA precedes the task of documenting natural language requirements in the requirements document.

Summary The use of Problem Frames helps to concentrate on the problem to solve and develop a better understanding of how the system to build is supposed to affect the surrounding world. The additional requirements document in natural language is the basis for discussions with all stakeholders.

6.2 Specification

Constraints After having produced a requirements document containing the requirements and assumptions on the system, the next logical step in the development process is to develop a detailed specification which should include a precise description of the functional behaviour of the system as well as a description of the general architecture. Ideally, the specification method should provide adequate means for describing the functional behaviour of the system, i.e., what the system will do and how this can be achieved, as well as means for describing the architecture of the solution. In our case this means that the specification method should provide means for describing state-based functional behaviour as well as means for describing individual components that communicate via shared variables. Another important aspect for our type of systems is the ability to specify abstract time and the execution order of components (see also Section 4). On the one hand the specification method should be formal enough to reason about the general structure of the solution, e.g., reasoning about consistency of interfaces between components and formal enough to reason about general aspects of the behaviour of state-based systems, e.g., deadlocks in a state machine and completeness of transitions with regard to all possible inputs.

On the other hand, the specification method should still be understandable by engineers who are not familiar with formal notations like Event-B.

RSML [17] is ideally suited for our task of specifying the functional behaviour of state-based automotive systems because it is easy enough to be understandable for engineers but still formal enough to reason about general aspects of state-based systems and fulfills the other constraints described above. The outcome of the specification phase is a specification document written in RSML which is then used as input for the formal modelling phase.

Description of method For the specification of the system we start with the requirements document and the Problem Frames model produced during the requirements development phase. These documents contain requirements and assumptions about the system to be developed but do not contain a precise description of the desired functional behaviour of the system. Thus, the task for the specification phase is to specify the desired functional behaviour such that it fulfills the set of requirements described in the requirements document. In order to structure the solution, the first step during specification is to think about the general architecture of the system. As with the decomposition of the problem in the requirements development phase, the solution is decomposed into components that describe specific aspects. For each component its interface is precisely defined using typed input and output variables. Components communicate with other components via shared variables, e.g., the output variables of component A serve as input variables to component B and vice versa. If necessary, a component may also contain internal variables to store values derived from input variables. Figure 3 shows an exemplary static structure of an embedded controller consisting of two components A and B and their interfaces.

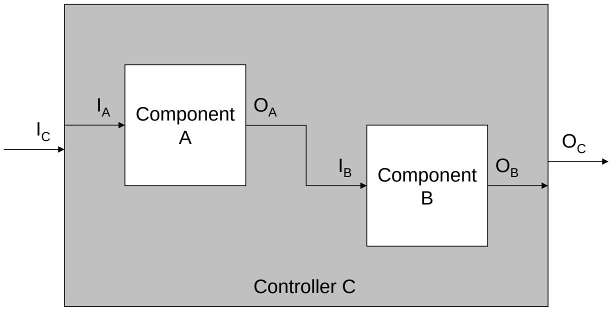

Fig. 3. RSML - Static structure with components and interfaces

The desired functional behaviour of a component is specified using two concepts. The first concept –called assignment specification– is to relate output variables directly with conditions on input variables using AND/OR tables (see Section 3.2). The following example taken from the Start/Stop System case study illustrates this concept.

Example 3: The value of the boolean output variable HMI_Stop_Ena is dependent on specific conditions on the input variables Clutch_Pedal, Steering_Wheel and Gearbox. These conditions are specified by the assignment specification shown in Figure 4.

Assignment: **HMI_Stop_Ena**

Condition: d

Clutch_Pedal = PRESSED	**T**	●	●
Steering_Wheel = USED	●	**T**	●
Gearbox $\neq$ NEUTRAL	●	●	**T**

Action(s): HMI_Stop_Ena := FALSE

Condition: $\neg d$

Action(s): HMI_Stop_Ena := TRUE

Fig. 4. RSML - Assignment specification for HMI_Stop_Ena

The second concept is to define a state machine whose transitions are guarded with conditions on the input variables. The state machine serves as an abstraction on complicated conditions on input variables and is described using graphical state diagrams showing the states and transitions but not the conditions on the transitions. AND/OR tables are used again to specify the transition conditions.

Another important task of the specification is to define the execution order of components and assignments to output variables in these components as well as the synchronization of parallel state machines. This execution order is later required for the formal model (see Section 6.3). In RSML [17] the execution order of transitions in state machines can be specified using so called triggering events. Triggering events can be produced by the environment at periodic intervals or as actions on transitions in other state machines. The inventors of RSML themselves state that triggering events are a powerful mechanism for specifying execution order but they may become very difficult to handle if they are applied to large complex systems [27]. Therefore, we decided not to use triggering events. Instead we use a simple language for expressing execution order of components and assignments in components. The language consists of the following concepts:

- **Names** Each assignment of an output, internal or state variable of a component is assigned a unique name
- **Parallel Execution Operator (||)**: the assignments left and right of the operator can be executed in arbitary order
- **Sequential Execution Operator (.)**: the assignment to the left of the operator must be executed before the assignment to the right of the operator

Using this simple notation it is possible to define *local execution orders* that specify the order of execution of assignments within each component. Similarly, a *global execution order* specifies the order of execution for the components of the specification.

Example 4: The following shows a local execution order for two assignments within the HMI component of the Start/Stop System: [HMI_Strt_Req||HMI_Stop_Ena]. The local execution order specifies that the assignments to HMI_Strt_Req and HMI_Stop_Ena can be executed in parallel because the variables are independent.

Summary Applying RSML for the specification of automotive applications showed very promising results. We were able to express the complete functional behaviour of the Start/Stop System in RSML. The language was formal enough to describe precisely the functional behaviour yet still readable by engineers which was very important in order to permit domain engineers to validate the specification against the requirements document without needing special training in formal methods. However, we did not have tool support for RSML which was both an advantage and a disadvantage. The advantage of not having a tool was that it allowed us to have more freedom in the structure of the specification. The disadvantage was that we did not have the possibility of automatically checking the specification for consistency.

6.3 Formal Modelling

Constraints There are two purposes of the formal modelling phase: One is to translate the specification into a formal model, i.e., a mathematically precise description of the functional behaviour of the system. The second purpose is to formalize the requirements in order to make them amenable for formal verification.
Such a formal model should provide the basis for formal verification (see Section 6.4). Thus, the formal language used for formal modelling must be formal enough to describe precisely the functional behaviour specified in the specification and to formalize the requirements we would like to prove on the formal model. In order to make these informal descriptions accessible to formal verification they have to be stated formally as well. Furthermore, the formal language must suit the application area (i.e., description of state-based systems) and provide means for structuring the formal model. In addition to that, the formal language must be concrete enough to generate code from the formal model.
Event-B [7] fulfills most of the constraints mentioned above. It is suited for the description of state-based systems since it is based on action transition systems and it is formal enough to describe precisely the functional behaviour as well as a large number of the properties we would like to prove about the system as invariants. Furthermore, it provides a refinement mechanism which allows us to start with an abstract formal model which can later be refined to a concrete model which provides the basis for code generation.

Description of method Formal modelling in the language Event-B typically starts with a very abstract model which is refined step-by-step until the system and the environment has been completely modelled. For the Start/Stop System the formal modelling starts with a very abstract model containing only the output of the Start/Stop System. This model is then refined step-by-step. In each refinement step additional components described in the specification document are added to the formal model. Typed input and output variables of components described in the specification are modelled as *variables* in the Event-B model. The types of these variables are specified using *type invariants*. Each assignment specification and each transition of a state machine described in the specification is modelled by *events* in Event-B, i.e., the conditions for the assignment are described as *guards* of the event whereas the assignment itself is described using an *action* of the event. It is important to note that the Event-B model also contains events for the system environment which models changes of system inputs. For example, the Event-B model for the Start/Stop System contains unguarded events modelling changes of input variables such as Clutch_Pedal, Gearbox, and Steering_Wheel.

Example 5: Figure 5 shows how the assignment specification for the output variable HMI_Stop_Ena in RSML (shown in Example 4) is translated into Event-B syntax.

```
variables
  HMI_Stop_Ena Clutch_Pedal
  Gearbox Steering_Wheel
invariants
  @inv1 HMI_Stop_Ena ∈ BOOL
  @inv2 Clutch_Pedal ∈ T_Clutch_Pedal
  @inv3 Steering_Wheel ∈ T_Steering_Wheel
  @inv4 Gearbox ∈ T_Gearbox
events
  event Set_HMI_Stop_Ena_FALSE
    when
      @grd1 Clutch_Pedal = PRESSED ∨
            Steering_Wheel = USED ∨
            Gearbox ≠ NEUTRAL
    then
      @act1 HMI_Stop_Ena := FALSE
  event Set_HMI_Stop_Ena_TRUE
    when
      @grd1 Clutch_Pedal ≠ PRESSED
      @grd2 Steering_Wheel ≠ USED
      @grd3 Gearbox = NEUTRAL
    then
      @act1 HMI_Stop_Ena := TRUE
  end
end
```

Fig. 5. Event-B model for HMI_Stop_Ena

As you can see in Figure 5 the output and input variables are modelled as Event-B variables. Their types are specified by Event-B invariants. The assignment specification for the output variable HMI_Stop_Ena is modelled as two Event-B events depending whether HMI_Stop_Ena is set to TRUE or FALSE.

The execution semantics for events in Event-B does not define an order in which the events are executed (see also Section 4). An *event* is enabled to be executed if its guards evaluate to TRUE. If more than one event is enabled, one of them is chosen non-deterministically for execution [28]. However, this would allow for an arbitrary order for execution of events which is not desired for embedded systems. Embedded systems often require a cyclic execution of events in which the events modelling the system reaction follow the events modelling the environment, i.e., it is required that for each change in the environment (e.g., change of input variables) the system is reacting by calculating an output. In order to specify a specific order in which events are executed one can add a so-called "program counter" to the formal model, e.g., an integer variable tracking the order. This integer variable is initialized during initialization of the formal model. In each event, this program counter is evaluated in the guard and increased after the execution of the event. Using this program counter it is possible to specify a cyclic execution order for events, i.e., the environment and all events modelling the reaction of the system to changes in the enviroment are executed in a cycle.

Example 6: Figure 6 shows the Event-B model for the output variable HMI_Stop_Ena with the added program counter. As you can see from Figure 6 the program counter is checked in the guard and increased by one in the action. Thus, in our example the event HMI_Set_HMI_Stop_Ena_FALSE is only executed if both guards evaluate to TRUE, i.e., the program counter must equal 10 and the first guard must evaluate to TRUE. It is assumed that there exists an environment event changing the input values Clutch_Pedal, Steering_Wheel and Gearbox which is executed before the events Set_HMI_Stop_Ena_FALSE and Set_HMI_Stop_Ena_TRUE. Furthermore, it is assumed that there exists another event that resets the program counter to its initial value.

```
variables
  HMI_Stop_Ena Clutch_Pedal
  Gearbox Steering_Wheel
  Program_Counter
invariants
  @inv1 HMI_Stop_Ena ∈ BOOL
  @inv2 Clutch_Pedal ∈ T_Clutch_Pedal
  @inv3 Steering_Wheel ∈ T_Steering_Wheel
  @inv4 Gearbox ∈ T_Gearbox
  @inv5 Program_Counter ∈ NAT
events
  event Set_HMI_Stop_Ena_FALSE
    when
      @grd1 Clutch_Pedal = PRESSED ∨
            Steering_Wheel = USED ∨
            Gearbox ≠ NEUTRAL
      @grd2 Program_Counter = 10
    then
      @act1 HMI_Stop_Ena := FALSE
      @act2 Program_Counter := 11
  event Set_HMI_Stop_Ena_TRUE
    when
      @grd1 Clutch_Pedal ≠ PRESSED
      @grd2 Steering_Wheel ≠ USED
      @grd3 Gearbox = NEUTRAL
      @grd4 Program_Counter = 10
    then
      @act1 HMI_Stop_Ena := TRUE
      @act2 Program_Counter := 11
  end
end
```

Fig. 6. Event-B model for HMI_Stop_Ena with program counter

Adding guards that evaluate the program counter and actions, setting the program counter manually for each event, is a cumbersome task. In order to automate this task a FLOW plugin [29] is currently being developed for Rodin which allows a graphical specification of the order of events and a generation of program counter variables for the Event-B model as well as the generation of proof obligations which, when proven, show that the flow specified by the FLOW plugin is feasible in the formal model.

After having produced the requirements document (see Section 6.1), the formalization of the requirements can be started, separately from the specification (see Section 6.2) and the development of the Event-B model of the system itself which was described in the previous paragraphs. The formalized requirements are used as the basis for formal verification of the Event-B model (see Section 6.4).

The translation of the natural language requirements is straightforward. The single sentences are analyzed and mapped to the input and output variables of the system.

The main feature of Event-B with which to state properties for a model is the concept of invariants. These invariants describe predicates that are proven to be always true. Certain safety properties can be easily described as invariants (e.g., if a defined output of the system is generally forbidden). An example from the Start/Stop System is that there should never be the request to start and the request to stop the engine at the same time. This kind of property is naturally suitable for formalization as invariants.
Most of the properties to be proven describe the reaction of the system to certain inputs (see Section 4). Event-B does not have a natural way of treating time. For reactions one has to carefully state a predicate that is always true, as the system always needs time to react to a certain input. **Example 7:** The natural language requirement presented in Example 1 in Section 6.1 is formalized as follows:

$$\begin{array}{lcll}
\text{Engine\_Status} & = & \text{Running} & \wedge \\
\text{DriverWantsToMoveCar} & = & \text{TRUE} & \wedge \\
\text{Program\_Counter} & = & \text{ReactionTime} & \\
\Rightarrow & & & \\
\text{SSE\_Stop\_Order} & = & \text{FALSE} &
\end{array}$$

Fig. 7. Formalized requirement

The part DriverWantsToMoveCar = TRUE has to be further defined according to the driver model described in the example in Section 6.1 and Program_Counter = ReactionTime has to be adjusted to the actual Event-B model. With SSE_Stop_Order the Start/Stop System is able to influence the engine. To prove this invariant in Event-B it is necessary to introduce intermediate invariants that guide the proof (see also Section 6.4). In Figure 6 the main step of the Event-B model concerning this requirement is presented. Referring to this part of the model the following intermediate invariant is proven:

The Program Counter has been adjusted to the model, i.e., Program_Counter = 11 is set. The part DriverWantsToMoveCar = TRUE is replaced by the driver model, i.e., (Steering_Wheel = USED ∨ Clutch_Pedal = PRESSED ∨ Gearbox ≠ NEUTRAL). The only part missing to prove the invariant in Figure 7 is that if HMI_Stop_Ena = FALSE then

$$\begin{array}{lcll}
\mathrm{Engine\_Status} & = & \mathrm{Running} & \wedge \\
(\mathrm{Steering\_Wheel} & = & \mathrm{USED} & \vee \\
\mathrm{Clutch\_Pedal} & = & \mathrm{PRESSED} & \vee \\
\mathrm{Gearbox} & \neq & \mathrm{NEUTRAL}) & \wedge \\
\mathrm{Program\_Counter} & = & 11 & \\
\Rightarrow & & & \\
\mathrm{HMI\_Stop\_Ena} & = & \mathrm{FALSE} &
\end{array}$$

Fig. 8. Intermediate invariant in Event-B

SSE_Stop_Order = FALSE (with some more reaction time).

Integration of the reaction time in the invariant is not an elegant solution to formulate invariants which complicate the desired properties. Further work is needed to increase the readability of these properties.

One difficulty arises in the case studies when requirements refer to former values of signals. Invariants should only refer to signals present at the moment because only these signals are available in Event-B. Of course these values can be stored in separate variables in Event-B, but depending on the modelled system and the desired property this will be too cumbersome to be feasible.

Problems also occur if the requirement refers to real-time time limits since there is no real-time in Event-B. A possibility has to be found to map real-time to a time concept in Event-B.

Please note there are further types of requirements that are not suited for formalization as invariants, e.g., requirements that are fulfilled by the execution order of the system.

Summary With Event-B and Rodin we were able to model the discrete part of our systems. Rodin has the great advantage of integrating the formal modelling phase and the formal verification phase so they can be treated in parallel — this is important in helping to eliminate errors as soon as possible. Processes like configuration management, variant management, team development, version management etc. have to be better supported. Scalability for industrial applications and more flexibility for decomposition and architecture have to be addressed in the future. For the formalization of requirements the concept of invariants in Event-B shows limitations. The concept of FLOWS is a start to address time and timing in Event-B.

6.4 Formal Verification

Constraints The formal modelling (see Section 6.3) and the formal verfication which is described in this section should not be seen as two separate, sequential tasks. During the development of the formal model the verification is started as soon as possible to find errors as early as possible.

The main constraint for formal verification is that effort needed to perform the proofs should be minimized. Therefore the selection of suitable tools which support a high level of automation is a key task to facilitate the use of formal methods.

The Rodin tool which is used for formal modelling also provides tool support for formal verification used in the case study to prove relevant properties of the system as well as consistency within the model.

Description of method There are the two main sources for proof obligation in Event-B: the refinement proofs and the proofs of invariants. Refinement proofs concentrate on the stepwise development of an Event-B model. Invariants address (amongst other, more technical issues) the desired properties of the system which were derived from the requirements document.

As mentioned in Section 6.3 most of the properties, i.e., invariants, to be proven describe reaction of the system to certain inputs. To prove invariants referring to reactions, a number of supporting intermediate invariants are needed to guide the proof during the different steps the modelled system performs to react to an input (see Figure 8). This of course increases the number of necessary proof obligations. Future work is needed to investigate easier ways to address this problem.

We had over 4000 generated proof obligations in the Start/Stop System, around 90% of proof obligations were proven automatically by the provers integrated in Rodin. A large majority of the remaining manual proofs were very simple and might be proven automatically in the future with better adjustment and further development of the provers.

Using only the provers integrated in Rodin has one drawback: if a proof fails it is not always easy to find the reason: The problem might be in the model itself, in the invariant or it could be that the prover was simply not able to find the proof and needs manual support. One possiblity is to use the model checker ProB [24], [30] to generate counterexamples to ease the treatment of the problem.

Summary With Rodin we were able to address formal modelling and formal verification in parallel. The provers integrated in Rodin support a large number of automated proofs which is a very promising result. Using the model-checker ProB in addition to the provers in Rodin is a possibility to generate counterexamples that ease the treatment of errors.

6.5 Code Generation

Constraints After having spent considerable effort to develop a proven formal model it should be used to generate code. The properties proven for the formal model have to be preserved in the code. Therefore there is a strong need for a certified code generator, i.e., a code generator that is proven to correctly translate Event-B models into executable code. Furthermore, it would be desirable if the code generator could be configured to produce target code for different embedded processors.

Description of method Code generation from Event-B models is currently ongoing research. Different approaches for code generation exist [25], [31]. As a proof of concept we have applied the code generator EB2C [25] to generate C-code from our Event-B models of the Start/Stop System. Using this approach every event in the Event-B model is translated into a C-function. Depending on the order of events specified in the Event-B model these functions are called from the main function of the C-program.

Summary Although we have applied EB2C as a proof of concept for generating code from our Event-B models the existing code generators are not yet stable enough to be used in industry. Further work is required to make these code generators stable and flexible enough to be deployed in industry.

7 Related Work

Costs and benefits of model-based development of embedded systems in the automotive industry have been examined in [32]. The book chapter describes the results of a global study by Altran Technologies, the chair of software and systems engineering and the chair of Information Management of the Technical University of Munich. This work intends to cover a gap in research analysing the status quo of model-based development and its effects on the economics. One of the authors of this work, Manfred Broy, has a vast literature on software engineering methods applied to the automotive sector, for example [33]. In [34] he presents a perspective which is very close to the one supported by our work. In his paper, Broy, discusses the need for a portfolio of models and methods and he emphasizes the importance of tool support. The paper argues that there is already scientific evidence to support the idea that solid engineering of software intensive systems can be achieved in the future, provided a number of issues are addressed. The problems tackled in our work have certainly been investigated by other authors as well. For example, in [35], the authors propose a methodology for safe integration of automotive software functions while in [36] Adaptive Cruise Control is discussed and formal verification results are presented to guarantee collision freedom (the reader will have to note that *Adaptive* Cruise Control is somewhat different from the case study considered in DEPLOY [3]).

As all of these works demonstrate, automotive applications are now at the centre of several research projects because of their increasing complexity and relevance in the car industry. The need for novel design and validation methods, and also for new tools able to improve robustness and safety led to the organization of a dedicated workshop [37]. The papers contained in the proceedings are of great interest for anyone working in this field. The broader theme of industrial adoption of formal methods (not limited to the automotive sector) has been discussed in [38]. This paper starts from the consideration that, historically, the use of formal methods originated and concentrated mostly in Europe and they have been used only by big companies developing safety critical applications. However, the author sees the adoption of formal methods increasing in other parts of the world and he discusses, in particular, the South America and Far East scenarios.

8 Conclusions

This article discusses several software engineering issues, some of which are still open at present. The lack of a rigorous and repeatable approach of many ”formal methods” significantly restricts the choice when it comes to identify a suitable formalism for a specific problem. In [12] this issue is historically investigated and the requirements of a ”formal method” are identified to discover that many so-called “methods” are actually no more than notations, i.e., just formalisms without an attached rigorously defined and repeatable, systematic approach. Event-B is not one of those. Its refinement strategy has been demonstrated to be useful when applied to several case studies in a number of projects like RODIN [39] and DEPLOY itself [15]. However, not even Event-B is a panacea applicable to every phase of software development. In this article, we present a strategy based on a formalism-based toolkit, i.e., a portfolio of formalisms where every specific phase of development has been attacked by a different and suitable notation. The adoption of a portfolio of instruments is supported by other research as discussed in Section 7. The overall strategy proved to be a successful one and, given the thorough documentation generated by the project ([40], [41], [26]), it promises to be repeatable by engineers with an initially limited knowledge of formal methods. The importance of training here cannot be underestimated.

We believe this work has clarifed several aspects of industrial deployment of formal methods in automotive applications. At the same time, the DEPLOY project also addressed other industrial sectors (transport, business and aerospace). The issues discussed in this article emphasize the importance of scalability and applicability/effectiveness of specific methods; the proposed solutions have not ignored these major aspects as explained in detail in Section 5.

Application of formal methods is still considered controversial by some researchers [42] and we believe that not addressing at least some of the criticisms here would be a missed opportunity. Formal methods are considered attractive by many researchers because concepts such as theorems, proof obligations, equations and others can be applied. However, academic attractiveness by itself does not justify industrial deployment. The work presented in this article shows how elaborating a methodolgy based on a portfolio of different formalims, each tuned to a specific phase of development, allowed for a better set of requirements and, eventually, better code. Another criticism is often based on the idea that specifications fulfilling the requirement of being interpreted formally are hard to write when compared with learning a new

[3] to learn more: www.youtube.com/watch?v=alS6EqpqT0E

programming language. DEPLOY, and in particular the work presented here, actually demostrated the opposite. On the other hand, the criticism that it is not possible to prove that formal methods can offer the same quality for less is still open, i.e., we have not empirically (numerically) shown that formal methods are cheaper. There is high confidence that the quality is better, but the added value is limited when the quality is already very good. This research direction is already active [32], collecting empirical evidence is part of our future objectives.

Overall, this article rejects the idea that a single notation can be a panacea and solve all the problems encountered in software development. At a first sight, this conclusion may also appear trivial and not of much significance but, looking at the amount of research and publications pointing in the opposite direction, we felt the need to emphasize this argument more strongly. Not surprisingly, we made faster progress by moving to the position of having a toolkit/portfolio of notations and using those we felt could assist us better in every single phase. It is very interesting to note how Maslow's law of the instrument [18] plays its role here. It is well known from quantum physics that experimenters do alter their outcome at the subatomic level. Funnily, at the software development level, this may also happen for cognitive reasons and prejudice and DEPLOY brought this point up quite early and clearly; it is one of the major lessons learnt from the project.

Acknowledgment

This work has been funded by the EU FP7 DEPLOY Project (Industrial deployment of system engineering methods providing high dependability and productivity). More details at www.deploy-project.eu.

References

[1] G. Booch, *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004.

[2] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, 1997.

[3] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language, Third Edition*. Addison-Wesley Professional, 2003.

[4] J. C. M. Baeten, "A brief history of process algebra," *Theor. Comput. Sci.*, vol. 335, no. 2-3, pp. 131–146, 2005.

[5] J.-R. Abrial, S. A. Schuman, and B. Meyer, *A Specification Language*. New York, NY, USA: Cambridge University Press, 1980.

[6] J.-R. Abrial, *The B-Book: Assigning programs to meanings*. New York, NY, USA: Cambridge University Press, 1996.

[7] ——, *The Event-B Book*. Cambridge, UK: Cambridge University Press, 2010.

[8] D. Bjørner and C. B. Jones, Eds., *The Vienna Development Method: The Meta-Language*, ser. Lecture Notes in Computer Science. Springer-Verlag, 1978, vol. 61. [Online]. Available: https://www.springerlink.com/content/ql766633l472/

[9] C. B. Jones, *Software Development: A Rigorous Approach*. Englewood Cliffs, N.J., USA: Prentice Hall International, 1980. [Online]. Available: http://portal.acm.org/citation.cfm?id=539771

[10] ——, *Systematic Software Development using VDM*, 2nd ed. Prentice Hall International, 1990. [Online]. Available: http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/Jones1990.pdf

[11] M. Mazzara and A. Bhattacharyya, "On modelling and analysis of dynamic reconfiguration of dependable real-time systems," in *DEPEND, International Conference on Dependability*, 2010.

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

[13] ——, "On methods for the formal specification of fault tolerant systems," in *DEPEND, International Conference on Dependability*, 2011.

[14] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal Methods: Practice and Experience," *ACM Computing Surveys*, vol. 41, no. 4, Oct 2009.

[15] "DEPLOY: Industrial deployment of system engineering methods providing high dependability and productivity." [Online]. Available: http://www.deploy-project.eu/

[16] M. Jackson, *Problem Frames: Analyzing and structuring software development problems*. Addison-Wesley, 2000.

[17] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese, "Requirements Specification for Process-Control Systems," *IEEE Trans. Softw. Eng.*, vol. 20, pp. 684–707, September 1994. [Online]. Available: http://portal.acm.org/citation.cfm?id=188229.188234

[18] A. H. Maslow, *The Psychology of Science: A Reconnaissance*. Harper & Row, 1966.

[19] M. Heimdahl and N. Leveson, "Completeness and consistency in hierarchical state-based requirements," *Software Engineering, IEEE Transactions on*, vol. 22, no. 6, pp. 363 –377, jun 1996.

[20] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, pp. 231–274, June 1987. [Online]. Available: http://dl.acm.org/citation.cfm?id=34884.34886

[21] "Overture: Formal modelling in VDM." [Online]. Available: http://www.overturetool.org

[22] "Event-B and the Rodin Platform." [Online]. Available: http://www.event-b.org/

[23] "ISO 26262 Road Vehicles – Functional Safety." [Online]. Available: http://www.iso.org

[24] M. Leuschel and M. J. Butler, "ProB: an automated analysis toolset for the B method," *STTT*, vol. 10, no. 2, pp. 185–203, 2008. [Online]. Available: http://dx.doi.org/10.1007/s10009-007-0063-9

[25] D. Méry and N. K. Singh, "Automatic code generation from Event-B models," in *Proceedings of the Second Symposium on Information and Communication Technology*, ser. SoICT '11. New York, NY, USA: ACM, 2011, pp. 179–188. [Online]. Available: http://doi.acm.org/10.1145/2069216.2069252

[26] K. Grau, R. Gmehlich, F. Loesch, J.-C. Deprez, R. D. Landtsheer, and C. Ponsard, "DEPLOY Deliverable D38: Report on Enhanced Deployment in the Automotive Sector," DEPLOY Project, Tech. Rep. D38, 2011.

[27] N. G. Leveson, M. P. E. Heimdahl, and J. D. Reese, "Designing Specification Languages for Process Control Systems: Lessons Learned and Steps to the Future," in *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, ser. LNCS, vol. 1687, September 1999, pp. 127–145.

[28] "Event-B language, RODIN deliverable 3.2." [Online]. Available: http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf

[29] A. Iliasov, "Use Case Scenarios as Verification Conditions: Event-B/Flow Approach," in *SERENE*, ser. Lecture Notes in Computer Science, E. Troubitsyna, Ed., vol. 6968. Springer, 2011, pp. 9–23.

[30] R. Gmehlich, K. Grau, S. Hallerstede, M. Leuschel, F. Lösch, and D. Plagge, "On fitting a formal method into practice," in *Proceedings ICFEM'2011*, ser. Lecture Notes in Computer Science, S. Qin and Z. Qiu, Eds., vol. 6991. Springer, 2011, pp. 195–210.

[31] A. Edmunds, A. Rezazadeh, and M. Butler, "From Event-B Models to Code: Sensing, Actuating, and the Environment," in *SBMF2011*, September 2011. [Online]. Available: http://eprints.ecs.soton.ac.uk/22771/

[32] M. Broy, S. Kirstan, H. Krcmar, B. Schaetz, and J. Zimmermann, "What is the benefit of a model-based design of embedded software systems in the car industry?" in *Emerging Technologies for the Evolution and Maintenance of Software Models*. IGI Global, 2011, pp. 410–443. [Online]. Available: http://www.igi-global.com/book/emerging-technologies-evolution-maintenance-software/55286

[33] M. Broy, "Challenges in automotive software engineering." in *ICSE'06*, 2006, pp. 33–42.

[34] ——, "Seamless method- and model-based software and systems engineering," in *The Future of Software Engineering*. Springer, 2010, pp. 33–47.

[35] M. Jersak, K. Richter, R. Ernst, T. U. Braunschweig, J. christian Braam, Z. yu Jiang, and F. Wolf, "Formal methods for integration of automotive software," in *In IEEE Design, Automation and Test in Europe Conference and Exhibition (DATE03 Designers Forum), IEEE*, 2003.

[36] S. M. Loos, A. Platzer, and L. Nistor, "Adaptive cruise control: Hybrid, distributed, and now formally verified," in *FM*, 2011, pp. 42–56.

[37] *CARS '10: Proceedings of the 1st Workshop on Critical Automotive applications: Robustness and Safety*. New York, NY, USA: ACM, 2010.

[38] "Formal methods in industry: The state of practice of formal methods in South America and Far East." [Online]. Available: http://deploy-eprints.ecs.soton.ac.uk/169/

[39] "RODIN: Rigorous Open Development Environment for Complex Systems." [Online]. Available: http://rodin.cs.ncl.ac.uk/

[40] C. Jones, "DEPLOY Deliverable D15: Advances in Methodological WPs," http://www.deploy-project.eu/pdf/D15final.pdf, Tech. Rep. D15, 2009.

[41] F. Loesch, R. Gmehlich, K. Grau, M. Mazzara, and C. Jones, "DEPLOY Deliverable D19: Pilot Deployment in the Automotive Sector," DEPLOY Project, Tech. Rep. D19, 2010.

[42] "Managing automotive software architectures." [Online]. Available: http://automotive-sw-architecture.blogspot.com