Effective Development and Verification of Railway Control Software*

Anne E. Haxthausen
DTU Informatics
Technical University of Denmark
DK-2800 Lyngby
ah@imm.dtu.dk

January 21, 2011

Abstract

This document presents a method for effective development of software for a product line of similar railway control systems. The software is constructed in three steps: first a specifications in a domain-specific language is created, then a formal behavioural controller model is automatically created from the specification, and finally the model is compiled into executable object code. Formal verification is performed automatically by tools at three levels: (1) the specification is checked to follow the rules of the domain, (2) the controller model is checked to ensure safety, and (3) the object code is verified to be a correct implementation of the controller model.

*This document is a delivery to Rail Net Denmark (Banedanmark) as a part of the Public Sector Consultancy service offered by the Technical University of Denmark.

Contents

1 Introduction

This document describes and recommends some research-based ideas and emerging trends for the development and verification of railway control software.

Background

January 2009 the Danish parliament decided to replace all the Danish railway signalling systems with new, modern railway control systems based on the European standard ERTMS. Central parts of these systems consist of safety-critical software that must be developed according to the CENELEC EN50128 standard for railway applications.

Such a large scale replacement gives rise to many challenges. One is how to develop the software to achieve the required safety integrity levels of the CENELEC EN50128 standard. Another challenge is how this can be done efficiently to keep the costs down and to achieve a shorter time-to-market period. This document suggests how to help these two challenges using re-configurable software, domain-specific languages, formal methods, and a higher degree of automation in the development process. The suggestions are based on research made by the author and Jan Peleska and their research groups.

Paper overview

First, in sections 2–3, it is described how railway control systems are conventionally developed. Then, in sections 4–5, ideas and recommendations for how the development process can become more efficient using a domain-specific language, formal methods and automation are given. In section 6 these ideas are put together to provide a complete method for automated, model-based development of a product line of railway control systems. In section 7 it is discussed how to develop the development tools to be used in the method. Section 8 gives a reference to a case study (a German tram control system) to which the method has been applied.

2 Re-configurable systems

A characteristic feature of railway control systems is the need for making an individual system for each installation. The reason for this lies in the fact that the requirements to each control system typically depend on individual parameters such as the railway network to be controlled and allowed train routes through that network. However, it is usually possible (and also a common practise) to design the software such that it consists of (1) a generic part that can be re-used for many systems and (2) data that is individual for each system. The latter is called the application data or configuration data, and the whole system is said to be *re-configurable*. This idea is illustrated in figure 1.

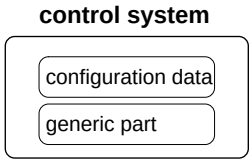

Figure 1: A re-configurable control system consisting of configuration data and a generic part.

Recommendation 1 It is strongly recommended to use re-configurable software systems as this allows for re-using generic software.

3 Conventional development of reconfigurable systems

Typically the development of reconfigurable systems proceeds along the following lines:

- Specification and design of a *generic control system* which can be instantiated with *configuration data* for concrete domains under control.
- *Manual* software development of generic system in programming languages like C/C++ or domain-specific languages (Sternol).
- *Informal, manual* verification of generic system (*"type certification"*).
- For each installation (i.e. for each concrete domain under control):
  - *Manual* instantiation of generic system by means of configuration data.
  - *Informal, manual* verification of the configuration data.
  - Generation of executable code using validated compilers.
  - Testing of the resulting concrete system.

Analysing this process it is notable that there are many manual tasks that take time. The manual construction of configuration data also requires some understanding of the software implementation, and may therefore be a potential source of errors. Furthermore, as the verification activities are informal and manual (typically using inspection techniques) some errors might be overseen. In best case these are found by the later testing, but this is not sure. One reason for this is the fact that testing is not exhaustive. The testing of the generic system can also only be done for a limited number of configuration data. Furthermore, often the testing of an instantiated, concrete system is done manually (without using an automated testing tool). As manual tests are very monotonous, it is very easy to lose the concentration and oversee some errors.

Hence, experience shows that it happens that errors are not found, despite the fact that much efforts have been put into the testing.

To make the development faster and catching more errors as early in the development cycle as possible, this motivates for the use of tools for

- automated construction and verification of configuration data and
- automated, formal verification of each instantiated system.

The next sections describe a development approach using such tools.

The above discussion also motivates for tools for automated testing and test case generation. It is out of the scope of this document to discuss that.

4 Automated construction from domain-specific descriptions

In recent years, *domain-specific, generative methods*[1] for software development have gained wide interest. One of the main objectives addressed by these methods is the possibility for a given domain to re-use various artifacts (e.g. code) when developing software.

The re-use of software for a family of similar systems can e.g. be obtained by developing re-configurable systems as suggested in section 2. Domain-specific methods typically use domain-specific languages and application generators for the construction of re-configurable applications. An *application generator* is a tool that takes a specification of an application as input and returns an application as output. It yields this application by instantiating the generic part of the application with configuration data that it derives from the specification. The specifications are formulated in a domain-specific language (DSL). In contrast to general-purpose specification and programming languages, a *domain-specific language* is a language dedicated to a specific application domain by using the terminology of that domain. Hence, it can be used by domain experts who are not specialists in the field of information technology. Typically the *applications* are software source code written in a high-level programming language, but they can also be design specifications or models written in a high-level design specification or modelling language for which there is a code generator or a compiler into machine code.

I suggest to use these ideas for the development of railway control systems. This means that for the construction of a family of similar control systems one should provide a *development framework* consisting of

- a domain-specific language (DSL) for specifying application-specific parameters using terms and concepts from the railway domain (that could for instance be track layouts and interlocking tables)
- an editor to support the editing of specifications in the domain-specific language

[1] For a good text book on this subject, see [2].

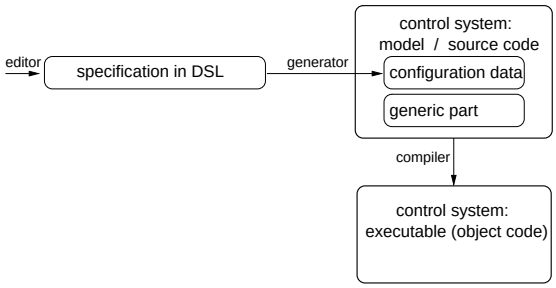

Figure 2: Generating a control system from a specification in a domain-specific language.

- a control system generator tool that takes DSL specifications as input, generates configuration data and combines this with a generic part common for all systems of the considered family

Hence, for each control system to be developed, the railway specialists should (1) use the editor to specify the application-specific parameters in the domain-specific language, (2) apply the generator to the specification to automatically generate a high-level description of the software (source code or model), and (3) then apply a compiler to this to produce an executable control system. This three step construction process is illustrated in figure 2. After the three steps the software should be integrated with hardware, but this step is out of the scope of this paper. Verification of the three first steps are discussed in next section. It is suggested to use the concepts of [1] to automatically test the software and hardware/software integration.

An advantage of using an application generator lies in the fact that it is much simpler to specify the parameters of a system in the domain-specific language and then apply a generator to produce the configuration data, than it is to program the configuration data directly. This speeds up the production time and reduces the risk of errors; furthermore, it can be done by domain experts without requiring the assistance of programming specialists.

Recommendation 2 It is recommended to provide a *domain-specific language* to specify the application-specific parameters of re-configurable control systems and an *application generator* to automatically construct configuration data from such specifications.

To facilitate later formal verification (see section 5.2) of the output of the application generator in the second step, it is recommended to let this output be a formal, verifiable model encoded in a high-level language such as SystemC [6] allowing it to be formally verified by a model checker tool as well as being compiled into executable code. (In this way model and source code coincide.)

5 Automated verification

For each of the three considered development steps, verification of the produced artifacts (specifications in DSL, control system models/code in a high-level modelling/programming language and executable control systems in an assembly or machine language, respectively) should be done. For the highest safety integrity levels, the CENELEC standard EN50128 strongly recommends to use formal verification methods[2] for that.

Below is suggested how to automate the formal verification of each of these three steps by providing new or using existing verification tools.

5.1 Specification checking

First (in step 1), when an application specification in a domain-specific language has been created, this has to be checked to be syntactically correct and well-formed. For instance, if the specification consists of a track layout and a train route table, one of the well-formedness checks could be that the points mentioned in the table are part of the track layout.

Recommendation 3 To formalise and automate this verification activity, it is suggested to provide a *specification checker* that automatically checks the syntax and all well-formedness requirements. This specification checker might be integrated with the editor.

5.2 Model checking

Secondly (in step 2), when (a model of) the control system has been generated from the domain-specific specification, this has to be verified to satisfy required safety properties (as, for example, the requirement that trains never meet at a track section).

5.2.1 Formalising the verification task

A common practise to perform such a verification task formally and at the same time fully automated is to use a model checker tool[3]. Such a tool needs as input:

- a *controller model*, i.e. a model of the behaviour of the control system,
- a *domain model*, i.e. a model of the behaviour of the physical environment[4] of the control system, and
- a formal specification of the *safety properties* that the system must fulfil.

[2] See [7] for an introduction to formal methods.

[3] See [7] for a description of the notions of model checking and model checkers.

[4] The environment consists of objects such as points and trains with which the control system interacts.

The *models* should represent a state transition system describing how the state of the system and its environment (when operating together) can evolve over time[5], and the safety properties should be some constraints on how the state is *allowed* to evolve over time. To be more precise the models should together include descriptions of:

- the state space (i.e. all states that can be obtained as combinations of states of the controller and the states of objects in its environment)
- the initial state(s)
- possible state transitions

A graphical illustration of a state transition system is given in figure 3.

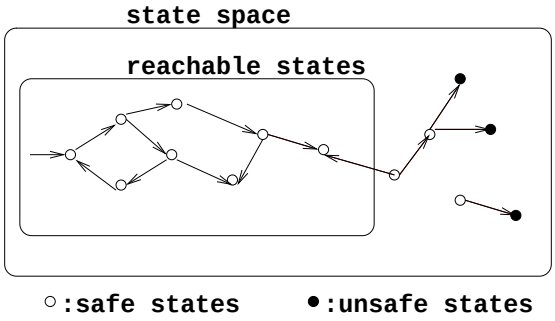

Figure 3: A state transition system consisting of (1) a state space of all possible combinations of states for individual objects (shown as black and white dots) and (2) possible state transitions from one state to another (shown as arrows). The state that only has an incoming arrow is an initial state. Only some of the states are reachable from the initial state.

The *formal specification* of the required safety properties should be a logical expression that can be used to determine which states are safe (shown by white dots in figure 3) and which are unsafe (shown by black dots in figure 3).

The *verification task* is then to check that the unsafe states can never be reached from the initial state by a sequence of state transitions (following the arrows in figure 3). In other terms it means that it should be checked that no unsafe state is within the set of reachable states. The process of making this checking is called *model checking*.

Model checking can be fully automated using a model checker, but may lead to state space explosions (i.e. the model checker tool runs out of memory) for railway control systems of realistic size. To avoid that problem it is recommended to use a special technique combining bounded model checking with inductive reasoning. The details of this technique are described in [8].

[5]Note, for concurrent, reactive systems, like railway control systems, there are usually many different ways in which the state can evolve over time.

Recommendation 4 It is recommended to use *bounded model checking and inductive reasoning* to verify the safety of the controller formally and at the same time fully automated.

5.2.2 Generating input to a model checker

The question is now: How should the models and safety properties be created?

The answer to this question is: The controller model should simply be the output generated in development step 2, cf. the discussion in the end of section 4. The domain model and safety properties should also be derivable from the application specification in DSL. (One should ensure that DSL specifications provide enough information such that this is possible.)

Recommendation 5 To automate the derivation of the formal models and safety conditions (the input to the model checker), it is recommended to provide *generator tools* that take a DSL specification as input and automatically generate these.

5.3 Object code verification

Finally (in step 3), when the control system model/code has been compiled into object code, it should be verified that the object code correctly implements the control system/model. According to the CENELEC EN58128 standard it is sufficient to use a validated/certified compiler. However, if an un-certified compiler is used or it is desirable to be more confident about the correctness, formal verification can be used for that.

The conventional approach for this is *compiler validation*: "once-and-for-all" it is validated that the compiler for any input produces object code that is a correct implementation of that input. However, such an approach is very time-consuming, especially if it should be done formally (see e.g. [5] for techniques for that), and furthermore it has to be performed again whenever modifications of the compiler have been performed. An alternative to compiler validation is *object code verification*: each time object code is generated (by an arbitrary compiler), the generated object code is verified to be a correct implementation of the high-level software model/code from which it was generated. Object code verification has the advantage over compiler verification that it is independent of changes in the compiler and can potentially be automated. The automation of object code verification is an ongoing research topic for which ideas have been given in [10].

Recommendation 6 To automate object code verification, it is suggested to provide an *object code verifier* that automatically performs the object code verification.

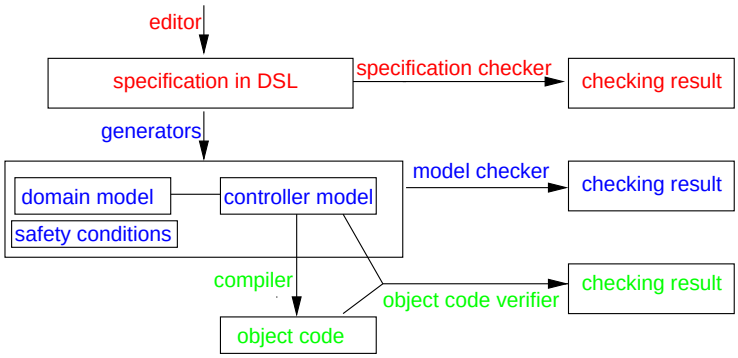

Figure 4: Development and verification steps.

6 Summary

Combining all the suggestions and recommendations given above this gives a complete model-driven development and verification approach for railway control systems. According to this approach, in order to develop software for a product line of similar railway control systems one should provide a framework (see figure 4) consisting of:

1. A domain-specific language (DSL).
2. A collection of development tools, including
   (a) a DSL specification editor and well-formedness checker,
   (b) generators producing models of the control system and its physical environment as well as safety conditions,
   (c) a model checker,
   (d) a compiler, and
   (e) an object code verifier.

For each control system to be generated, the user should use the editor to specify the application-specific parameters in the domain-specific language and check the description by means of the specification checker. Next, the generators produce models of the control system and its physical environment from this specification, together with the safety requirements which are automatically verified using the model checker. Finally – since the formal controller model can be directly compiled – object code is generated by a conventional compiler, and it is checked by the object code verifier that the object code is behaviourally equivalent to the control system model. In this way it is ensured that the safety properties established for the control system model also hold for the object code.

7 Development of the development tools

Since the suggested development process is based on the use of a domain-specific language and some development tools (editor, generators and checkers), these tools should also be developed according to the CENELEC standard. One possibility is to use formal methods for that. Formal specification of the DSL language and tools can for instance be done in a formal specification language such as RSL [4], VDM [9, 3] or Z [11] that is suited for the specification of data types and associated functions. Another possibility is to use language development frameworks that facilitate the development of languages and tools such as editors and code generators.

8 Case study

A case study has been performed applying the presented ideas to a tramway control system in Germany. This case study is presented in [8].

9 Related Work

The ideas given in this presentation are based on research that has been made by the author and Jan Peleska and their teams.

Acknowledgements

I would like to express my thanks to Jan Peleska for his visionary ideas and for collaborating with me over many years on the ideas presented in this paper. Finally, but not least, I would like to thank Kirsten Mark Hansen, Banedanmark, for very useful feedback on an earlier version of this document.

References

[1] B. Badban, M. Fränzle, J. Peleska, and T. Teige. Test automation for hybrid systems. In *Proceedings of the Third International Workshop on SOFTWARE QUALITY ASSURANCE (SOQUA 2006)*, Portland Oregon, USA, November 2006.

[2] Ulrich W. Eisenecker and Krzysztof Czarnecki. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[3] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, Second edition, 2009.

[4] Chris George, Peter Haff, Klaus Havelund, Anne E. Haxthausen, Robert Milne, Claus Bendix Nielsen, Søren Prehn, and Kim Ritter Wagner. *The RAISE Specification Language*. The BCS Practitioners Series. Prentice-Hall, 1992.

[5] Gerhard Goos and Wolf Zimmermann. Verification of compilers. In *Correct System Design*, pages 201–230. Springer, 1999.

[6] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[7] Anne E. Haxthausen. An Introduction to Formal Methods for the Development of Safety-critical Applications. Technical report, DTU Informatics, Technical University of Denmark, August 2010.

[8] Anne E. Haxthausen, Jan Peleska, and Sebastian Kinder. A Formal Approach for the Construction and Verification of Railway Control Systems. *Formal Aspects of Computing*, online first 2009. Special issue in Honour of Dines Bjørner and Zhou Chaochen on Occasion of their 70th Birthdays.

[9] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1990.

[10] Jan Peleska and Anne E. Haxthausen. Object Code Verification for Safety-Critical Railway Control Systems. In *Proceedings of Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2007), Braunschweig, Germany*. GZVB e.V., 2007. ISBN 13:978-3-937655-09-3.

[11] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.