In: Railways: Types, Design and Safety Issues ISBN:978-1-62417-139-0
Editors: C. Reinhardt and K. Shroeder

Chapter 5

EFFICIENT DEVELOPMENT AND VERIFICATION OF SAFE RAILWAY CONTROL SOFTWARE

Anne E. Haxthausen[1,*] ***and Jan Peleska***[2,†]
[1]DTU Informatics, Technical University of Denmark,
Kongens Lyngby, Denmark
[2]Department of Mathematics and Computer Science,
Universität Bremen, Germany

Abstract

This chapter describes some approaches and emerging trends to ensure traffic safety at railways. The focus is on railway control systems whose role it is to ensure safe train movements through the railway network. Such systems are clearly safety-critical as failures may endanger human lives, and therefore they are subject to various standards (like the CENELEC standards used in Europe) to ensure a certain level of safety integrity. There are many challenges in developing railway control systems. One is to ensure safety. The chapter describes a trend of using mathematically well-founded models in the railway system development process making it possible to formally analyse the systems before they are built, just like models are used in other engineering disciplines. Another challenge is the demand for shorter time-to-market periods and higher competition among suppliers. In this chapter we suggest how to help this using

[*]E-mail address: ah@imm.dtu.dk
[†]E-mail address:jp@informatik.uni-bremen.de

a higher degree of automation in the development, verification, validation and test processes.

Keywords: railway control systems, formal methods, verification, testing

1. Introduction

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

These and coming years many countries are installing new, modern computer-based railway control systems[1]. These systems become more and more advanced, and thereby also more and more complex. Consequently, there are many challenges in developing such systems. Central parts of the modern systems consist of safety-critical software. As it is well-known that software is often full of bugs, a major challenge is to develop the software in a way that minimises the risk of errors. Another challenge is how this can be done efficiently to keep the costs down and to achieve a shorter time-to-market period. This chapter 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.

Paper overview. First, in sections 2–3, it is described how railway control systems are conventionally developed, and problems of the conventional process are identified. Then, in sections 4–6, ideas and recommendations for how the development process can become more effective and efficient using a domain-specific language, formal methods and automation are given. In Section 7 these ideas are put together to provide a complete method for automated, model-based development of a product line of railway control systems. Section 7 also gives references to two case studies to which the method has successfully been applied. Finally, a conclusion is drawn in Section 8.

[1] In this chapter we use the term *railway control system* as a common term for all software systems used for safe control of railways. This includes e.g. interlocking systems and train control and protection systems.

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 – in case of an interlocking system – the railway network to be controlled and allowed train routes through that network. Also train control computers are configured, for example, with the track atlas for the routes it is supposed to travel on. However, it is usually possible (and also a common practice) 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.

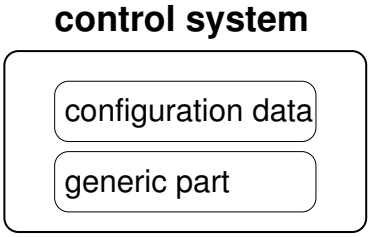

Figure 1. A re-configurable control system consisting of configuration data and a generic part.

Recommendation 1 It is recommended to use re-configurable software systems as this allows for re-using generic software[2].

3. Conventional Development of Re-configurable Systems

Typically the development of re-configurable systems proceeds in the following steps:

[2]Indeed, according to the authors' knowledge about train control systems built by European manufacturers, this recommendation is already considered as a "best practice" today.

- Specification and design of a *generic control system* which can be instantiated with *configuration data* for concrete domains under control.
- *Manual* software development of the generic system in programming languages like C, C++, and Step7 or domain-specific languages like Sternol.
- *Informal, manual* verification and validation (V&V) of the generic system for the purpose of *type certification*[3].
- For each installation (i.e. for each concrete domain under control):
  - *Manual* instantiation of the 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 (hardware and integrated software).

Analysing this process it is notable that there are many manual tasks involving considerable effort. 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 and manually selected configuration test cases) some errors might be overseen. In the best case these errors are found in the later testing stages, but this cannot be guaranteed. One reason for this is the fact that testing is not exhaustive because the testing of the generic system can only be performed for a limited number of configuration data, due to the unmanageable number of possible configurations. 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 the use of tools for

[3] In the railway domain the certification acknowledges a novel *type* of interlocking system, track element or train to be "generally fit for purpose". This implies that the system can be used in all different contexts which are based on the admissible configurations specified without requiring a re-certification.

- automated construction and verification of configuration data,
- automated, formal verification of each instantiated system,
- automated test case generation and test execution for the integrated HW/SW system.

The next sections describe a development approach using such tools.

4. Automated Construction from Domain-Specific Descriptions

In recent years, *domain-specific, generative methods*[4] 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 product line (family) of similar systems can 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.

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

[4]For a good text book on this subject, see [5].

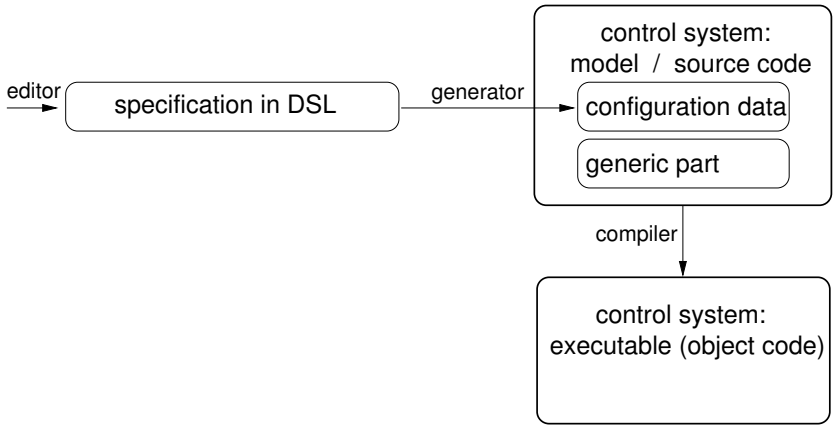

Figure 2. Generating a control system from a specification in a domain-specific language.

- 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
- 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 product line/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 (source code or model) of the software, and then

(3) apply a compiler to this in order to produce executable code ready to be integrated on the control system's target hardware.

This three-step construction process is illustrated in Figure 2. Verification and testing of the three steps are discussed in the next two sections.

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.

Numerous suggestions for domain-specific languages in the railway domain have been given in recent years; we mention [15] for DSLs applicable to interlocking system development and [16] to a DSL applicable to European Train Control System (ETCS) on-board computer software generation.

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 [10] 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 development steps considered above, verification of the produced artefacts (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) has to be performed.

For the highest safety integrity level, the European CENELEC standard EN50128 [6] for railway applications strongly recommends to use formal verification methods[5] for this purpose. This is motivated by the fact that formal methods are very strong in catching errors: instead of testing the correctness

[5]See [11] for an introduction to formal methods, and [7] for the role of formal methods in software development of railway applications.

of artefacts for some samples, say program states, formal methods aim at proving that the desired properties hold everywhere, for example, in every reachable state of a program. One of the most famous examples of this, is the formal verification of some safety-critical parts of the software for metro line 14 in Paris [2]. In this case many errors were found during the formal verification, and as a result of that no errors were found in the verified parts during the testing or while the line has been in operation.

Below is suggested how to automate the formal verification of each of the three steps (mentioned at the beginning of this section) by providing new verification tools as well as using existing verification tools.

5.1. Specification Checking

First (after completion of Step (1) above), when an application specification in a domain-specific language has been created, this has to be checked with respect to syntactic correctness and well-formedness. For instance, if the specification consists of a track layout, one of the well-formedness checks verifies that each point in the track layout is connected to three other track sections.

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 should be integrated with the editor.

5.2. Model Checking

Second, (in Step (2)), when a high-level description (in our case, high-level code and configuration data) of the control system has been generated from the domain-specific specification, this has to be verified to satisfy the required safety properties (as, for example, the requirement that trains never meet on a track section).

5.2.1. Formalising the Verification Task

A common practice to perform such a verification task in a mathematically formal, comprehensive, and at the same time fully automated way, is to use a model checker tool[6]. Such a tool needs as input:

[6]See [11] for a description of the notions of model checking and model checkers.

- a *controller model*, i.e. a model of the behaviour of the control system,
- a *physical model*, i.e. a model of the behaviour of the physical environment[7] of the control system, and
- a formal specification of the *safety properties* to be fulfilled by the system.

The models are represented by state transition systems describing how the state of the system and its environment (when operating together) *can* evolve over time[8]. The safety properties consist of constraints on how the state is *allowed* to evolve over time, so that the avoidance of hazards is ensured. To be more precise, the models contain descriptions of

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

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

The *formal specification* of the required safety properties is given by 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 words, it has to be verified 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, but may lead to state space explosions, with the consequence that the model checking tool runs out of memory while unfolding the state space for railway control systems of realistic size. To avoid that problem several techniques have been proposed [4]. A particularly promising method for the railway domain consists of *bounded model checking with inductive reasoning*: instead of unfolding the complete state space and

[7]For interlocking systems, the environment consists of track elements (points, signals, axle counters etc.) and of trains moving through the controlled network and interacting with the interlocking system.

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

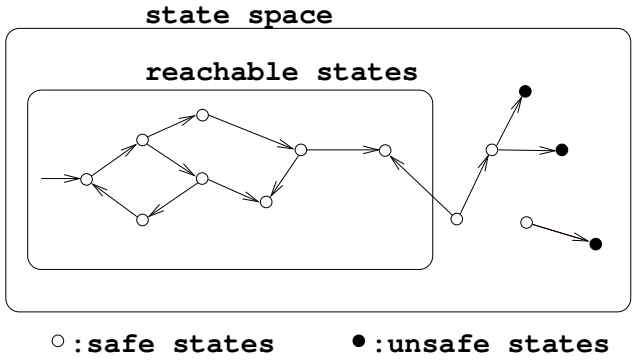

Figure 3. A state transition system consisting of (1) a state space of all possible combinations of states for individual objects (shown as black and white dots) and (2) possible state transitions from one state to another (shown as arrows). The state that only has an incoming arrow is an initial state.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

risking state explosions, only subsets of state configurations reachable within a limited number n of steps from the current configuration are investigated at a time. Inductive reasoning is then applied to show that if the system remains safe for at least n steps from any member of a given *subset* of configurations, it has to be safe in *every* reachable state. The details of this technique are described in [13].

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

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 physical model and safety properties should also be derivable from the application specification in DSL. It only has to be ensured that DSL specifications provide enough information for this purpose.

Recommendation 5 It is recommended to provide *generator tools* taking DSL specifications as input and transforming them into formal models and safety conditions (the inputs to the model checker) in an automated way.

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*: it is validated "once-and-for-all" that for any syntactically valid input the compiler produces object code which 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. [9] for a description of formal compiler validation techniques). Furthermore, the process has to be repeated 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 be automated in principle. The automation of object code verification is an ongoing research topic for which ideas have been given in [18].

Recommendation 6 It is recommended to use object code verification for un-certified compilers and in cases where a very high level of confidence is needed. To automate object code verification, it is suggested to provide an *object code verifier* that automatically performs the object code verification.

6. Test Automation

In the sections above we have sketched how a considerable amount of the development artefacts produced in a DSL-based transformational approach to railway control system development can be verified in an automated way, using model checkers and object code verifiers. It remains to be discussed whether

there are any testing activities left to be performed and – if any – how these should be executed in a way allowing for a maximum degree of automation.

6.1. Why Testing?

The foremost answer to this question is "Because it is required by the standards to be observed in the railway domain!": just as results from theoretical physics have to be validated by experiments, even a formally verified control system has to be validated by testing. There are no indications whatsoever that future versions of applicable standards might drop the requirement to test the outcome of a development, even if comprehensive formal correctness proofs were available. In-depth analysis of this question provides a more detailed line of reasoning why testing is an inevitable part of the whole V&V process.

Different verification perspectives. Assurance for safety-critical developments is achieved by applying different verification perspectives: The formal verification perspective focuses on the logical correctness of development artefacts *under certain hypotheses*. Applying our 3-step development approach to interlocking systems, for example, model checking relies on the hypothesis that the railway network has been described completely and without any errors in the DSL specification, and that the track elements involved will be installed as specified. Testing is a way of validating these hypotheses. Exercising each pre-planned route through the network during acceptance testing, for example, will make sure that the network has been built properly and with the intended elements. More subtle hypotheses, such as, for example, timing constraints to be met by the control system and its connected track elements, can only be validated by means of system integration testing[9].

Correctness of hardware/software integration. Even in presence of complete and correct software code, tests are required to verify that the generated machine code operates correctly on the target hardware: logically correct calculations may fail due to insufficient register word size, verified algorithms may produce errors on the target hardware because certain timing constraints between CPU, memory and interface busses and peripheral hardware are not met,

[9] There exist formal approaches to worst case execution time analysis, but these are still limited to local controller hardware and cannot be extended to complete systems involving networks of controllers, trains and hardware peripherals.

and computer networks could fail to operate correctly due to incompatible protocol implementations.

Main focus of testing. In the light of these considerations we conclude that testing is an inevitable activity among the V&V processes of the product life cycle. For the automated V&V approach advocated in this chapter, however, its focus is shifted from checking logical correctness of algorithms and software component integration to hardware/software and system integration.

6.2. Model-Based Testing

In model-based testing (MBT) test strategies, testing environments and the desired behaviour of the system under test (SUT) are expressed by models, allowing for a systematic (and even automated) derivation of concrete test procedures [1]. In its most powerful form, test cases, concrete test data and test procedures executing the test cases and checking SUT reactions against expected results are automatically generated from an SUT model and an optional environment model restricting the possible interactions of the SUT with its operational environment. Though still considered a "leading edge technology" today, automated MBT has established itself as a novel approach to testing that can efficiently handle industrial size problems [19], and performance data are available for application in the railway domain [14].

DSL specification as test model. The utilisation of MBT suggests itself for the V&V strategy described here, because models are already available on different levels of abstraction: the DSL specification of the control system to be developed is the most abstract model, and the concrete high-level code and associated configuration data is a more concrete variant. We select the DSL specification model as the input for MBT, because this also allows us to test against the generator transforming the DSL specification into the high-level description of control system and configuration data according to Step (2) described in Section 4: if the generator applied in Step (2) would "forget" to transform a portion of the DSL specification into configuration data, this can be uncovered during the MBT phase. Since the translation of the DSL specification into an internal representation suitable for test case and test data generation uses a transformation which does not share any algorithms or code with the generator of Step (2)

it is unlikely that both generator and MBT tool would simultaneously fail to process the same portion of the DSL specification correctly.

Requirements tracing. Systematic testing includes requirements tracing: each requirement is associated with one or more test cases suitable for checking its realisation by the SUT. The relation between requirements and test cases is usually $n : m$, because one requirement may need several test cases for its verification, and one test case may contribute to the verification of several requirements. Test cases are associated with test procedures executing them in a specific order. Each test procedure is linked to the test execution results produced during its execution. This results in a complete collection of links[10], so that each requirement can be traced to all the test results obtained for its verification.

Identifying requirements in the model – general case. In general the identification of requirements in a model is a manual process. It is performed by associating certain model elements or – in the most complex case – logical formula specifying certain traces[11] through the model with requirements. For example, the requirement "the train shall never exceed the admissible maximum velocity specified in the track atlas" would be linked to model elements responsible for determining the current position (odometer), for determining the current velocity (speedometer), for accessing the track atlas and for triggering the emergency brake in case of speed limit violations. All traces through the model leading to a speed violation and causing the emergency brake to be triggered are *witnesses* for the requirement. Test cases stimulating such witness traces in the SUT are suitable for testing the requirement. Some modelling formalisms support explicit representation of the links between requirements and model elements or model traces; the most prominent example is the System Modelling Language SysML [21].

Identifying requirements in the model – special case of automated railway control system developments. For railway control systems, at least the majority of safety-related requirements and a certain amount of user requirements

[10] This collection is usually called the *traceability matrix*. Observe, however, that more than one matrix is needed: one for the $n : m$ relation 'requirements $\leftrightarrow$ test cases', one for 'test cases $\leftrightarrow$ test procedures' (also $n : m$) and one for 'test procedures $\leftrightarrow$ test results' (1:1 relation).

[11] A *trace* is a finite sequence of model states which can occur during a simulation of the model.

can be automatically identified in the DSL specification model, because they stem from generic requirements that can be instantiated for the concrete control system to be developed. Consider, for example, an interlocking system whose DSL specification consists of the railway network to be controlled and interlocking tables describing the required routes through the network, their required point positions and mutual exclusion requirements for conflicting routes whose simultaneous allocation to trains would lead to collisions. Consider the following examples of generic requirements (a detailed presentation of this example is given in [17]).

- It shall be possible to allocate each specified route.
- Non-conflicting routes may be allocated in a concurrent way.
- No pair of conflicting routes shall be simultaneously allocated.
- Allocation to a train is only granted after the points are locked in the positions required for the requested route.
- De-allocation of a route is only performed when it is empty and entering the route has been blocked.
- If a point fails, every route visiting this point shall be blocked.

For the concrete railway network to be controlled each requirement is instantiated for the concrete routes to be provided through the network, for each subset of pairwise non-conflicting routes, for each pair of conflicting routes, and for each point, respectively.

Automated model-based testing. The automated identification of requirements also enables automated test case elaboration directly from the DSL specification model, by constructing witnesses for every concrete requirements. This process has been comprehensively described in [15]. We illustrate the technique using the requirements listed above in a HW/SW integration test scenario shown in Figure 4.

The SUT consists of one or more computers receiving route requests from trains and location information about trains obtained from sensors (e. g., axle counters). On reception of a non-conflicting request the SUT sends switching commands to the points on the route under consideration. It collects feed back information about the point status. Signals guarding conflicting routes are

switched into the 'halt' state. If all points could be locked in the requested position and the feed back from the signals indicates that entry to conflicting routes is blocked, the signal guarding the entry into the requested route is switched into the 'go' state. Feed back from points and signals is continuously monitored, so that failures lead immediately to blocking the routes affected by these failures.

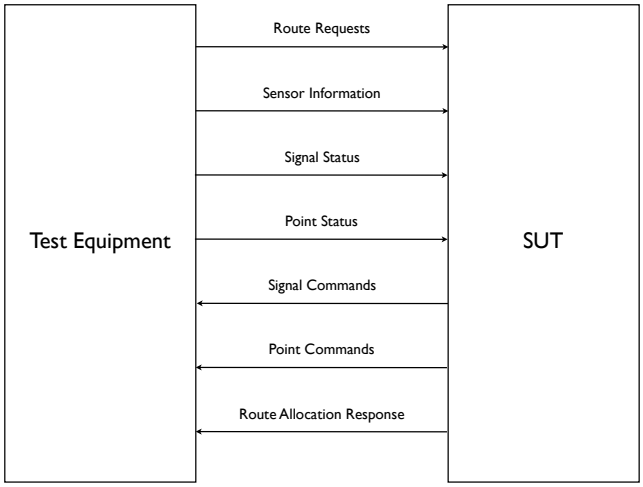

Figure 4. HW/SW integration test configuration for interlocking system controller.

The automated MBT generator reads the DSL specification and first creates *symbolic test cases* which are not yet associated with concrete test data. For every route r, for example, a symbolic test case specifies "request r while no conflicting routes have been allocated", another test case requires to "let train travel along the allocated route r until its end", so that route de-allocation can be checked. Further symbolic test cases deal with concurrent allocation of non-conflicting routes, request rejection for conflicting routes and point or signal failures.

Symbolic test cases are automatically transformed into concrete ones by means of libraries simulating concrete behaviours of trains when requesting and traveling on routes. Other library functions simulate the feed back behaviour of points and signals. Determining the parameters of these library calls may require a *constraint solver* calculating concrete data solving logical conditions: if a test case requires, for example, that route r should be de-allocated before

another route r', the solver determines admissible speed values for trains on r and r', so that the train leaves r while r' is still occupied. The library calls are compiled into test procedures which can be automatically executed on the test equipment. In a HW/SW integration test setting, all inputs to the SUT are generated by the test equipment consisting of one or more computers interacting with the SUT via its system interfaces. With the help of the library, the test equipment is capable of simulating route requests, sensor information and feed back from track elements as required by the test cases under consideration. Moreover, it receives SUT responses and checks them according to the expected results specifications associated with the test cases.

Recommendation 7 We recommend to perform automated model-based testing for ensuring the correctness of HW/SW integration and for validating the completeness of the automated generation of control software and configuration data.

7. The Complete Development Process

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 5) 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,
   (e) an object code verifier, and
   (f) an automated model-based test generator.

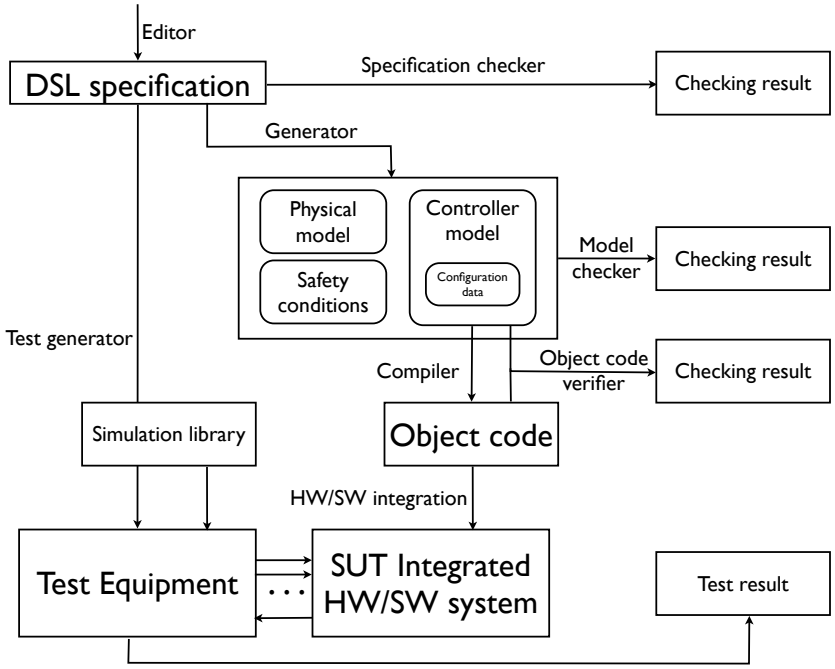

Figure 5. Development and verification steps.

For each control system to be generated, users should apply 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. Next – 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. Finally, HW/SW integration testing and – for larger distributed control systems – system integration testing ensures the correct cooperation of hardware and software, and validates the completeness of the generated control system code and configuration data.

Case studies have been performed applying the presented method to a tramway control system [13, 15] in Germany and a relay interlocking system in Denmark [12]. In these case studies, first prototypes of domain-specific languages and development tools were developed, and then these prototypes were successfully applied to construct and verify the systems. The model-based testing approach is currently applied in industrial testing campaigns for railway control systems [14].

Conclusion

In this chapter we have given a number recommendations for how the conventional development, verification and testing of railway control systems can be improved by using domain-specific languages, formal methods and automation. We suggested to use *formal methods* to achieve a high safety integrity level, *automation* to make the development, verification and testing processes more efficient, and *railway domain-specific languages* to provide more railway friendly user interfaces. Our recommendations have been successfully applied to several case studies, and the automated testing approach is already applied in industrial projects.

We plan in the future to apply and refine these ideas for the new ERTMS based railway control systems that are going to be implemented in Denmark over the next decade.

For complementary and competing approaches for the development, verification and testing of railway control systems the reader is referred to the contributions in proceedings like [20] and books like [8]. For a survey of recent trends and the role of formal methods in software development of railway applications, the reader is referred to [7, 3].

Acknowledgments

A part of this chapter is based on a delivery made to Rail Net Denmark (Banedanmark) as a part of the Public Sector Consultancy service offered by the Technical University of Denmark.

We would like to thank Kirsten Mark Hansen, Lloyd's Register EMEA, for reviewing the chapter and giving valuable feedback on that.

References

[1] P. Baker, Z. R. Dai, J. Grabowski, O. Haugen, I. Schieferdecker, and C. Williams. *Model Driven Testing – Using the UML Testing Profile*. Springer, Berlin, Heidelberg, New York, 2008.

[2] P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier. Météor: A Successful Application of B in a Large Project. In J. M. Wing, J. Woodcock, and J. Davies, editors, *Proceedings of FM'99: World Congress on Formal Methods*, Lecture Notes in Computer Science, pages 369–387. Springer-Verlag, 1999.

[3] D. Bjørner. New Results and Current Trends in Formal Techniques for the Development of Software for Transportation Systems. In *Proceedings of the Symposium on Formal Methods for Railway Operation and Control Systems (FORMS'2003), Budapest/Hungary*. L'Harmattan Hongrie, May 15-16 2003.

[4] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[5] U. W. Eisenecker and K. Czarnecki. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[6] European Committee for Electrotechnical Standardization. *EN 50128 – Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems*. CENELEC, Brussels, 2001.

[7] A. Fantechi. The Role of Formal Methods in Software Development for Railway Applications. In *Railway Safety, Reliability and Security: Technologies and System Engineering*, pages 282–297. IGI Global, 2012.

[8] F. Flammini, editor. *Railway Safety, Reliability and Security: Technologies and System Engineering*. IGI Global, 2012.

[9] G. Goos and W. Zimmermann. Verification of compilers. In *Correct System Design*, pages 201–230. Springer, 1999.

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

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

[12] A. E. Haxthausen. Towards a Framework for Modelling and Verification of Relay Interlocking Systems. In *16th Monterey Workshop: Modelling, Development and Verification of Adaptive Systems: the Grand Challenge for Robust Software*, number 6662 in Lecture Notes in Computer Science, pages 176–192. Springer, 2011. Invited paper.

[13] A. E. Haxthausen, J. Peleska, and S. Kinder. A Formal Approach for the Construction and Verification of Railway Control Systems. *Formal Aspects of Computing*, 23(2):191–219, 2011. Electronic version: DOI: 10.1007/s00165-009-0143-6.

[14] H. Löding and J. Peleska. Timed moore automata: Test data generation and model checking. In *International Conference on Software Testing, Verification, and Validation, ICST2008*, pages 449–458, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[15] K. Mewes. *Domain-specific Modelling of Railway Control Systems with Integrated Verification and Validation*. Verlag Dr. Hut, München, 2010.

[16] J. Peleska, J. Feuser, and A. E. Haxthausen. *The Model-Driven openETCS Paradigm for Secure, Safe and Certifiable Train Control Systems*, pages 22–52. IGI Global, June 2012.

[17] J. Peleska, D. Große, A. E. Haxthausen, and R. Drechsler. Automated verification for train control systems. In E. Schnieder and G. Tarnai, editors, *Proceedings of the FORMS/FORMAT 2004 - Formal Methods for Automation and Safety in Railway and Automotive Systems*, pages 252–265. Technical University of Braunschweig, December 2004. ISBN 3-9803363-8-7.

[18] J. Peleska and A. 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.

[19] J. Peleska, A. Honisch, F. Lapschies, H. Löding, H. Schmid, P. Smuda, E. Vorobev, and C. Zahlten. A real-world benchmark model for testing concurrent real-time systems in the automotive domain. In B. Wolff and F. Zaidi, editors, *Testing Software and Systems. Proceedings of the 23rd IFIP WG 6.1 International Conference, ICTSS 2011*, volume 7019 of *LNCS*, pages 146–161, Heidelberg Dordrecht London New York, November 2011. IFIP WG 6.1, Springer.

[20] E. Schnieder and G. Tarnai, editors. *Proceedings of Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2010), Braunschweig, Germany*. Springer, 2011.

[21] OMG Systems Modeling Language (OMG SysML$^{\mathrm{TM}}$). Technical Report Version 1.2, SysML Modelling team, June 2010. http://www.sysml.org/docs/specs/OMGSysML-v1.2-10-06-02.pdf.