# A formal and pragmatic approach to engineering safety-critical rail vehicle control software

Michael Wasilewski[1], Wilhelm Hasselbring[2]
[1]Vossloh Locomotives GmbH
24152 Kiel
http://www.vossloh-locomotives.com/
[2]Universität Kiel
Institut für Informatik, Software Engineering Group, 24118 Kiel
http://se.informatik.uni-kiel.de/

**Abstract:** The engineering processes for safety-critical systems, for instance in the health care or transportation domains, are regulated by law. For software in the railroad industry in Europe the certification procedures have to obey the norm EN50128.

This paper presents the method that was introduced and employed for the development and the successful certification of the software for the vehicle control unit (VCU) of the Vossloh Locomotives' G6 shunting locomotives. The primary goal in the development of the software was conformity to EN50128, the secondary goal is a cost-efficient process without sacrificing safety. To achieve these goals our method is based on formal techniques, but also designed to be easily applicable in our context (pragmatics). Central to our method are functional trees as a design specification mechanism. The outcome of employing this method was the successful certification of the locomotive G6 without any software-related problems.

## 1 Introduction

As for other means of transportation, the use of computer-based control systems in rail vehicles increased significantly over the last years and is still growing. Due to the fact that by rail heavy weights are moved with high velocities, faults in the control systems' software can have catastrophic consequences for human life and material goods. This potential risk leads to safety and certification requirements that control software for rail vehicles has to fulfill. At the same time manufacturers of rail vehicles have to develop software with limited resources and budgets at high quality and within tight project schedules.

At Vossloh Locomotives we therefore focus on the most time and resource consuming activities of the software development process. These are also the activities, where most faults occur: the specification, verification, implementation and validation of the software modules and their algorithms and the documentation of these activities.

The contribution of this paper is the presentation of a formal and pragmatic method to engineer software components of safety-critical systems, together with an industrial evaluation of this method.

In Section 2, the project context for engineering and certifying the locomotive G6 Vehicle Control Unit, is briefly introduced. Section 3 presents the employed method for developing the software modules, which is based on Binary Decision Diagrams. The guiding principle of this approach is simplicity, both for the engineers and for the certification process. So far, the method is applied manually without dedicated tool support. Such tool support is subject to future work, as will be discussed in Section 4 and indicated in the concluding Section 5. However, the presented method was already successfully employed for developing and certifying the VCU of the locomotive G6 [VL110].

## 2 Project Context

The development of software for safety-critical systems for rail vehicles starts with the definition of the control architecture.

An example control architecture is displayed in Figure 1. The components of this architecture are the Vehicle Control Unit (VCU), a Drive Control Unit (DCU) and an I/O control module. The DCU is connected with high-voltage power lines to the 3-phase traction motors. The I/O module provides an interface to additional devices, in our example a brake lever. The architectural elements VCU, DCU and I/O module are connected via a data communication bus.

For this paper the development of the safety-critical software for the VCU of the Vossloh shunting locomotive G6 [VL110] is presented. This includes a discussion of legal certification requirements and the design to achieve these requirements.

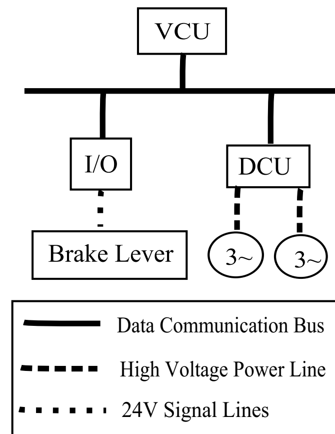


Figure 1: Typical Rail Vehicle Control Architecture

**Scope:** The scope of the following considerations will be a signal processing system that receives input signals from its environment and provides output signals which cause a change in the environment of the signal processing system. Out of our method's scope are the concrete signal sources and sinks of the signal processing system; thus, we abstract from the concrete signals. Here, it is only relevant that the behavior of this signal processing system may cause a safety risk.

**Legal Certification Requirements:** Whenever the use of a technology, such as Nuclear Power Plants, Railroads, or Airplanes, implies a potential risk to other legal assets, the use of these technologies should be regulated by law. For software in the European railroad industry, the norm EN50128 [CEN09] is relevant, which requires a software development process based on the V-Model [RHB+07]. In Germany the software is certified for use in rail verhicles by the Eisenbahnbundesamt (EBA) based upon an expert's report of an assessment agency, e.g. TüV. Subject to the assessment are the process planning, the suitability of methods and tools and the documentation of the activities.

**Documentation effort:** The V-Model software development process is partitioned into activities for requirements engineering, software architecture and design, and software module implementation. For these activities, Table 1 lists some quantities of the documentation effort for certifying the locomotive G6 to fulfill the documentation requirements

| Nr. | Phase | Effort | Functional Test Cases |
|---|---|---|---|
| 1 | Software Requirements | 4 documents 1000 pages | 1300 |
| 2 | Architecture and Design | 6 documents 1500 pages | none |
| 3 | Software Modules | 270 Modules 1350 documents 60000 pages | 22000 |

Table 1: Documentation effort – Locomotive Vossloh G6

**Focus on the software module documentation activities:** The documentation of the software modules was identified at Vossloh Locomotives as the most resource consuming activity of a software development process to meet the requirements of EN50128.
The focus in the whole software development process for a rail vehicle is therefore put on a time and cost effective approach to describe the software modules fulfilling the requirements of EN50128 and covering about 22000 functional points.

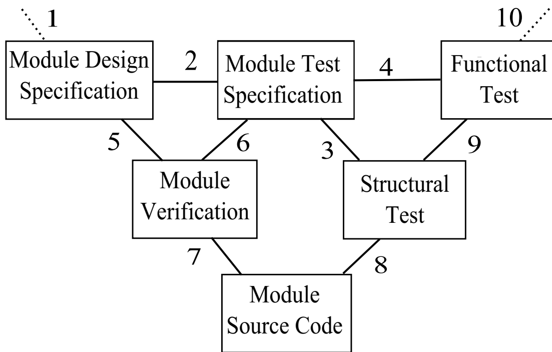


Figure 2: Module Development according to V-Modell

The development activities for software modules are shown in Figure 2. First activity is the specification of the interfaces and the functionalities (1) in a module design specification. All functional requirements have to be tested by procedures (2) defined in the module test specification.
The test procedures are divided into structural tests (white box) (3) and functional tests (black box) (4). A first verification step asserts whether the functional requirements are met (5) and testable by the defined procedure (6). The verification is documented in a module verification report. Next the source code is implemented (7). The source code verification is based on the structural sests (8) from the module test specification (3). After compilation, the executable code is validated (9) according to the functional tests from the module test specification (4). Finally, the module test report asserts the correct development according to EN50128 (10). Five documents are created per module.
The software development strategy follows a product line architecture [PBvdL05]. The software module development is part of domain engineering. Realizing a required function of a concrete product is part of application engineering. The software engineer specifies an architecture, where high-level concrete application requirements are mapped to module requirements. This allows the re-use of the modules in multiple software projects.

# 3 Software Module Development

This section describes the software module implementation method as it has been introduced and performed for the development of the Vehicle Control Unit for the Vossloh G6 locomotive. The method is based on an extension of Binary Decision Diagrams [Ake78] and Binary Functions [Bry86]. We present the formal basis for the extension of these ideas to describe functions that use discrete and continuous signals (Subsection 3.1), the graph-based representation (Subsection 3.2), the employed development process for an example (Subsection 3.3), and the compliance with high-level requirements (Subsection 3.4).

## 3.1 Formal Basis

A VCU is a part of a signal-processing system. We formally specify the various types of signals by means of set theory. For our method, the *uniqueness* and *completeness* of the defined sets is an important property, i.e. we define disjoint partitions of signal value sets.

In the following, $n$ is defined as the number of valid values of a signal. A signal (such as a brake lever) is represented by the set of possible signal values (such as the positions of a brake lever).

**Discrete Signals:** Discrete Signals are used to represent well-defined states of the environment. Signals are represented as integers. An example can be the position of a Brake System Control Lever as shown in Table 2 with $n = 3$ valid positions.

| Signal value | Code | Command |
|---|---|---|
| 1 | $C_1 = \{1\}$ | Brake Cylinder Pressure Increase |
| 2 | $C_2 = \{2\}$ | Brake Cylinder Pressure Constant |
| 3 | $C_3 = \{3\}$ | Brake Cylinder Pressure Decrease |
| all others | $C_\Omega = \mathbb{Z}\backslash\{1, 2, 3\}$ | Brake Cylinder Full Pressure |

Table 2: Example for values of discrete signals

*Definition of a Discrete Signal:*
Any discrete signal value is an element of one of the following disjoint subsets of $\mathbb{Z}$:

$$C_1, C_2, ..., C_n, C_\Omega \subseteq \mathbb{Z}$$

The set $C_\Omega$ represents the “unknown values.”

*Completeness of a Discrete Signal:*
The completeness of a discrete signal is given if for its subsets the following holds:

$$\bigcup_{i=1}^{n} C_i \cup C_\Omega = \mathbb{Z}$$

*Uniqueness of a Discrete Signal:*
The uniqueness of a discrete signal is given if for its subsets the following holds:

$$\forall i, j \in \mathbb{N} | i \leq n \wedge j \leq n \wedge i \neq j \Rightarrow C_i \cap C_j = \{\}$$

$$\forall i \in \mathbb{N} | i \leq n \Rightarrow C_i \cap C_\Omega = \{\}$$

**Continuous Signals:** Continuous Signals are used to represent physical values such as voltages, temperatures etc. and are represented by real numbers.

Continuous signals are discretized into ranges. With this approach the same mechanisms as for discrete signals can be used. An example can be the temperature of an engine coolant as shown in Table 3.

| Value | Range | Description |
|---|---|---|
| $< 10°C$ | $R_{LO}$ | Engine Under-Temperature |
| $10°C \geq T < 70°C$ | $R_1$ | Engine Cold |
| $70°C \geq T < 110°C$ | $R_2$ | Engine Nominal Temperature |
| $> 110°C$ | $R_{HI}$ | Engine Over-Temperature |

Table 3: Example for values of continuous signals

*Definition of a Continuous Signal:*
Any continuous signal value is an element of one of the following disjoint subsets of $\mathbb{R}$:

$$R_{LO}, R_1, R_2, ..., R_n, R_{HI} \subseteq \mathbb{R}$$

The set $R_{LO}$ represents the "underrun range" and $R_{HI}$ represents the "overrun range."

*Completeness of a Continuous Signal:*
The completeness of a continuous signal is given if for its subsets the following holds:

$$R_{LO} \cup \bigcup_{i=1}^{n} R_i \cup R_{HI} = \mathbb{R}$$

*Uniqueness of a Continuous Signal:*
The uniqueness of a continuous signal is given if for its subsets the following holds:

$$\forall i, j \in \mathbb{N} | i \leq n \wedge j \leq n \wedge i \neq j) \Rightarrow R_i \cap R_j = \{\}$$

$$\forall i \in \mathbb{N} | i \leq n \Rightarrow R_i \cap (R_{HI} \cup R_{LO}) = \{\}$$

$$R_{HI} \cap R_{LO} = \{\}$$

**Binary Signals** Binary signals are used to represent YES/NO decisions. They are expressed by a single bit. The representation as a single bit is important as it excludes any wrong values. The completeness and uniqueness of binary signals are, thus, obvious.

*Definition of a Binary Signal:*
Any binary signal value is an element of a set with exactly two different elements. Example: {0,1} or {TRUE,FALSE}.

### 3.2 Graph-Based Representation

Through the formal definition of signals we obtain a basis for specifying the functions by means of so-called *functional trees*. We employ three basic patterns as shown in Figure 3 which we can be used for synthesizing complete functions represented as functional trees.

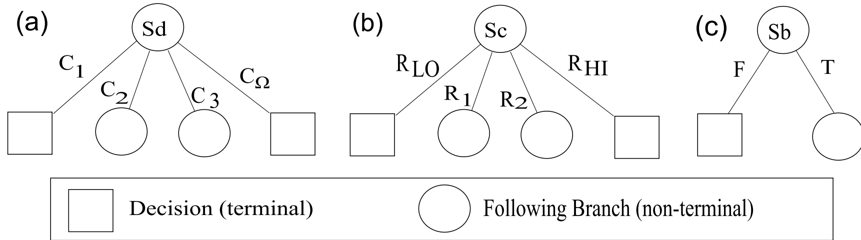


Figure 3: Basic elements for our functional specification

The patterns describe whether a decision for a result is reached or an additional, following branch is taken:

- Discrete Signals (Figure 3a): Results ($C_1$,$C_\Omega$) and additional branches ($C_2$,$C_3$)
- Continuous Signals (Figure 3b): Results ($R_{HI}$,$R_{LO}$) and additional branches ($R_1$,$R_2$)
- Binary Signals (Figure 3c): Results (F) and additional branches (T)

### 3.3 Module development process activities

In the following, we discuss the module development process activities by means of an example, the brake control.

| Signal Name | Type | Description | Range/Code | Order |
|---|---|---|---|---|
| Sd | Discrete | Brake Lever Position | $C_1$: Apply Brake<br>$C_2$: Constant Brake<br>$C_3$: Release Brake<br>$C_\Omega$: Unknown Position | 1 |
| Sb1 | Binary | Driver Vital Signal | T: Driver vital<br>F: Driver non-vital | 2 |
| Sb2 | Binary | Pressure Reservoir Switch | T: Reservoir Air available<br>F: Reservoir Air exhausted | 3 |

Table 4: Example functional signals

**Construction of the functional tree (design specification):** To specify a functional tree, we use the signals in Table 4. This is an example for the computation of a brake system action.

The functional tree is constructed by joining the specification patterns in Figure 3 according to the order of the signals in Table 4. Essential for the signal order is not the specific order itself, but the fact that there exists an order. By recursively connecting following specification patterns to the branches of a previous specification patterns we obtain a functional tree. The construction of the functional tree terminates if all paths end in a leaf (terminal) that represents the result of the corresponding function.

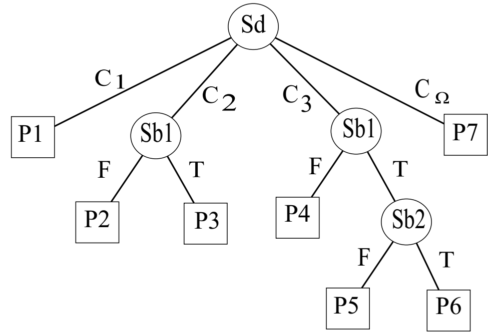


Figure 4: Functional tree for the functional signals in Table 4.

The functional tree in Figure 4 is equivalent to a logical function in disjunctive normal form (DNF) where one path in the functional tree corresponds to one logical AND connected element in a DNF functional equation.

Specification patterns are connected until all paths end in a decision. The decisions are the result of the computation that is represented as a path through the functional tree. This activity complies to step (1) in Figure 2. These trees can be expressed graphically as well as in a table. One specification item is a path through the whole functional tree. As we can see in this example, the Table 5 contains 7 possible paths through the whole functional tree while the approach to cover all possible combinations would result in 16 possibilities (Sd:4 x Sb1:2 x Sb2:2).

**Specification of Tests:** The specification of tests is divided into structural tests and functional tests. It complies to step (2) in Figure 2. After structuring the design into a functional tree, we have to verify that the function's code is structured accordingly.

The basis for the source code verification is shown in Table 5. The verification strategy will be to find an execution path through the source code that corresponds to the path in the functional tree (Design Specification). Therefore we have the same number of verification points as we have identified paths. The statement of the verification points complies to step (3) in Figure 2.

| Path ID | Sd | Sb1 | Sb2 | Result |
|---|---|---|---|---|
| P1 | $C_1$ | N/A | N/A | Apply Brake (x=0) |
| P2 | $C_2$ | F | N/A | Apply Brake (x=0) |
| P3 | $C_2$ | T | N/A | Constant Brake (x=1) |
| P4 | $C_3$ | F | N/A | Apply Brake (x=0) |
| P5 | $C_3$ | T | F | Apply Brake (x=0) |
| P6 | $C_3$ | T | T | Release Brake (x=2) |
| P7 | $C_\Omega$ | N/A | N/A | Apply Brake (x=0) |

Table 5: Functional table

Listing 1: Source code example

```
1  Trace = 0;
2  if(Sd == 1)
3     Trace |= 0x1;
4     x = 0;}                  /* P1 */
5  else if(Sd == 2)
6     Trace |= 0x2;
7     if(Sb1 == TRUE)
8        Trace |= 0x10;
9        x = 1;                /* P3 */
10    else
11       x = 0;                /* P2 */
12 else if(Sd == 3)
13    Trace |= 0x4;
14    if(Sb1 == TRUE)
15       Trace |= 0x10;
16       if(Sb2 == TRUE)
17          Trace |= 0x20;
18          x = 2;              /* P6 */
19       else
20          x = 0;              /* P5 */
21    else
22       x = 0;                 /* P4 */
23 else
24    x = 0;                    /* P7 */
```

After defining the source code verification points, we can define a test case for each path by just setting the input values as specified and performing a test for the specified result. The selection of the test cases complies to step (4) in Figure 2.

**Verification of Design and Test Coverage:** The module verification proves that the functional design specification of the module corresponds to a unique and complete set of paths of a functional tree. This step is mandatory for the compliance with EN50128. It uses equivalence classes, which are a highly recommended verification technique in EN50128. Our selection of paths guarantees the uniqueness and completeness of the signal's value sets. Therefore the module verification can be done recursively by the proof of completeness of the tree at the leafs (for instance, Sb1 to P2 and P3, resp. Sb2 to P5 and P6 in Figure 4). Based upon this first step, the completeness can be proven on the next level (Sb1 with precondition $Sd = C_3$) and is finished if the root node is reached (Sd). This activity complies to step (5) in Figure 2. Another issue is the proof that the test cases cover the whole functionality. This is given by the 1-to-1 mapping of functional requirements to test cases. This activity complies to step (6) in Figure 2.

**Generation/Implementation of the Source Code:** The source code is written manually as a transformation of the functional tree into conditional execution paths. There is no conversion into any Boolean logic. This eliminates a source of faults and simplifies the task of programming significantly. This activity complies to step (7) in Figure 2. The code fragment in Listing 1 presents an example.

The `Trace` variable has been introduced to identify the execution path. This trace variable is computed by setting it to 0 at the beginning of the computation and setting one specific bit depending of the executed path.

**Verification of the Generated Source Code:** The source code verification is based upon the verification points selected in the test specification of the function.

This activity complies to step (8) in Figure 2. In our example we can map the verification points to the source code lines according to Table 6. The main purpose of the source code verification is, to prove that the execution paths of the source code have an equivalent structure as the paths through the functional tree. This proof is the base of the statement that the specified tests provide a complete condition and path coverage of the source code. The source code verification is a requirement of EN50128. This verification technique also uses equivalence classes, which are highly recommended by EN50128.

| Verification Point | Line | Result |
|---|---|---|
| P1 | 04 | OK |
| P2 | 11 | OK |
| P3 | 09 | OK |
| P4 | 22 | OK |
| P5 | 20 | OK |
| P6 | 18 | OK |
| P7 | 24 | OK |

Table 6: Source Code Verification Results

**Module Testing:** The module testing is performed by setting the input variables and executing the code. This activity complies to step (9) in Figure 2. After each computation step both the result and the trace information are available. This is shown for our example in Table 7.

By recording the trace, we not only validate the result of a test case but additionally assert that the result is achieved by executing the paths which represents the requirement for this particular test case. In our example the result x=0 appears several times, but can always be traced back to the line of code at which it was set. After completing the tests, we are able to make a final statement on the usability of the module in the software project that has to be developed according to EN50128. This complies to step (10) in Figure 2.

| Test Case | Tested Path | Result | hex Trace | binary Trace |
|---|---|---|---|---|
| Tc1 | P1 | 0 | 0x1 | 00 0001 |
| Tc2 | P2 | 0 | 0x2 | 01 0010 |
| Tc3 | P3 | 1 | 0x12 | 01 0010 |
| Tc4 | P4 | 0 | 0x4 | 00 0100 |
| Tc5 | P5 | 0 | 0x14 | 01 0100 |
| Tc6 | P6 | 2 | 0x34 | 11 0100 |
| Tc7 | P7 | 0 | 0x0 | 00 0000 |

Table 7: Module Test Results

| ID | Requirement Text | Module Paths |
|---|---|---|
| Rq1 | An invalid position of the Brake Command Lever shall always apply full pressure on the Brake Cylinders | P1 to P6: Valid Positions, not applicable<br>P2: Brake applied |
| Rq2 | The Brakes shall never be released with an inactive Driver Vital Signal | P1 to P4,P7: Brake applied<br>P5 and P6: Brake released when Driver Vital Sign active |

Table 8: Module Test Results

### 3.4 Compliance with High Level Requirements

The compliance of the module design to fulfill the functional software requirements is asserted by comparing the informally written high-level requirements of a software requirements specification to the formally written design elements of the module design specification. This is shown in Table 8.

## 4 Related Work

Advanced model checking techniques can be seen as related work [BBB$^+$04]. The idea behind model checking is to avoid having humans construct proofs. Many important programs, such as vehicle control units, have ongoing behavior and ideally run forever; they don't just start and stop. Temporal logic has been established as a way to describe and reason about these programs. Then, if a program can be specified in temporal logic, it can be realized as a finite state program. A model checker checks whether a finite state graph is a model of a temporal logic specification. A great challenge with model checking is the state-explosion problem, which means that the number of the states may go exponentially high with the number of components of the system. Techniques such as symbolic and bounded model checking achieved significant results to overcome the state-explosion problem [CES09]. Our approach is based on binary decision diagrams which are an enabling technology for model checking [BBB$^+$04, CES09]. So far, we do not employ advanced tools, because our primary motivation is to use the method both for implementation and certification. For automatic model checking, it would be necessary to translate the requirements texts of Table 8 into temporal logic formulas. We envision several areas for tool support as will be indicated in the following section.

A comparable project is the SACEM software [GH90] for train control of the RER Line A in Paris. It consists of 21000 line of ADA code while the Vossloh G6 locomotive has 22000 functional points. Comparability is limited by the fact that the software validation included the whole specification process in the B language, while our approach is restricted to the software modules. The effort for the SACEM project was 100 men years [WLBF09].

# 5 Conclusions and Outlook

Our presented method strives for both formality and pragmatics. The primary goal is to engineer safe vehicle control units with a method that facilitates building safe vehicle control units and that is approachable to both our engineers and the certification inspectors. Formality is achieved by the mathematical foundation as introduced in Section 3. Pragmatics is achieved by deliberately neglecting advanced tool support. The method is simple such that the certification inspectors can easily retrace the activities of our engineers.

Based upon certification requirements, the software controlling rail vehicles in Europe has to be developed according to EN50128. In the development process, the documentation of the software modules has been identified as the most cost and time consuming part. To standardize this process at Vossloh Locomotives, the specification of the functionality of modules has been based upon a formal definition of the terms completeness and uniqueness. The process employs a constructive model building, based upon the functional requirements to describe single computation steps. This allows to use a manual verification technique. This formal method fulfills the requirements of the EN50128 as follows:

- Specification and proof of uniqueness and completeness requirements
- Simple extraction of test cases with traceability to requirements
- Proof of complete test coverage of requirements and test cases
- Simple creation of readable and maintainable source code with traceability to the requirements
- Proof of structural correspondence of source code and requirements
- Proof of complete condition and path coverage of test cases

To achieve the compliance of the development process to EN50128, we avoid the transformation of functional requirements into Boolean logic. Instead, we employ functional trees. This eliminates potential sources of errors and significantly simplifies the coding process. Additionally full traceability of test results to requirements by execution tracing validates the correctness of the algorithms.

The practical use of this approach is already evaluated in the development of the Vossloh Class G6 shunting locomotive. The whole documentation of over 60000 pages has been generated manually with a three person team within less than two years. For further locomotive projects a re-use rate of over 70% of the software modules is expected, thus making the process long-term productive. The certification authorities have not requested any further extensions of the software documentation or changes of the software development process to certify the locomotive. The development of the software is considered as a critical issue in many other rail vehicle projects.

Future work will address tool support for routine tasks in our method. As discussed in Section 4, the use of model checking tools could become an option. Particularly, tool support for model-based and model-driven code instrumentation [BH09], model-driven

testing [BH08] and trace analysis [RvHG+08] is on our agenda. Another topic for future work addresses the scalability of our methods toward more complex systems. Appropriate tool support will be an important factor. One concrete idea is to define a domain-specific language for our functional trees and to generate the instrumented code automatically from this representation. A coupled transformation could generate test cases and input to some model checker from the same or from some extended representation.

## References


[Ake78] S.B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, 27:509–516, 1978.

[BBB+04] R. Buschermöhle, M. Brörkens, I. Brückner, W. Damm, W. Hasselbring, B. Josko, C. Schulte, and T. Wolf. Model Checking - Grundlagen und Praxiserfahrungen. *Informatik-Spektrum*, 27(2):146–158, April 2004.

[BH08] Stefan Bärisch and Wilhelm Hasselbring. Model-Driven Test Case Construction by Domain Experts. In *Proc. 1st Workshop on Model-based Testing in Practice (MoTiP 2008)*, pages 9–18, 2008.

[BH09] Marko Boskovic and Wilhelm Hasselbring. Model Driven Performance Measurement and Assessment with MoDePeMART. In *Proc. MODELS 2009*, volume 5795 of *LNCS*, pages 62–76. Springer-Verlag, 2009.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[CEN09] CENELEC. *EN50128 - Railway Applications: Software for Railway Control and Protection Systems*. CENELEC, 2009.

[CES09] Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis. Model checking: algorithmic verification and debugging. *Commun. ACM*, 52(11):74–84, 2009.

[GH90] G. Guiho and C. Hennebert. SACEM software validation. In *Proceedings of the 12th international conference on Software engineering*, ICSE '90, pages 186–191, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank van der Linden, editors. *Software Product Line Engineering*. Springer, Berlin Heidelberg New York, August 2005.

[RHB+07] A. Rausch, R. Höhn, M. Broy, K. Bergner, and S. Höppner. *Das V-Modell XT: Grundlagen, Methodik und Anwendungen*. dpunkt.verlag, Heidelberg, 2007.

[RvHG+08] Matthias Rohr, André van Hoorn, Simon Giesecke, Jasminka Matevska, and Wilhelm Hasselbring. Trace-Context Sensitive Performance Models from Monitoring Data of Software Systems. In *Proc. TIMERS 2008*, pages 37–44, 2008.

[VL110] Vossloh Locomotives GmbH, Kiel. *Diesel-hydraulische Lokomotive G6*, 2010. `http://www.vossloh-locomotives.com/cms/de/products_and_services/diesel-hydraulic_locomotives/g6/g6_1.html`.

[WLBF09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41:19:1–19:36, October 2009.