Chapter 17: Applications of formal methods, modelling and testing strategies for safe software development

Author(s):
*A. Fantechi (Univ. of Florence, alessandro.fantechi@unifi.it)
A. Ferrari (ISTI-CNR, alessio.ferrari@isti.cnr.it)
S. Gnesi (ISTI-CNR, stefania.gnesi@isti.cnr.it)
*(Include all authors' names, affiliations, and contact info. Indicate lead corresponding author with *)*

The challenges posed by the new scenarios of railway transportation (liberalization, distinction between infrastructure and operation, high speed, European interoperability, etc.) have a dramatic impact on the safety issues. This impact is counterbalanced by a growing adoption of innovative signalling equipment (most notable example is ERTMS/ETCS) and monitoring systems (such as on board and wayside diagnosis systems). Each one of these devices includes some software, which in the end makes up the major part of their design costs; the malleability of software is paramount for the innovation of solutions. On the other hand, it is notorious how software is often plagued by bugs that may threaten its correct functioning: how can the high safety standards assumed as normal practice in railway operation be compatible with such threats?

This chapter wants to briefly summarize the current answers to such a question. Although the question regards the whole software life cycle, we concentrate on those aspects that are peculiar of the development of safe railway-related software: in a sense, we consider that generic Software Engineering best practices are already known to the reader. In particular in Section 46.1 we introduce the safety guidelines in effect for software development in this domain, in Section 46.2 we introduce the foundations of software testing, in Section 46.3 we introduce formal methods, with their applications in the railway domain in Sect. 46.4, and in Sect. 46.5 we introduce model-based software development.

46.1 CENELEC EN50128 standard, principles and criteria for software development

In safety-critical domains software development is often subject to a certification process, that in general has the aim to verify that the products have been developed in conformity with specific software safety guidelines, domain-specific documents issued by national or international institutions.

The definition of these guidelines is a very long and complex process that needs to mediate between different stakeholder, such as:

- *manufacturers*, interested to keep development costs low, but also paying attention to safety of their products in order not to run the risk of losing clients because of accidents due to failures of their products;
- *certification bodies* that take the responsibility to certify that the product is safe with a reasonably high certainty, according to the running laws and norms, so tend to give less priority to production costs;
- *clients* (often themselves service providers to final users, as is the case of railway operators), who are interested to balance cost containment with safety assurance.

Due to the contrasting interests of the stakeholders, the guideline production is a slow process that goes through necessary tradeoffs. Typically, a new edition of guidelines takes about ten years: once issued, they become a reference standard for the particular domain until the next edition; also for this reason, the railway signalling sector has been historically reluctant to technological innovation compared to other domains, especially for those functions that have significant impact on the safety of the railway.

46.1.1 Process and product certification

A first distinction to be done when speaking of computerized systems (and in particular, of software) certification is between *process* and *product certification.* Process certification is aimed to guarantee that the production process has followed given guidelines and norms, that have been adopted in order to guarantee that the process can deliver products of an expected quality level. Instead, product certification wants to guarantee that a particular product has been designed and developed according to determined quality guidelines.

Safety certification focuses more on *product* certification, but also it usually requires that the development process does comply to some quality and maturity standards.

The CENELEC safety standards for the railway sector (EN 50126, EN 50128, EN 50129), have much in common with the IEC 61508 standard for embedded systems, but they reflect the peculiar railway safety culture, that has a history that goes back more than one century.

EN 50126 defines RAMS (Reliability, Availability, Maintainability and Safety) concepts in relation to railway signalling. EN 50129 gives guidelines for the design of safety in hardware, while EN 50128 gives the guidelines for software production. Notice that EN 50128, as well the other ones of the family, have been issued for railway signalling systems. Although their use has spread beyond, to a full range of railway applications, new specific standards are emerging, such as prEN50657 intended for software on board of rolling stock.

46.1.2 Software development Cycle

EN 50128 (CENELEC 2011) does not mandate any specific software development method, but rather it describes the generic phases that the software development process should follow, and the attached documentation. One of the examples that are given about a possible software life cycle is the so called V-model (Fig. 1), where every design phase (left branch of the V)

corresponds to a verification phase (right branch of the V). The dotted arrows represent the relation between the tests carried on in the right branch and the artefacts produced in each phase of the left branch.

Place/Insert Figure 17.1 Here

Figure 1. The V-model for the software life-cycle of safety-critical railway systems

The focus on verification and testing activities, typical of the safety guidelines, is well represented by the V-model, that graphically makes evident that verification and testing costs are comparable with the costs of design and development, and that the former activities need to be prepared during the latter ones.

Indeed, EN50128 does very little guidance on technical choices, e.g. how to develop a suitable software architecture for the composition of separate software components (apart from a specific section dedicated to the possible usage of Object-Oriented Design), but is more concerned on which impact the different usual software engineering techniques can have on the safety of produced software. For this reason we will concentrate in the following section on those techniques that more directly can help to avoid the presence of bugs in produced software.

It is also worth noticing that EN50128 requires that verification and testing activities are carried out independently, and accurately defines the different roles involved in such activities, according to the Safety Integrity Level of the software component under development.

46.1.3 Safety Integrity level

The *software Safety Integrity Level* (SIL) is defined by EN 50128 as an attribute of a software

component that indicates its required robustness degree in terms of protection w.r.t. software faults. SIL has a value between 0 and 4: the higher the SIL, the higher should be the assurance that the software is free from faults. The SIL is assigned to components by a system *safety assessment* and *SIL apportionment* process on the basis of the impact on safety a failure of the component can have (see Chapter 7). Hence, the most serious consequences a failure of a component can have, the highest the SIL: SIL 0 is the level of a component with no effect on safety, SIL 4 is the level of a highly safety critical component1.

The assignment of the SIL level to a software component implies that it should be developed and verified using specific techniques that are considered suitable for that level. Although the techniques for software development in EN 50128 are relevant to cope with all types of software errors, special emphasis is on safety relevant errors. EN 50128 lists the techniques in a series of tables related to the software development phases, classified on the strength of suggestion of their usage, graduated along the SIL, as:

M = Mandatory

HR = Highly Recommended (which means that if not used, it should be justified in a proper document)

R = Recommended

- = No indication in favour or against

NR = Not Recommended (which means that its usage should be justified in a proper document)

Every entry of the table, that is, every listed technique, gives a reference, either to a description

[1] Besides SIL 0, there is software without any relevance for safety, which is not in the scope of the EN50128 standard.

text or to a sub-table which details the technique. Many tables include also notes that give indications on the recommended combination of techniques. We will give examples of recommendations in the following sections.

It is worth noticing that, besides software SIL levels, the EN50128 standard provides a classification also for the tools that are used along the development of the software, including tools for testing and formal verification. According to the norm, tools of class T1 do not generate output that can directly or indirectly contribute to the executable code (including data) of the software; tools of class T2 support test or verification, and errors in the tools can fail to reveal defects in the software or in the design, but cannot directly generate errors in the executable software; tools of class T3 generates outputs that can directly or indirectly contribute to the executable code (including data) of the safety related system. For each class, the norm lists the evidence that shall be provided about the actual role of each tool in the process, and about its validation.

46.2. The testing of software

Testing activities consist in the systematic research of faults in the software. In the railway safety-critical domain, testing is fundamental to ensure system safety, and its cost is comparable to the cost of the actual software coding. In this chapter, we will first give some preliminary definitions that are useful to understand the remainder of the sections, and then we will discuss the different types of testing activities, at different degrees of granularity (*unit testing*, *integration testing*, *system testing*) and in different development phases (*regression testing*, *mutation testing*), that are normally carried out in the railway domain. We give here a brief account on the foundations of testing, which is actually a separate discipline inside software

engineering, with many reference textbooks (see, e.g., Myers et al. 2011).

46.2.1 Preliminary Definitions

A program *P* can be in principle represented as a function $P: D \rightarrow R$, in which *D* is the space of the input data, and R is the space of the results. Correctness of a program could be then defined as a Boolean function $ok(P, d)$, applied on the program *P*, with the input data *d*. The function returns *true* if *P* produces the expected value for the input *d*, and *false* otherwise.

A *test T* (also called *test suite*, or *test set*) is a subset of the input data *D*. Furthermore, any $t \in T$ is a *test case*. A program *P* is *correct* for a test *T*, if for each $t \in T$, we have $ok(P, t) = true$.

A test *T* is considered *passed* if it does not reveal any fault - i.e., if the program is correct for the test *T* - and it is considered *failed* otherwise.

A *selection criterion C* for a program *P* is a set of predicates on *D*. We say that a test *T* is *selected* if the following conditions hold:

- $\forall t \in T \exists c \in C: c(t) = true$. This means that each test case satisfy at least one of the predicates of the selection criterion.
- $\forall c \in C \exists t \in T: c(t) = true$. This means that each predicate selects at least one test case.

An *exhaustive test* is given by a criterion that selects all possible input values *(T = D).* A passed exhaustive test implies that the program is correct for all input values. Since it is usual impossible to exhaustively test a program, the selection of a test has the aim to approximate exhaustivity with the lowest possible cost (that is, number of test cases). Indeed, it can be demonstrated that the testing of a program can reveal faults, but cannot prove their absence.

Notice also that the above definition assume that *ok(P, d)* is known, which means that for each test case the requirements tell the expected outcome of the execution: this is the meaning of the

horizontal links between the two sides of the V-model in Fig. 1.

Given these minimal definitions, we can start discussing the different types of tests that are normally used in the railway domain.

46.2.2 Unit Testing

Unit testing aims to verify whether a code unit (i.e., a function, a module, or a class) is correct with respect to its expected behaviour. Depending on the selection criteria adopted to define the test cases, we distinguish among *functional testing*, *structural testing* and *statistical testing*.

Functional Testing

Functional testing, also known as *black box testing,* or *requirement-based testing,* is performed on the basis of the functional requirements that determine which features are implemented by the unit under test. Functional testing does not look at the code of the unit, but only at its input/output behaviour. Commonly used criteria to select the tests are:

- *Equivalence Class Partitioning (ECPT):* the input domain of the unit is partitioned into equivalence classes, with the hypothesis that one test case for each class represents all the values for the same class.
- *Boundary Value Analysis (BVAN):* test cases are selected based on the boundary values of the equivalence classes. This enables to check typical programming errors, in which, e.g., a *less or equal* condition is erroneously replaced with a *less* condition.
- *Test Case by Error Guessing (TCEG):* test cases are selected by domain experts, based on their intuition, experience on the code and on the application domain.

Structural Testing and Coverage Criteria

With structural testing, the selection criterion is derived directly from the structure of the code unit, and in particular from its *flow graph*. The selected tests are those that exercise all the *structures* of the program. Hence, one must choose the reference structure (i.e., statement, branch, basic condition, compound condition, path), and a measure shall be used to indicate if all structures are exercised: this measure is called *coverage*.

- *Statement coverage* is pursued by choosing the tests according to their ability to cover the code statements, that is, the nodes of the flow graph. The statement coverage value is calculated as the ratio between the number of statements executed and the total number of statements. A test is selected by this criterion if its coverage value is equal to 1 (100%). This ensures that all statements are executed at least once.
- *Branch coverage* (or *decision coverage*) requires that all branches of the flow graph are executed, i.e., each branch belongs to at least one of the paths exercised by the test. The branch coverage value is calculated as the ratio between the number of branches executed and the total number of branches, and a test is selected if its coverage value is equal to 1. It is also referred as *decision coverage* since it exercises all the true/false outcomes of conditional statements, which are the sources of the branches in the flow graph.
- *Basic condition coverage* requires that all the basic Boolean conditions included in conditional statements are exercised, which means that for all the basic conditions both true and false outcomes are exercised. This does not guarantee that branch coverage equals 1, since some *combinations* of basic condition values might not be exercised, and some branches might not be executed.
- *Compound condition coverage* requires that all possible combinations of Boolean values

obtained from basic conditions are exercised at least once. This implies 2^n test cases, where *n* is the number of basic conditions.

- *Modified condition decision coverage (MCDC)* considers the Boolean values obtained from basic conditions in relation to their context in Boolean expressions. Boolean expressions are all those structures that fall in the syntactic category of Boolean expressions. This coverage is calculated as the ratio between the covered Boolean expressions, and the total number of Boolean expressions. Here, it is useful to consider an example:

```
If ((x > 0)                    cond1
       && (y < -2              cond2
       ||  y == 0              cond3
       ||  y > 2)              cond4
   )
```

Here, we have five Boolean expressions, i.e., four basic conditions, and the overall Boolean expression. The MCDC criterion selects the test cases shown in Table 1.

Table 1 Selected test cases according to the MCDC criterion

Test case	**cond1**	**cond2**	**cond3**	**cond4**	**Result**
1	F				F
2	T	T			T
3	T	F	T		T
4	T	F	F	T	T
5	T	F	F	F	F

Note that MCDC implies branch coverage, but with linear cost, instead of the exponential cost required by *compound condition coverage*.

Place/Insert Figure 17.2 Here

Figure 2. Relations between the different coverage criteria

- *Path coverage* requires that all paths of the program are exercised by the test. It is measured as the ratio between the number of paths exercised and the total number of paths. The total number of paths exponentially grows with the number of decisions that are independent and not nested, but such number becomes unbounded in case of cycles. Hence, when cycles are involved, the path coverage criterion considers a *finite* number of cycle executions (i.e., iterations). In this case, we speak about *k-coverage*, in which *k* is the number of iterations considered. Normally, *k*=1, which means that two test cases are defined, one in which no iteration is performed (the decision is false), and one in which one iteration is performed. Another way to reduce the exponential number of paths to be considered, is to evaluate the so-called *McCabe number*, defined as the number of paths in the flow graph that are linearly independent, which is equal to the number of decisions in the code, plus one. In this case, a test is selected if the number of paths that are linearly independent and that are exercised by the test is equal to the *McCabe number*. Sometimes, paths can be *unfeasible*, in the sense that they cannot be exercised by any input. Unfeasible paths are common in railway safety critical systems, due to the presence of defensive programming structures, which allow to cope with hardware or software failures. Due to the presence of unfeasible paths, path coverage might never be satisfied by a test. Hence, when evaluating path coverage, it is reasonable to limit it to the

feasible paths, and to evaluate the nature of the unfeasible paths through code inspection. Alternatively, one may consider error seeding in the code to force the software to execute these unfeasible paths.

Fig. 2 shows the relations between the different coverage criteria. The criteria are partially ordered. The criteria at the bottom of the figure are those that are weaker, but also less expensive. At the top of the figure, appear the criteria that are stronger, but also more expensive. Other criteria have been proposed in the literature on testing, we have limited ourselves to the most common ones.

Statistical Testing

While in functional and structural tests the test data are provided by a deterministic criterion, in statistical tests instead they are random, and can be based on the generation of pseudo-random test data according to an expected distribution of the input data to the program. Note that in both cases of non-statistical test (functional and structural), the selection criteria are deterministic and define *ranges* of values for the test cases. The test data may be chosen randomly in these ranges: in practice this corresponds to combine the statistical test with the earlier ones. A common way to conduct the testing is to first apply a functional test, or a statistical test, or a combination of the two, and then to measure the coverage according to the desired coverage criterion. If the resulting coverage is considered sufficient for testing purposes, the unit test ends, otherwise new tests are defined to increase the coverage.

Performing Unit Tests

To practically perform unit tests, one should define:

- *Driver:* module that includes the function to invoke the unit under test, passing

previously defined the test cases;

- *Stub:* dummy module that presents the same interface of a module invoked by the unit under test;
- *Oracle:* an entity (program or human user) that decides whether the test passed or failed;
- *Code Instrumentation:* inclusion in the code of instructions that allow to see if the execution of a test actually exercises the structures of the code, and that derive its coverage measure.

46.2.3. Integration Testing

After the coding phase, and after an adequate unit testing, it is appropriate to perform an *integration test* to verify the correctness of the overall program in order to be sure that there are no anomalies due to incorrect interactions between the various modules. Two methods are normally used: the *non-incremental* and the *incremental* approach.

The *non-incremental* approach or *big bang test*, assembles all the previously tested modules and performs the overall analysis of the system. The *incremental* approach instead consists in testing individual modules, and then connecting them to the caller or called modules, testing their composition, and so on, until the completion of the system. This approach does not require all modules to be already tested, as required by the non-incremental approach. In addition, it allows to locate interface anomalies more easily, and to exercise the module multiple times. With this approach, one may adopt a *top-down* strategy in incrementally assembling the components, which starts from the main module to gradually integrate the called modules, or a *bottom-up* strategy, which begins to integrate the units that provide basic functionalities, and terminates with the main as last integrated unit.

Coverage criteria have been defined also for integration testing. In particular, a commonly used criterion is the *procedure call coverage*, which indicates the amount of procedure calls exercised by an integration test.

46.2.4 System Testing

When the integration test has already exercised all the functionality of the entire system, it may still be necessary to test certain global properties that are not strictly related to individual system features but to the system as a whole. This is the role of *system testing*, which typically includes:

- *Stress/overload test:* checks that the system complies with the specifications and behaves correctly in overload conditions (e.g., high number of users, high number of connections with other systems).
- *Stability test:* checks the correct behaviour of the system even when it is used for long periods of time. For example, if dynamic memory allocation is used, this test checks that stack overflow errors occur only after an established period of time.
- *Robustness test:* the system is provided with unexpected or incorrect input data, and one checks its behaviour. A typical example is typing random input on the user interface to check whether the interface shows some anomalous behaviour.
- *Compatibility test:* verifies the correct behaviour of the software when connected to hardware devices to which it is expected to be compatible.
- *Interoperability test:* verifies the correct behaviour of the system when connected with other system of similar nature (e.g., other products provided by different vendors), once the communication protocol is established.
- *Safety test:* verifies the correct behaviour of the system in presence of violations of its

safety conditions, e.g., when a violation occurs the system switches to fail-safe mode. Specialized instruments are often used to perform system testing of railway control systems. In particular, *ad-hoc* hardware and software simulation tools are normally employed to recreate the environment in which the software will be actually executed.

46.2.5 Regression Testing

When a novel version of the product is released, it is necessary to repeat the testing to check whether the changes to the product have introduced faults that were not present before. One speaks in this case of *regression testing*, which aims to minimize the cost of testing using the tests performed on previous versions. This can be done by reusing the same drivers, stubs, oracles and by repeating the test cases of the previous versions.

Before performing regression testing, an impact analysis is explicitly required by EN50128. When the novel product is tested, one shall consider the changes introduced, and assess their impact in terms of testing. A small change should introduce minimal effort in terms of additional tests. However, one should consider that structural tests are often not incremental. Hence, even a small change could greatly impact on the coverage, and then one should define test cases that bring the coverage to the desired values. In case of even small changes to safety-critical railway systems, at least the safety tests have to be repeated completely.

46.2.6 Mutation Testing

Mutation testing helps to evaluate the ability of the performed tests to reveal potential errors. To this end, faults are intentionally introduced in the code. An ideal test should reveal those faults. If the previously defined tests do not reveal those faults, the test is not adequate, and more sophisticated tests have to be defined. The intentional introduction of faults is also used to

evaluate the robustness and safety of the software, and, in this case, it is referred as *fault injection* or *fault based testing*.

46.2.7 Testing according to EN 50128

We have already seen the importance of testing in the EN 50128 standard, exemplified by the V-model. Recommendation for the mentioned testing techniques varies according to the SIL:

- Functional/Black-box Testing is considered Mandatory by EN 50128 for software components at SIL3/SIL4, and Highly Recommended for software at SIL0/SIl1/SIL2. In particular, from SIL1 to SIL4, ECPT and BVAN are Highly Recommended.
- Statement coverage is Highly Recommended from SIL1 to SIL4, while the mentioned finer coverage measures are moreover Highly Recommended for SIL3/SIL4.

46.3. Formal Methods for the development and verification of software

Testing cannot be used for definitely assuring the absence of bugs. Even if, in general, proving the absence of bugs is an undecidable problem, in many cases formal arguments can be used to demonstrate their absence. Nowadays, the necessity of formal methods as an essential step in the design process of industrial safety-critical systems is indeed widely recognized.

In its more general definition, the term *formal methods* encompasses all notations having a precise mathematical semantics, together with their associated analysis and development methods, that allow to describe and reason about the behaviour and functionality of a system in a formal manner, with the aim to produce an implementation of the system that is provably free from defects. The application of mathematical methods in the development and verification of software is very labor intensive, and thus expensive. Therefore, it is often not feasible to check

all the wanted properties of a complete computer program in detail. It is more cost effective to first determine what the crucial components of the software are. These parts can then be isolated and studied in detail by creating mathematical models of these sections and verifying them.

In order to reason about formal description, several different notations and techniques have been developed. We refer to (Garavel and Graf 2013) (Gnesi and Margaria 2013) (Woodcock et al. 2009) for more complete discussions over the different methods and their applications; in the following sections we give a brief introduction to the main different aspects and concepts that can be grouped under this term.

46.3.1. Formal Specification

The languages used for formal specifications are characterised by the ability to describe the notion of internal state of the target system, and by their focus on the description of how the operations of the system modify this state. The underlying foundations are in discrete mathematics, set theory, category theory, and logic.

The B method

The B method (Abrial 1996) targets software development from specification through refinement, down to implementation and automatic code generation, with formal verification at each refinement step: writing and refining a specification produces a series of *proof obligations* that need to be discharged by formal proofs. The B method is accompanied by support tools, such as tools for the derivation of proof obligations, theorem provers, and code generation tools.

The Z notation

The Z notation (Spivey 1989) is a formal specification language used for describing and modelling computing systems. Z is based on the standard mathematical notation used in

axiomatic set theory, lambda calculus, and first-order predicate logic. All expressions in Z notation are typed, thereby avoiding some of the paradoxes of naive set theory. Z contains a standardized catalog (called the *mathematical toolkit*) of commonly used mathematical functions and predicates. The Z notation has been at the origin of many other systems as for example Alloy2 (Jackson 2012) and its related tool Alloy Analyser, which adapts and extends Z to bring in fully automatic (but partial) analysis.

Automata-Based Modelling

In this case it is the concurrent behaviour of the system being specified that stands at the heart of the model. The main idea is to define how the system reacts to a set of stimuli or events. A state of the resulting transition system represents a particular configuration of the modelled system. This formalism, and the derived ones such as Statecharts (Harel 1987) and their dialects, are particularly adequate for the specification of reactive, concurrent, or communicating systems, and also protocols. They are however less appropriate to model systems where the sets of states and transitions are difficult to express.

Modelling Languages for Real-Time Systems

Extending the simple automata framework gives rise to several interesting formalisms for the specification of real-time systems. When dealing with such systems, the modelling language must be able to cope with the physical concept of time (or duration), since examples of real-time systems include control systems that react in dynamic environments. At this regard we can mention Lustre (Halbwachs et al. 1991) that is a (textual) synchronous dataflow language, and

[2] http://alloy.mit.edu/alloy/

SCADE[3] (Berry 2007), a complete modelling environment that provides a graphical notation based on Lustre. Both provide a notation for expressing synchronous concurrency based on data flow.

Other graphical formalisms have proved to be suitable for the modelling of real-time systems. One of the most popular is based on networks of timed automata (Alur and Dill 1994). Basically, timed automata extend classic automata with clock variables (that evolve continuously but can only be compared with discrete values), communication channels, and guarded transitions.

46.3.2. Formal verification

To ensure a certain behaviour for a specification, it is essential to obtain a rigorous demonstration. Rather than simply constructing specifications and models one is interested in proving properties about them.

Model Checking

A formal verification technique that has recently acquired popularity also in industrial applications is Model Checking (Clarke et al 1999), an automated technique that, given a finite-state model of a system and a property stated in some appropriate logical formalism (such as temporal logic), checks the validity of this property on the model. Several temporal logics have been defined for expressing interesting properties. A temporal logic is an extension of the classical propositional logic in which the interpretation structure is made of a succession of states at different time instants. An example is the popular CTL (Computation Tree Logic), a *branching time* temporal logic, whose interpretation structure (also called *Kripke structure*) is the

[3] http://www.esterel-technologies.com

computation tree that encodes all the computations departing from the initial states.

Formal verification by means of model checking consists in verifying that a Kripke structure *M*, modelling the behaviour of a system, satisfies a temporal logic formula φ, expressing a desired property for *M.* A first simple algorithm to implement model checking works by labelling each state of *M* with the subformulae of φ that hold in that state, starting with the ones having length 0, that is with atomic propositions, then to subformulae of length 1, where a logic operator is used to connect atomic propositions, then to subformulae of length 2, and so on. This algorithm requires a navigation of the state space, and can be designed to show a linear complexity with respect to the number of states of *M.* One of the interesting features of model checking is that, when a formula is found not to be satisfied, the subformula labeling collected on the states can be used to provide a *counterexample,* that is, an execution path that leads to the violation of the property, thus helping the debugging of the model.

The simple model checking algorithm sketched above needs to explore the entire state space, incurring in the so called *exponential state space explosion*, since the state space often has a size exponential in the number of independent variables of the system.

Many techniques have been developed to attack this problem: among them, two approaches are the most prominent and most widely adopted. The first one is based on a symbolic encoding of the state space by means of boolean functions, compactly represented by Binary Decision Diagrams (BDD) (Bryant 1986). The second approach considers only a part of the state space that is sufficient to verify the formula, and within this approach we can distinguish local model checking and bounded model checking: the latter is particularly convenient since the problem of checking a formula over a finite depth computation tree can be encoded as a satisfiability

problem, and hence efficiently solved by current very powerful *SAT-solvers* (Biere et al. 1999). The availability of efficient model checking tools able to work on large state space sizes has favoured, from the second half of the nineties, their diffusion in industrial applications. The most popular model checkers of an academic origin are:

1. SMV4 (McMillan 1993), developed by the Carnegie Mellon University, for the CTL logic, based on a BDD representation of the state space;
2. NuSMV5 (Cimatti et al 2002), developed by Fondazione Bruno Kessler, a re-engineered version of SMV which includes a Bounded Model Checking engine;
3. SPIN6 (Holzmann 2003), a model checker for the linear temporal logic LTL, developed at Bell Labs, for which the PROMELA language for the model definition has been designed.

1 Software Model Checking

The first decade of model checking has seen its major applications in the hardware verification domain; meanwhile, applications to software have been made at system level, or at early software design. Later, applications within the model-based development (see Section 46.5) have instead considered models at a lower level of design, closer to implementation. But such an approach requires an established process, hence excluding software written by hand directly

4 http://www.cs.cmu.edu/~modelcheck/smv.html

5 http://nusmv.fbk.eu

6 http://spinroot.com/spin/whatispin.html

from requirements: indeed, software is often received from third parties, who do not disclose their development process. In such cases direct verification of code correctness should be conducted, especially when the code is safety-related: testing is the usual choice, but testing cannot guarantee exhaustiveness.

Direct application of model checking to code is however still a challenge, because the correspondence between a piece of code and a finite state model on which temporal logic formulae can be proved is not immediate: in many cases software has, at least theoretically, an infinite number of states, or at best, the state space is just huge.

Pioneering work on direct application of model checking to code (also known as Software Model Checking) has been made at NASA since the late nineties by adopting in the time two strategies: first, by translating code into the input language of an existing model checker – in particular, translating into PROMELA, the input language for SPIN. Second, by developing ad hoc model checkers that directly deal with programs as input, such as JavaPathFinder7 (Havelund and Pressburger 2000). In both cases, there is the need to extract a finite state abstract model from the code, with the aim of cutting the state space size to a manageable size: the development of advanced abstraction techniques has only recently allowed a large scale application of software model checking. JavaPathFinder has been used to verify software deployed on space probes. Some software model checkers, such as CBMC (CBMC), hide the formality to the user by providing ”built-in” default properties to be proven: absence of division by zero, safe usage of pointers, safe array bounds, etc. On this ground, such tools are in competition with tools based on

7 http://javapathfinder.sourceforge.net

Abstract Interpretation, discussed in the next section. It is likely that, in the next years, software model checking will gain a growing industrial acceptance also in the railway domain, due to its ability to prove the absence of typical software bugs, not only for proving safety properties, but also to guarantee correct behaviour of non safety related software. Applying model checkers directly to production safety-related software calls for their qualification at level T2.

Abstract Interpretation

Contrary to testing, that is a form of dynamic analysis focusing on checking functional and control flow properties of the code, static analysis aims at automatically verifying properties of the code without actually executing it. In the recent years, we have assisted to a raising spread of abstract interpretation, a particular static analysis technique. Abstract interpretation is based on the theoretical framework developed by Patrick and Radhia Cousot in the seventies (Cousot and Cousot 1977). However, due to the absence of effective analyses techniques and to the lack of sufficient computer power, only after twenty years software tools have been developed for supporting the industrial application of the technology (e.g., Astree[8] (Cousot et al 2005), Polyspace[9]). In this case the focus is mainly on the analysis of source code for runtime error detection, which means detecting variables overflow/underflow, division by zero, dereferencing of non-initialized pointers, out-of-bound array access and all those errors that, might them occur, bring to undefined behaviour of the program.

8 https://www.absint.com/products.htm

9 https://mathworks.com/products/polyspace/

Since the correctness of the source can be not decidable at the program level, the tools implementing abstract interpretation work on a conservative and sound approximation of the variable values in terms of intervals, and consider the state space of the program at this level of abstraction. The problem boils down to solve a system of equations that represent an over-approximate version of the program state space. Finding errors at this higher level of abstraction does not imply that the bug also holds in the real program. The presence of false positives after the analysis is actually the drawback of abstract interpretation that hampers the possibility of fully automating the process (Ferrari et al. 2013a). Uncertain failure states (i.e., statements for which the tool cannot decide whether there will be an error or not) have normally to be checked manually and several approaches have been put into practice to automatically reduce these false alarms.

Automated theorem proving

Another possibility for the automatic verification of systems properties is by Theorem Proving. Theorem Provers favour automation of deduction rather than expressiveness. Construction of proofs is automatic, once the proof engine has been adequately parameterised. Obviously these systems assume the decidability of at least a large fragment of the underlying theory. Theorem Provers are powerful tools, capable of solving difficult problems and are often used by domain experts in an interactive way. The interaction may be at a very detailed level, where the user guides the inferences made by the system, or at a much higher level where the user determines intermediate lemmas to be proved on the way to the proof of a conjecture.

The language in which the conjecture, hypotheses, and axioms (generically known as *formulae*) are written is a logic, often classical 1st order logic, but also non-classical logic or higher order

logic. These languages allow a precise formal statement of the necessary information, which can then be manipulated by a Theorem Prover.

The proofs produced by Theorem Provers describe how and why the conjecture follows from the axioms and hypotheses, in a manner that can be understood and agreed upon by domain experts and developers. There are many Theorem Provers systems readily available for use, the most popular are Coq[10] (Barras et al 1997), HOL[11] (Nipkow et al 2002), and PVS[12] (Owre 1992).

Software verification is an obvious and attractive goal for Theorem Proving technology. It has been used in various applications, including diagnosis and scheduling algorithms for fault tolerant architectures, and requirements specification for portions of safety critical systems.

46.4 Railway applications of formal methods and formal verification

Formal methods since thirty years have promised to be the solution for the safety certification headaches of railway software designers. The employment of very stable technology and the quest for the highest possible guarantees have been key aspects in the adoption of computer-controlled equipment in railway applications. EN50128 indicates Formal Methods as Highly Recommended for SIL3/4 and Recommended for SIL1/2 in the Software Requirement Specification phase, as well as in the Software Design phase. Formal proof is also HR for SIL3/4

10 https://coq.inria.fr

11 https://hol-theorem-prover.org

12 http://pvs.csl.sri.com

and R for SIL1/2 in the Verification phase, and HR for SIL3/4 for configuration data preparation. Formal proof, or verification, of safety is therefore seen as a necessity. Moreover, consolidated tools and techniques have been selected more frequently in actual applications. We mention just a few of them, with no claim of completeness. References (Flammini 2012) (Boulanger 2014) give more examples of railway applications.

46.4.1 Formal Specification

B method

The B method has been successfully applied to several railway signalling systems. The SACEM system for the control of a line of Paris RER (Da Silva et al. 1993) is the first acclaimed industrial application of B. B has been adopted for many later designs of similar systems by Matra (now absorbed by Siemens). One of the most striking applications has been the Paris automatic metro line 14. The paper (Behm et al. 1999) on this application of B reports that several errors were found and corrected during proof activities conducted at the specification and refinement stages. By contrast, no further bugs were detected by the various testing activities that followed the B development. The success of B has had a major impact in the sector of railway signalling by influencing the definition of the EN 50128 guidelines.

46.4.2 Formal verification

Model checking

The revised EN50128 marks in 2011 the first appearance of model checking as one of the recommended formal verification techniques in a norm regarding safety critical software.

Indeed, model checking is now a common mean to gain confidence in a design, especially

concerning the most critical parts. Safety verification of the logics of interlocking systems has been since many years an area of application of model checking by many research groups, see e.g., (Fantechi 2013), (Hansen et al. 2010), (James et al. 2014), (Vu et al. 2016), and is still an open-research area due to the difficulty of verification of large designs due to raising state space explosion problems that overcome the capacity of model checking tools.

Abstract Interpretation

Due to the recent diffusion of tools supporting static analysis by Abstract Interpretation, published stories about the application of this technology to the railway domain are rather limited, although static analysis is HR for SIL1 to SIL4 by EN50128. (Ferrari et al., 2013a) provides one of the few contributions in this sense, showing how abstract interpretation was proficiently used in conjunction with model-based testing, to reduce the cost of the testing activities of a railway signalling manufacturer by 70%.

46.5 Model-based Development

Although several successful experiences on the use of formal methods in railway systems have been documented in the literature, formal methods are still perceived as experimental techniques by railway practitioners. The reasons are manifold. On the one hand, handling model checkers and theorem provers requires specialised knowledge that is often beyond the competencies of railway engineers. On the other hand, the introduction of formal methods requires a radical restructuring of the development process of companies, and does not allow to easily reuse code and other process artefacts that were produced with the traditional process. In addition, available formal tools are designed to be used by experts, and rarely have engineering-friendly interfaces

that could make their use more intuitive for practitioners. Hence, while it has taken more than twenty years for consolidating the usage of formal methods in the development process of railway manufacturers, the *model-based* design paradigm has gained ground much faster. Modelling has indeed the same degree of recommendation of formal methods by EN50128. The defining principle of this approach is that the whole development shall be based on graphical model abstractions, from which an implementation can be manually or automatically derived. Tools supporting this technology allows to perform *simulations* and *tests* of the system models before the actual deployment: the objective is not different from the one of formal methods, that is detecting design defects before the actual implementation, but while formal methods are perceived as rigid and difficult, model-based design is regarded as closer to the needs of developers, that consider graphical simulation as more intuitive than formal verification. This trend has given increasing importance to tools such as MagicDraw13, IBM Rational Rhapsody14, SCADE, and the tool suite Matlab/Simulink/Stateflow15. MagicDraw and Rhapsody provide capabilities for designing high-level models according to the well-known UML language, and provide support for SysML, which is an extension of the UML language for *system engineering*.

Instead, SCADE and the tool suite Matlab/Simulink/Stateflow are focused on block-diagrams

13 http://www.nomagic.com/products/magicdraw.html

14 http://www-03.ibm.com/software/products/en/ratirhapfami

15 http://www.mathworks.com/products/simulink/

and on the *statecharts formalism* (Harel 1987), which is particularly suitable to represent the state-based behaviour of embedded systems used in railways. While the former tools are oriented to represent the high-level artefacts of the development process – i.e., requirements, architecture, design – the latter are more oriented to model the lower-level behaviour of systems.

Without going into the details of each tool, in this section we discuss the main capabilities offered by the model-based development paradigm, and their impact in the development of railway safety-critical systems. The capability we discuss are: *modelling*, *simulation*, *testing*, *code generation*.

46.5.1. Modelling

Regardless of the tool adopted for modelling, it is first crucial to identify the right level of abstraction at which models are represented. The level of abstraction depends on what is the purpose of the model: if the model is used to define the *architecture* of the railway system, with different high-level sub-systems interacting one with the other, the use of high-level languages such as SysML, or UML is more suitable; instead, if the model is used to define the *behaviour* of the system, and one wants to generate executable code from it, statecharts or UML state machines are more suitable for the goal. One key step in both cases is a clear definition of the interfaces between systems, and between software components. For models that are going to be used for code generation, it is preferable to have interfaces that are composed solely of input and output *variables*, and not function calls. This eases decoupling among model components, and facilitates their testing, since one does not have to create complex stubs for external function calls. Another aspect to consider when modelling for code generation is the need to constrain the modelling language. Indeed, in railway systems, the generated code has to abide to the strict

requirements of EN 50128, which are normally internally refined by the companies, and that forbid the use of some typical code features, such as global variables, pointers, and dynamic memory allocation. To ensure that the generated code abides to the guidelines, the modelling language has to be restricted to a *safe* subset. For example, for Stateflow, a safe subset to be employed in railway applications was defined by (Ferrari et al. 2013b).

46.5.2. Simulation

One of the main benefits of having graphical models is the possibility of animating the models, i.e., observing their dynamic behaviour both in terms of input/output variables, and in terms of internal states dynamics. In this way, models become *executable specifications*, and the modeller can observe the system behaviour before the system is actually deployed, by providing appropriate stubs for input and output that simulate the *environment* of the specification. The simulation feature is available in all the mentioned platforms, and is particularly useful in the debugging phase. Indeed, the modeller is able to *visualise* where, in the model, an error actually occurred, and which is the context of execution of the model when the error occurred (i.e., the other states currently active in the model). This contextualised visualisation is something that is not possible when debugging code, and becomes useful also to detect faults in the model when a test on the model fails.

46.5.3. Model-based Testing

As mentioned before in this chapter, testing is one of the key activities in the development of railway systems. When using a model-based paradigm tests are normally executed on models, and existing tools give the possibility to automatically generate tests based on coverage criteria. For statecharts, coverage criteria are normally state coverage and transition coverage (equivalent

to statement coverage and branch coverage for code testing). While automatic test case generation can allow to easily reaching fixed coverage criteria, it does not ensure that the system actually complies with its requirements. For this reason, functional tests are defined and performed by means of *ad-hoc* stubs that generate the appropriate input signals for the models.

If one wants to generate code from the models, one additional step is required: tests executed on the models have to be repeated on the generated code, and (a) the input/output behaviour of the code have to match with the behaviour of the models, and (b) the resulting coverage for a test have to match between models and code. In this way, one ensures that the generated code behaviour matches with the behaviour observed on models, and that the code generator introduces no additional behaviour. This step is often called *translation validation* (Conrad 2009), and was applied in the railway domain by (Ferrari et al. 2013a).

46.5.4. Code Generation

With code generation, the code that will be deployed in the microprocessor(s) of the system is automatically generated from the models. Some modelling tools – normally those that allow to model statecharts or UML state machines – enable to generate the complete source code, while others – normally those that allow to define the architecture and high-level design of the software – generate *skeletons* for the classes or functions of the code, which have to be manually completed by the developer. Even in case of complete generation, the developer normally has to manually define *glue* code, i.e., an adapter module, to attach the generated code to the drivers. It is worth remarking that, for example, the code generator from SCADE is qualified[16] for railway

[16] According to the *tool qualification* process of EN50128:2011 the T3 most severe class of qualification is required for code generators.

systems, which, in principle, implies that the models exhibit the same behaviour of the code. In case nonqualified code generators are used, one has to ensure model-to-code compliance by means of translation validation, as mentioned in the previous section.

46.6 Conclusions

This chapter has given just a brief account of techniques that can be used to produce safe software for railway systems. Looking again at the EN50128 standard, one can see that, for space limits, we have not cited many techniques that are also commonly used (e.g. defensive programming, diversity, etc..) and we have not addressed the important distinction between *generic* and *specific* applications nor *data preparation* techniques, both used when a railway signalling application has to be deployed for a specific track or line layout. But we think that reading this chapter is a good introduction to the issue of developing safe software.

References

Abrial, J.R. 1996. The B-Book. Cambridge, MA: Cambridge University Press.

Alur, R., and Dill, D. L. 1994. A theory of timed automata. *Theoretical computer science*, 126(2):183-235.

Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J. C., Gimenez, E., et al 1997. *The Coq proof assistant reference manual: Version 6.1, RT-203*. Le Chesnay, France: INRIA.

Behm, P., Benoit, P., Faivre, A., and Meynadier, J. M. 1999. METEOR: A successful application of B in a large project. In *Lecture Notes in Computer Science 1708*, ed J. M. Wing, J. Woodcock, and J. Davies, 369-387. Berlin Heidelberg: Springer Verlag.

Berry, G. 2007. SCADE: Synchronous design and validation of embedded control software. In *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, ed S. Ramesh, and P. Sampath, 19-33. Dordrecht, Netherlands: Springer Netherlands.

Biere, A., Cimatti, A., Clarke, E., and Zhu, Y. 1999. Symbolic model checking without BDDs. In *Lecture Notes in Computer Science 1579*, ed W. R. Cleaveland, 193-207. Berlin Heidelberg: Springer Verlag.

Boulanger, J.L. 2014. Formal Methods Applied to Industrial Complex Systems. Hoboken, NJ: John Wiley & Sons.

Bryant, R. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677-691.

Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., and Tacchella, A. 2002. Nusmv 2: An opensource tool for symbolic model checking. In *Lecture Notes in Computer Science 2404*, ed E. Brinksma, and K. G. Larsen, 359-364. Berlin Heidelberg: Springer Verlag.

Clarke, E. M., Grumberg, O., & Peled, D. (1999). Model checking. Cambridge, MA: MIT press.

Cousot, P., and Cousot, R. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ed R.

M. Graham, M. A. Harrison, and R. Sethi, 238-353. New York, NY: ACM Press.

Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X., 2005. The ASTRÉE analyzer. In *Lecture Notes in Computer Science* 3444, ed S. Sagiv, 21-30. Berlin Heidelberg: Springer Verlag.

Conrad, M. 2009. Testing-based translation validation of generated code in the context of IEC 61508. *Formal Methods in System Design*, 35(3):389-401.

CENELEC (European Committee for Electrotechnical Standardization), 2011. EN 50128:2011 – Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems.

Da Silva, C., Dehbonei, Y. and Mejia, F. 1993. Formal specification in the development of industrial applications: Subway speed control system. In *5th IFIP Conference on Formal Description Techniques for Distributed Systems and Communication Protocols*, ed M. Diaz, and R. Groz, 199–213. Amsterdam, The Netherlands: North-Holland Publishing Co.

Fantechi, A. 2013. Twenty-Five Years of Formal Methods and Railways: What Next? In *Lecture Notes in Computer Science 8368*, ed S. Counsell, and M. Núñez, 167-183. Berlin Heidelberg: Springer Verlag.

Ferrari, A., Magnani, G., Grasso, D., Fantechi, A., and Tempestini, M. 2013a. Adoption of model-based testing and abstract interpretation by a railway signalling manufacturer. *Adoption and Optimization of Embedded and Real-Time Communication Systems*, ed S. Virtanen, 126-144. Hershey, PA: IGI Global.

Ferrari, A., Fantechi, A., Magnani, G., Grasso, D., and Tempestini, M. 2013b. The metrô rio case

study. *Science of Computer Programming*, 78(7):828-842.

Flammini, F. 2012. Railway Safety, Reliability, and Security: Technologies and Systems Engineering. Hershey, PA: IGI Global.

Garavel, H., and Graf, S. 2012. Formal Methods for Safe and Secure Computer Systems. BSI Study 875. https://goo.gl/oIxlho (accessed October 28, 2016).

Gnesi, S., and Margaria, T. 2013. Formal Methods for Industrial Critical Systems: A Survey of Applications. Chichester: Wiley-IEEE Press.

Hansen, H. H., Ketema, J., Luttik, B., Mousavi, M. R., van de Pol, J., and dos Santos, O. M. 2010. Automated Verification of Executable UML Models. In *Lecture Notes in Computer Science 6957*, ed B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue, 225–250. Berlin Heidelberg: Springer Verlag.

Havelund, K., and Pressburger, T. 2000. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366-381.

Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. 1991. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305-1320.

Harel, D. 1987. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231-274.

Havelund, K., and Pressburger, T. 2000. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4): 366-381.

Holzmann, G. 2003. Spin model checker, the: primer and reference manual. Boston, MA: Addison-Wesley Professional.

James, P., Möller, F., Nguyen, H. N., Roggenbach, M., Schneider, S., Treharne, H., Trumble, M., and Williams, D. 2014. Verification of Scheme Plans Using CSP||B. In *Lecture Notes in Computer Science 8368*, ed S. Counsell, and M. Nùñez, 189–204. Berlin Heidelberg: Springer Verlag.

Jackson, D. 2012. Software Abstractions: Logic, Language and Analysis. Cambridge, MA: MIT Press.

McMillan, K. L. 1993. The SMV system. In *Symbolic Model Checking*, ed K. L. McMillan, 61-85. New York, NY: Springer US.

Myers, G. J., Sandler, C., and Badgett, T. 2011. The art of software testing. Hoboken, NJ: John Wiley & Sons.

Nipkow, T., Paulson, L. C., and Wenzel, M. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. New York, NY: Springer Science & Business Media.

Owre, S., Rushby, J. M., and Shankar, N. 1992. PVS: A prototype verification system. In *Lecture Notes in Computer Science 607*, ed D. Kapur, 748-752. Berlin Heidelberg: Springer Verlag.

Spivey, J. M.,1989. The Z Notation: A Reference Manual. Upper Saddle River, NJ: Prentice-Hall.

Vu, L. H., Haxthausen, A.E., and Peleska, J. 2016. Formal modelling and verification of interlocking systems featuring sequential release. Science of Computer Programming, 133: 91-115 (2017)

Woodcock, J., Larsen, P.G., Bicarregui, J., and Fitzgerald, J. 2009. Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4):19:1–19:36.