Daniel Schwencke, German Aerospace Center (DLR), Institute of Transportation Systems, Braunschweig, Germany

Test Case Generation for a Level Crossing Controller

Author

Daniel Schwencke graduated in Computer Science and received his PhD from the Technical University of Braunschweig. In 2011 he joined the DLR Institute of Transportation Systems as researcher, working in the field of railway safety, including system development and authorization processes as well as human reliability. Since 2015 he is part of the verification and validation department, conducting research on model based testing of signaling systems and test automation. He is involved in the European X2Rail-2 project.

German Aerospace Center (DLR), Institute of Transportation Systems, Lilienthalplatz 7, 38108 Braunschweig, Germany

+49 531 295 3416

daniel.schwencke@dlr.de

1. Introduction

Formal methods (FM) can be used for the precise specification, property-ensuring development and exhaustive property verification of systems. Thus they are especially suited for highly safety or mission critical applications. Railway signaling systems clearly belong to these applications, and there are indeed several industrial projects where FM have been successfully applied; especially to core interlocking and communication-based train control (CBTC) systems. But despite their potential, FM are not very wide-spread in the sector. Several studies [1, 2, 3] regarding their diffusion have been conducted. The main determinants for adoption that emerge from those studies are

- the maturity of available tools,
- the learnability of the tools (learning curve),
- the perceived benefits-of-use and perceived ease-of-use by engineers and professionals, and
- the compatibility with already existing tools or toolchains.

Also, the choice of a method and tool among the many different FM available can require high expertise. According to some of the studies, the cost of the potential tools to be used appears to not be an essential determinant.

Work Package 5 of the X2Rail-2 project seeks to foster the use of FM in railway signaling by providing an introduction and overview of formal methods [4] and demonstrating their use and benefit. For the latter, four different formal and one classical development methods are applied by different project partners to a level crossing (LX) controller specified by the Swedish railway infrastructure manager Trafikverket. This includes

- the refinement-based B method,
- model-based design with SCADE,
- configuration-based development with Prover iLock, and
- contract-based programming in SPARK

for formal development as well as the ladder logic-based Westrace system for the classical development. For all of these developments, the safety properties from the LX specification are planned to be formally verified afterwards using the High Level Language (HLL). Since that means proving them exhaustively, they are of less interest for testing.

However, there are further non-safety functional requirements in the specification which remain for testing. The extended abstract at hand reports on an automatic test case generation (TCG) approach of a test suite testing these requirements. In fact, this approach is based on formal methods as well, since the test case generator applies symbolic execution and theorem solving techniques: given a behavioral model of the system under test (SUT), the former method finds feasible paths through the model, while the latter completes the test case by determining suitable test data. This way, the test design task is partly automated, ensures a structural coverage of the model and the modeling process usually leads to a high test suite quality. The different LX controller implementations are tested as black box systems, each one with the same generated test cases. In order to simplify the integration of the different implementations with the test environment, a common test interface has been drawn up.

2. Overview of the Approach

2.1 Tooling

For the work presented here, the “Automatic Test Generation” (ATG) Add-On [5] of the UML/SysML tool Rational Rhapsody of IBM is used for the TCG. This, in turn, is based on the “TestConductor” Add-On [5]. Altogether Rhapsody and the two add-ons form a tightly integrated environment, which covers the whole model-based testing process starting from model creation (Rational Rhapsody), covering TCG (ATG Add-On) and stretching to test execution (TestConductor Add-On). All of this is proprietary, commercial software. Rhapsody version 8.4 is used, running on a Windows 10 PC.

Reasons for the choice of Rhapsody ATG included its high flexibility (e.g. broad support of SysML elements and code constructs) and the integrated tool chain from model creation to test execution. Different tools for system model based TCG like Conformiq Designer or RT-Tester MBT exist, having different strengths like more options to influence the TCG algorithm or supposedly better performance. Also, different FM and semi-FM tools like ProB or Simulink come with TCG capabilities, but usually bound to a special modelling language (as opposed to the wide-spread multi-purpose UML/SysML languages).

2.2 Process

In Figure 1 the steps of the test generation (left-hand side) and test execution (right-hand side) are shown. The tools used for the single steps are given in the white boxes attached to each step. While many steps are largely automated, the main manual effort lies in the “TCG Model Creation”, and – in our case, due to testing of several external SUT – also in the “Implementation Integration”. The most important steps are described in the next section below.

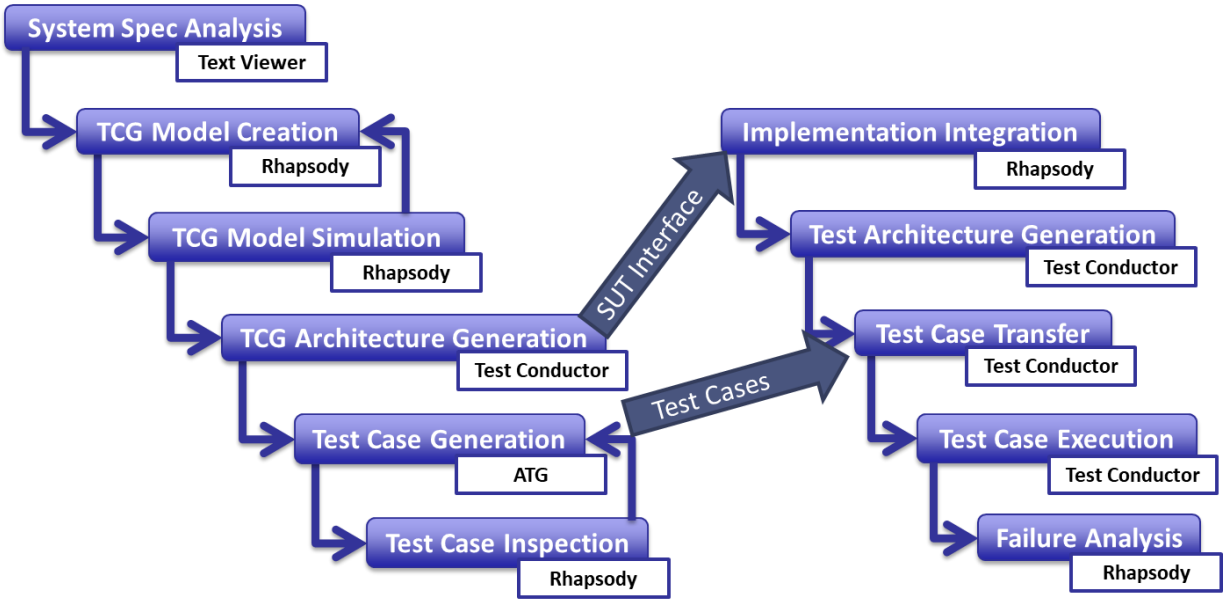

Figure 1: Test generation and execution process with Rhapsody and Add-Ons

3. Application to the LX Controller

3.1 Application Example: The Alex Level Crossing

As an application example for the formal methods mentioned in the introduction and for the TCG an LX controller was chosen by the work package participants. The specification by

Trafikverket of the system called “Alex” is relatively recent and the system is supposed to replace several types of existing LX in Sweden in the future. For the work packages’ purposes, the scope was limited to the variant controlled by an interlocking (no autonomous LX / no private road barriers), which reduced the 375 requirements to 133 requirements in scope. The functions in scope include the interfaces to interlocking, a local control box and speed sensors as well as the control of road-facing lights, barriers, sound, obstacle detection and track-facing signals. Many of these features are configurable in the number and variant of the controlled objects as well as in some delays. An average LX configuration amounts to a system complexity of about 40 Boolean inputs and 50 Boolean outputs.

3.2 TCG Model Creation

At first sight, the creation of the executable SysML system model for the TCG is similar to model-based system development. External interface, the system environment, its structure and its behavior (mainly as SysML statecharts) need to be modeled, as well as requirements that should be traceable and configurations that should be supported. However, a test model should abstract from the real system behavior, e. g. by means of aggregation or omission (in order to reduce the number and length of the generated test cases). On the other hand it is also legitimate to explicitly model behavioral variants that are implicit in an implementation (in order to force generation of corresponding test cases). The statechart modeling the behavior for control of the track-side LX components (signal and distant signal) is shown in Figure 2.

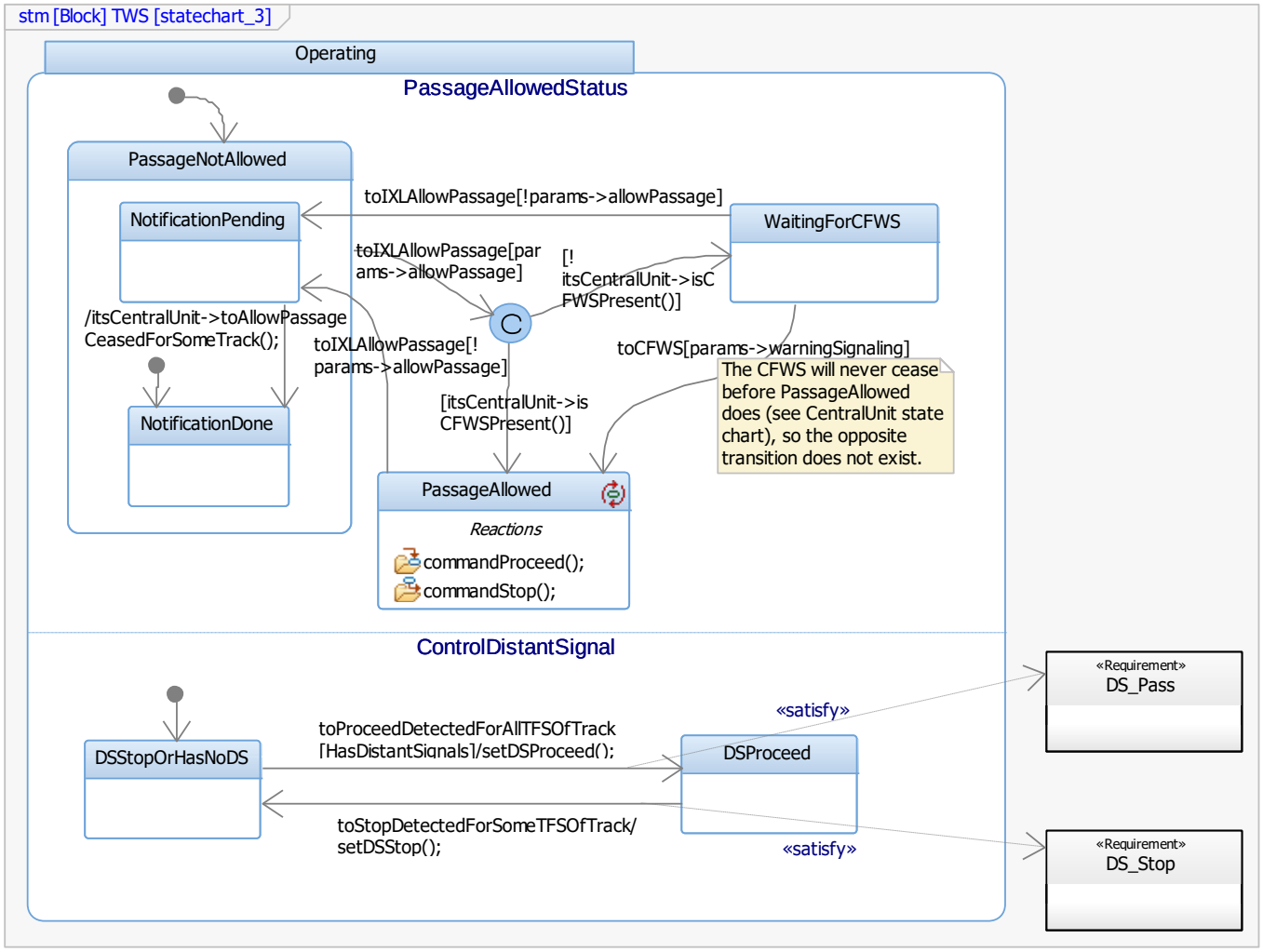

Figure 2: Behavior of trackside LX components control as SysML statechart. Requirements on distant signal control are linked to transitions according to the LX specification (which seems to require that distant signal control depends on detected rather than commanded main signal aspect).

3.3 Test Case Generation

In the “TCG Architecture Generation” step, which is a prerequisite for the TCG step, the SUT part of the model is fixed. Based on this, one defines the sub-interfaces of the SUT that should be used for stimulation (input interface) and recorded (input and output interface) by the generator. Also, one can choose whether Rhapsody ATG will try to reach structural coverage of the model (states, transitions, and operations), coverage of the generated C++ code (modified condition/decision coverage), or both. The current coverage of the different elements is displayed by ATG during generation, see Figure 3. The resulting test cases can be displayed as sequence diagrams in Rhapsody. They can be edited and completed by further generated or manually designed test cases. An example of a resulting test case is shown in Figure 4.

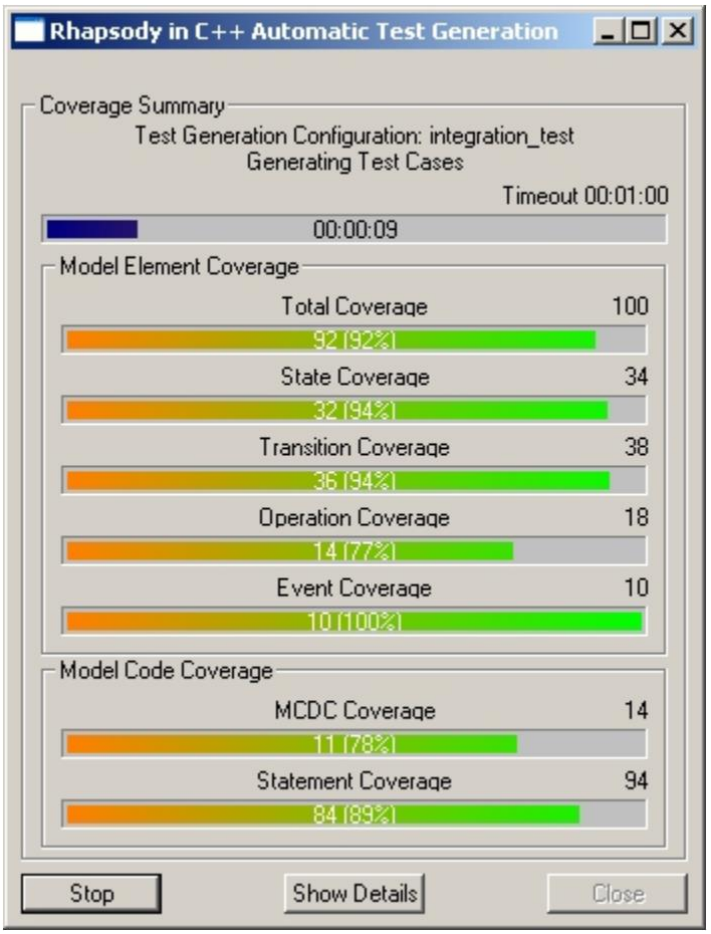

Figure 3: Status window shown by ATG during TCG (taken from [6], p. 52)

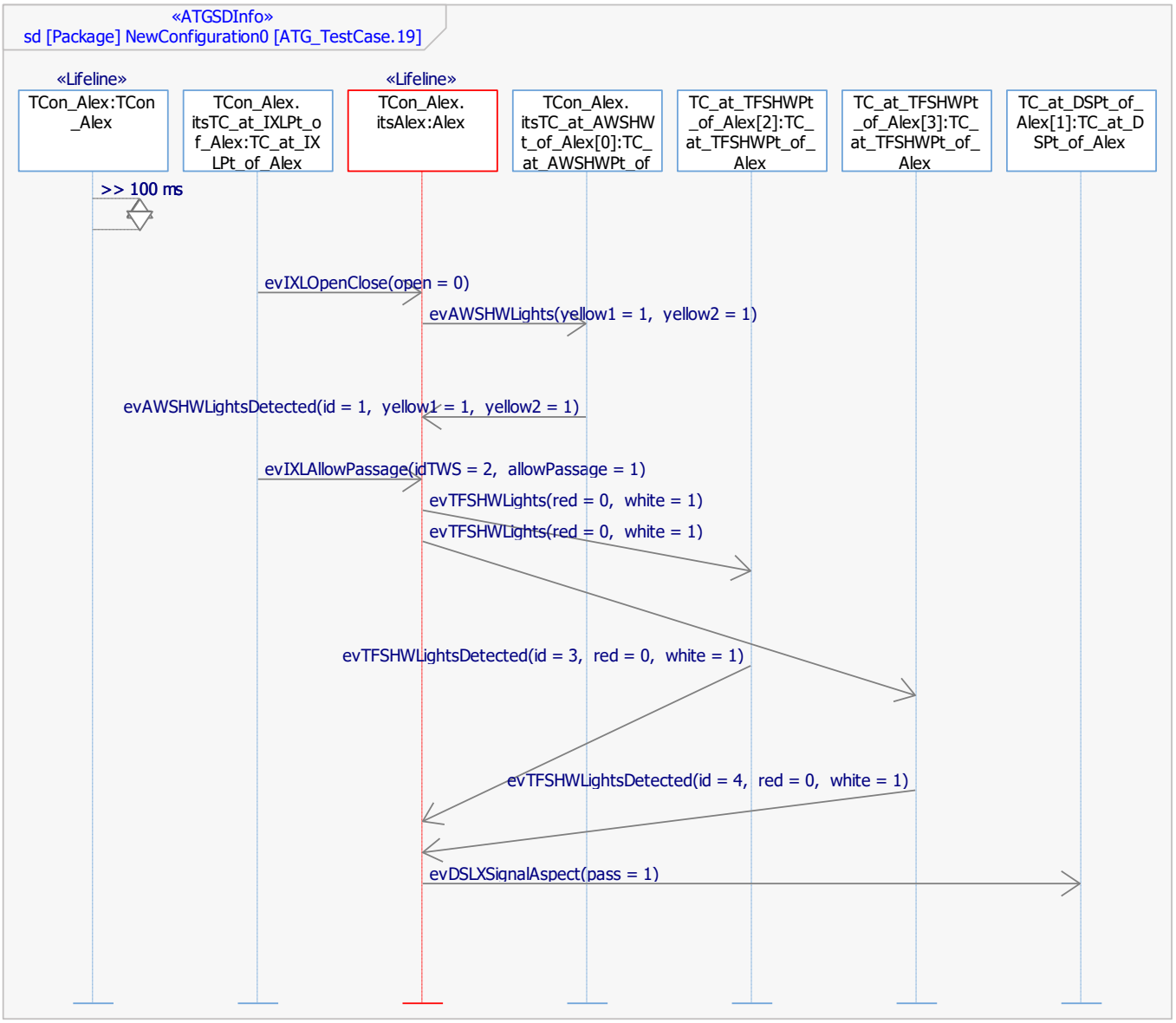

Figure 4: Test case generated by ATG as SysML sequence diagram. The SUT (LX controller) lifeline is colored in red and receives inputs from and sends outputs to adjacent systems (interlocking, different instances of road- and track-facing lights, distant signal).

3.4 Implementation Integration

Rhapsody TestConductor provides convenient means to execute the test cases generated by ATG on a Rhapsody model. In our case, we use TestConductor to execute the generated test cases on the different external software implementations. To this end, a test interface suitable for all implementations under test has been defined in discussion with the developers involved. Several issues needed to be solved:

1. The general mismatch between the relay-based LX interface and the message-based test environment was resolved by means of an implementation wrapper common for all implementations.
2. Some implementers work with real-time execution frameworks, while others prefer to work with an execution-independent fixed cycle time. This was resolved by transmission of the “implementation time” to the test environment each cycle, by adapting the test environment to use that “external clock”, and by keeping the TCG model independent of the Windows system clock.

3. Different programming languages are used by the test environment (C++) and the implementations (generated C code, Ada code, Westrace simulator executable). This was resolved by compiling the implementations into libraries linked by the test environment, or by using another (C++) wrapper in the Westrace case that was compiled together with the test environment.
4. While controlling single cycles of LX implementations from the test environment would be possible for the formal developments, this is not possible for the Westrace simulator. Thus, shared memories for the input and output data for the implementations were set up together with mutual exclusive access from the test environment and the implementation side.
5. The test interface needs to be able to cope with different system configurations that affect the interface (the number of Boolean in- and outputs to transmit). Here, the interface was laid out to cover an agreed fixed maximum number of configured objects such as signals or barriers.

Altogether those issues and the underlying technical details required quite some discussions, clear conceptual work and more effort than expected initially. Nevertheless, it also saves from some effort since the test execution framework of TestConductor can directly be used (simple automated test execution) and only one test interface needs to be defined, maintained and considered during test execution / failure analysis. The test interface did not require any additional behavior of the implementations apart from reading inputs and writing outputs each cycle, including a few test control in-/outputs consumed/generated on the control cycle level.

4. Discussion of the Approach

Currently, first test cases have been generated and executed on an implementation, confirming that our approach is feasible. Yet, no comprehensive test results are available. However, having run through the complete process depicted in Figure 1, we can already report on several topics that seem crucial to us:

System abstraction (TCG model creation step). Since the model for TCG is a system model and is moreover required to be executable, this may push the modeler towards modeling an implementation. This can easily lead to spending time on modeling behavior not needed for testing or to unacceptably long test generation time. It can also lead to particular interpretations of the specification, which may result in unjustified failing tests. It seems advisable to start with clear test goals and system scope, check requirements thoroughly and either make them precise or indicate intentionally left room for interpretation, simplify the external system interfaces as much as possible, and then fix a set of environment assumptions. After that, abstractions of the system behavior can be made and modeled, which sometimes is non-trivial due to interdependencies. Here it is helpful to think in terms of system functions rather than in architectural structures. One rather simple behavioral abstraction made in the LX example was to consider only “normal” behavior as a first step; failure scenarios may be added later.

Managing model configurations and variants (TCG model creation). We decided to create a “150% model” (i.e. there is one model which contains the behavior for all the configurations and variants), which can be arbitrarily configured and where the environment model can be varied through variant points. This leads to less redundancy and better maintainability than individual models, but also leads to a more cluttered model and usually to less than 100% structural coverage by the tests generated for a particular configuration.

Model validation (TCG model simulation). It is important that the TCG model is validated. Rhapsody offers interactive simulation of a model, which is a nice visual tool to check particular model behavior. Also, the modeling process itself often guides the modeler to think about possible issues, and several later steps (Rhapsody's model consistency check, compilation of the generated code, test case inspection) may uncover problems in the model. However, a more direct validation of the model against the requirements has not been performed in our case; at least attaching the tested requirements to model elements made the modeler think about the realization of those requirements in the model.

Scalability (test case generation). It is well-known that many formal methods have high algorithmic complexity. This also applies to TCG with ATG: the time needed to generate a test suite that fully covers the model quickly rises with growing model size. One important factor is the complexity of the model's input interface. Here the relay-based LX interface has proven advantageous, since it consists of a series of wires that can be modeled by simple Boolean values. So far, the generation time has not exceeded a couple of minutes, so that it is estimated to stay within reasonable range also for the future completed model. Note that in ATG the interface can be further restricted by choosing messages and parameter values in a flexible manner for each TCG run, and by modeling environment restrictions in the test components connected to the input interface. A second factor is the complexity of the behavioral model itself; in particular deep nesting may increase generation time. The TCG output complexity (number and size of test cases) seems unproblematic (so far around 20 test cases with an average of less than 10 steps). Test cases that are prefix of another one are automatically removed. Other redundancies may occur, but seem rather insignificant.

Managing implementation variants, versions and configurations (implementation integration to failure analysis). In the given project context, the five different implementations were prepared in four different configurations, amounting to 20 systems under test. In order to be able to adapt test cases individually if necessary, and not to mix test results, it was decided to create 20 separate test architectures for them (which is an automated process in Rhapsody). The test suite for the corresponding configuration needs to be copied into the architecture. Each architecture comes with a code generation component where the library containing the implementation/configuration combination of interest can be selected. Also variants of modeled variation points can be chosen here, a possible mechanism to choose the right variant of the (configuration dependent) implementation wrapper. If, in addition, one considers that several versions of the 20 SUT may be released over time, and different test suites may be generated for them, it becomes apparent that thorough planning of the project structure in Rhapsody is of utmost importance for efficient test execution. Regarding the different test suites for different configurations and resulting from different environment models, a similar planning is necessary for the test generation.

5. Conclusion

The work on TCG from an LX controller model within the X2Rail-2 WP5 indicates that such an approach can be successfully applied to mid-sized signaling systems. Rhapsody and its testing add-ons provide an integrated, flexible and largely automated framework to create a system model, generate tests from it and execute them. The model-based process followed seems to supports a good quality of the central model, although in our case no extensive validation against the system requirements was performed.

The effort and costs to set up and apply such a process for the first time may be high; training

and tool support will be necessary. Also finding the right abstraction level for a system model can be challenging, and it still might be necessary to add some test cases by hand. However, there are convincing benefits:

- Creating a system model often uncovers problems in the system requirements specification early which can save from costly iterations in development, and human error during test design is reduced.
- Since the generated test cases cover the elements of the model they are generated from (which are typically more fine-grained than requirements), they have the potential to detect more errors than those manually designed (covering the mere requirements).
- To have the main portion of functional test cases ready in a model will pay off whenever changes need to be made to the test suite – no matter if during initial development, maintenance or while creating new product version. The central place to implement those changes is the model – and an automatic regeneration of the test cases will consistently apply the change to the suite.
- A cross-platform and implementation-approach-independent test suite with a standard interface allows for reuse (possibly including its further development in future projects).

In the remainder of the project, our claim that the TCG scales well for the LX application needs to be confirmed as soon as the model is completed. It will be interesting to see how many errors will be detected during the execution of the generated test cases on the different implementations, and how model-based testing and formal verification complement each other. Further research will be necessary to try out the limits of the ATG test case generator to compare with manually created test suites and to generate further kinds of tests, e.g. for failure scenarios.

References

[1] S. Bacherini, A. Fantechi, M. Tempestini, and N. Zingoni, “A Story About Formal Methods Adoption by a Railway Signaling Manufacturer,” in Lecture Notes in Computer Science, vol. 4085, 2006, pp. 179–189. https://doi.org/10.1007/11813040_13

[2] D. Basile et al., “On the Industrial Uptake of Formal Methods in the Railway Domain,” in Lecture Notes in Computer Science, vol. 11023, 2018, pp. 20–29. https://doi.org/10.1007/978-3-319-98938-9_2

[3] M. H. ter Beek et al., “Adopting Formal Methods in an Industrial Setting: The Railways Case,” in Lecture Notes in Computer Science, vol. 11800, 2019, pp. 762–772. https://doi.org/10.1007/978-3-030-30942-8_46

[4] Formal Methods (Taxonomy and Survey), Proposed Methods and Applications. Public Deliverable 5.1 of X2Rail-2, revision 1.5 from 16/05/2018, available at https://projects.shift2rail.org/s2r_ip2_n.aspx?p=X2RAIL-2

[5] TestConductor and ATG Web documentation on the IBM website, https://www.ibm.com/support/knowledgecenter/en/SSB2MU_8.4.0/com.btc.tcatg.user.doc/topics/com.btc.tcatg.user.doc.html, accessed on 13/11/2019 3:50 pm.

[6] IBM® Rational® Rhapsody® Automatic Test Generation Add On User Guide, Release 3.6.2