Applying Model-based Testing to a
Train Control System

Klaus Didrich, Stefan Herbst
Siemens Transportation Systems
TS RA E2
Elsenstr. 87-96, 12435 Berlin
Germany
klaus.didrich@siemens.com
herbst.stefan@siemens.com

Marlon Vieira
Siemens Corporate Research
SCR SE
755 College Road East, Princeton, NJ
USA
marlon.vieira@siemens.com

Abstract: In this paper, we report on lessons learned with the application of model-based testing for the system validation of an embedded train control system. The applied approach is based on UML diagrams annotated with category-partition information. The approach consists of modeling, that is, creating UML use cases and activity diagrams from existing documentation, enhancing the models with testing requirements such as test coverage information and data variations, generating test cases in the form of executable scripts, and finally executing the test cases on a simulator in order to stimulate the application under test and verify its response automatically. This paper illustrates all these phases with an example from a train controller development project and proceeds to show that the approach is especially useful for verifying systems that can run in a simulated environment.

1 Introduction

Model-based testing (MBT) aims to automatically generate an efficient set of test cases based on the description (i.e. models) of a system. MBT has its roots in hardware testing; most notably telephone switches [El-Far01]. However, it has recently spread to a variety of software domains. With the wide acceptance of UML and the advent of the UML 2.0 standard, many new MBT approaches which use some kind of UML diagrams and some properly defined coverage criteria for the derivation of tests, have appeared and continue to appear today.

The goal of the research reported here is to investigate practical approaches for applying MBT in an industrial context. This paper describes the approach used to not only create, but also execute, test cases to achieve the system validation of an embedded train control system. The findings reported in this paper are from a project being developed in the context of the ETCS standard [EU96], in which Siemens is heavily involved. ETCS (European Train Control System) defines the requirements for a train control system for cross-border traffic.

The approach described builds on and combines existing techniques for graph coverage and data coverage. It starts by using UML use cases and activity diagrams to describe which functionalities should be tested and how to test them respectively. The category-partition method [OB88] is then applied to introduce data into the UML model. Important issues addressed by the approach described are (a) how to bridge the abstraction contained in the
models to executable test cases; (b) how to ensure coverage with the minimum number of test cases; and (c) how to demonstrate direct traceability between the required functionality and the results of test case execution. Solutions to these issues are discussed later in the paper.

To apply the approach described in this paper, we employ a tool built by Siemens Corporate Research, the Test Development Environment, using UML (TDE/UML). More information about TDE/UML can be found in [HVR05]. [DF06] describes the test method used in a previous ETCS project.

The paper is organized as follows: Section 2 presents the research background and related works. Section 3 discusses the applied approach to model-based testing, including details about modeling, generating and executing the test cases. Finally, Section 4 offers some lessons learned about MBT in practice, and draws conclusions accordingly.

2 Background

Despite encouraging results achieved through quality improvement approaches, the software industry is still far from achieving defect-free software. Testing (i.e. defect identification) remains an important activity within software development and maintenance. The use of models to automatically generate test cases is a promising approach for improving the testing process. Automatic test generation has been extensively studied in recent years. Several testing approaches have been proposed based on data and/or control models. Essentially, approaches based on data models use the specification of a set of possible values to verify a parameter under test. Such approaches are very useful provided the data model is sufficiently refined to capture a (sub)system’s behavior. Dalal et. al. present a method [DKLL98] and report their findings [DJKL99] with testing based on a data model. Ostrand and Balcer [OB88] describe a data model-based method that uses partitioning to generate functional tests. However, if the behavior of a system depends on the sequences of actions, that is, subsequent operations depend on preceding ones, additional modeling (i.e. control information) is necessary in order to provide information for effective testing. Generally, this is the case when testing embedded systems. Although testing processes for embedded systems share some commonalities with other applications, in certain respects they differ: (a) the separation between the application development and execution platforms, necessitating the critical task of verifying the application in both real and simulated environments, (b) the large variety of execution platforms and interacting devices, necessitating a validation of the interaction of the application with different devices (using real or simulated instances), (c) the emerging certification standards to be observed; different projects can have similarities on account of a shared standard, but differences on account of customization, and (d) the tight resource and timing constraints on the execution platform – running a test case on a real train in the field, for example, is a very expensive undertaking, requiring careful planning.

3 Supporting embedded system validation with MBT

This section discusses the approach used for the validation of an embedded train controller. Broadly speaking, the process (depicted in Figure 1) consists of the following steps: (1) modeling of the system behavior, (2) enhancement of the models with testing information, (3) generation of test suites, and (4) execution of the test cases.
It should be noted that steps (1) and (2) are generally performed several times as an iterative cycle. Typically, new or missed functions are added to the model (or existing ones respecified/corrected) and the model is then enhanced to refine the functionality. Frequently also, mismatches become apparent when the tester attempts to execute the test cases (step 4); the models then need to be refined (step 2) again to close the gap between the model and implementation and permit effective script generation.

To better convey the concepts used in the approach described, we will be illustrating them with an example. The example chosen is part of an ETCS-specific project, representing basic functionalities of the application. We shall not go into the technical details of these functionalities; however, the tasks of modeling and generating a test case for the “start-of-mission procedure” will be discussed.

### 3.1 Modeling

We use UML use case diagrams to describe the relationship between the diverse use cases specified for the system and the actors interacting with the system in accordance with these use cases. UML activity diagrams are used to model the logic captured by individual use cases and provide the basis for testing the functionalities and business rules described by the use cases.

Modeling application behavior in this approach is to define use cases and describe how to test them with activity diagrams. The different paths through the diagram are subsequently used to construct test cases during test case generation. In an ideal situation, use case modeling is performed by analysts during the requirements phase. Alternatively, the models can be created for testing at a later point. Generally speaking, the models (i.e. use case + activity diagrams) can be used as a representation of the functionalities to be tested and as a roadmap for testing them.

### 3.2 Model Enhancement

Models typically need to be enhanced to capture the information needed to generate useful test cases. We can classify these enhancements into two types: the first is the refinement of the activities of activity diagrams. This refinement generally leads to an improvement in the accuracy (i.e. a lower level of abstraction) of the functionality described, thereby increasing the efficiency of the test cases. The second type of model enhancement is characterized by annotations in the activity diagrams. The approach described supports annotations by providing custom stereotypes for use in notes anchored to a particular activity in the diagram.
These stereotypes represent additional test requirements and are typically added by the test
designer. Examples of annotations are data inputs in the form of “categories”. These data
inputs are combined with the generated test paths and influence the test generation process
primarily in terms of the number of test cases generated, the functional coverage attained and
the data used in each test case. This combination of an activity diagram with data annotation
enables our approach to generate test cases that fulfill specific coverage by graph and data
coverage criteria.

Fundamental to our UML-based testing approach is the category-partition method [OB88],
which has been the object of long-term research at Siemens Corporate Research. The
category-partition method identifies behavioral equivalence classes within the structure of an
application under test. A category or partition is defined by specifying all the possible data
choices that it can represent. Such choices can be either data values, references to other
categories, or a combination of both. The data values become inputs for test cases and can be
elements in guard conditions of transitions. In our approach, we represent categories and
choices using the notation of UML classes and class attributes respectively. Figure 2 shows
an extract of the classes for the ETCS project example.

![Figure 2 – Categories and choices for the ETCS project example](image)

All data objects relevant to a use case are modeled as test variables in the activity diagram.
They are used to express the guard conditions in branches and to specify the data variations
for test generation. The categories defined by classes (Figure 2) provide the input data: the
data choices in a category are the possible values of the variable. During test generation, the
activity requires one of the data choices of the category as an input in order to become a
complete test step. Before test variables are used in branching conditions or for data
variations, they must be defined and associated with an activity (i.e. supplemented by an
annotation). Once a variable has been defined, it can be used in the guard condition of a
transition. The syntax for describing guard conditions is based on that defined for the
Siemens Test Specification Language (TSL).

To illustrate the above concepts, parts of the “start-of-mission procedure” activity diagram
are highlighted in Figure 3. In order to be performed, the use case requires a specific OBU
(on-board unit) mode and CAB to be active (i.e. there is a precondition). To model this, we
introduce the “mode” and “CAB” variables from the categories with the same names
(Figure 2). Then a guard on a transition out from the initial state “[mode = SB and CAB =
active]” declares that “mode” needs to be equal to “SB” (i.e. standby) and “CAB” needs to
be active. Otherwise, the test case will stop executing. Later in this use case, the driver is
required to enter/revalidate his ID; in the event of an invalid “DriverID”, execution will also
cease. In the test generation phase, the test paths containing this transition would only have
the data combinations that have specified “Valid” as the data choice for DriveID. In the case
of the defined variable “StoredLevel” (which contains 4 permissible choices according to Figure 2), the generator will choose the value for use in the test cases, since there are no restrictions (no guard on a transition) to limit the selected choices.

Figure 3 – Example of a “start-of-mission procedure” activity diagram

3.3 Test Case Generation

Before proceeding with a description of the test generation characteristics, we would like to emphasize that our approach generates a set of functional conformance tests. These tests verify compliance of what is implemented with what was modeled. It is assumed that the implementation behaves in a deterministic and externally controllable way. Otherwise, the generated test cases may be ineffective. From the UML activity diagram and the category data, test paths are created. As mentioned earlier, we use TDE/UML to generate these test cases. TDE/UML allows the user to specify a wide range of parameters for test case generation.

Based on previous experience with template generators, we established the following design principles for the ETCS Test Framework: (1) generated files must not require any manual changes, (2) generated test scripts should be executable directly, (3) the framework should be flexible to allow the addition of new choices without manual coding and easy incorporation of changes into scripts, and (4) the reuse of test cases at other locations, in other projects and in different test environments, should be supported.

Figure 4 – ETCS test framework architecture
We finally obtained a multi-layered architecture (Figure 4) with a strict separation of responsibilities:

(A) **Generic layer:** This layer is responsible for the generic test case logic. The most important method in this layer appears in pseudo-code as follows:

```plaintext
method run
    initTestEnvironment
    establishPrecondition
    checkPrecondition
    body
    checkPostcondition
end
```

All the methods called by the method run are virtual and need to be filled with concrete instances. The generic layer also provides functionality to record intermediate results and to generate an overall verdict of the test case execution. Certain details, such as exception handling, timeout handling and managing the test protocol, which are all handled at this level, have been omitted.

(B) **Test environment layer:** This layer provides the necessary functionality for setting up the test environment. It defines the method initTestEnvironment. In the case of the test environment which we are using in our project it is necessary to introduce other virtual functions that define the track data to be used, or the initial state of the environment, e.g. the states of signals and switches. The test environment layer also contains a library, so the functionality of the test environment can be accessed in a more test-friendly way.

(C) **Project customization layer:** This layer contains definitions for easy access to the proper track data and train types, as well as a project-specific library of methods.

(D) **Use case layer:** The use case layer is coded by the test engineer, who has to provide definitions for the establishPrecondition method, and all the methods that will be used in the automatic definition of the checkPrecondition, body and checkPostcondition methods. The test script generation tool needs to be able to produce such a list. The coding must be done in such a way that it will allow the same method to be used for different test cases. The test engineer may use variables containing the information regarding which choice is currently valid for which category, in order to determine what speed is required in a particular test case. If applicable, the test engineer has to provide code to check the output and record an intermediate result. Coding the use case layer is more complicated than coding a script for a single test case and it may be that the coding of the use case layer will take more time than the coding of individual test cases. However, once done, it offers more flexibility, for example, if it is decided that more test cases (more choices in a category) are required, or if the logic changes.

(E) **Test case layer:** The test case layer is generated automatically. For each combination of a test path and test data the proper variable definitions and method calls are generated. Since everything else is taken care of in the other layers, the automatically generated code is quite
simple and appears as follows:

```ruby
class TC_Path0_Data0_ABSTRACT inherits from ...
  method new
    var_cat1 = “choice1”
    var_cat2 = “choice2”
  end
  method checkPrecondition
    checkMethod1
    checkMethod2
  end
  method body
    methodStep1
    methodStep2
  end
  method checkPost-condition
    checkMethodX
    checkMethodX+1
  end
end
```

The test script generation tool thus only requires a limited knowledge of the test environment. The script language is of course fixed but the set-up of the test environment does not have to be coded in the generated test script. Note that no editing of the generated code is necessary. Everything that needs to be defined is defined in the use case layer.

### 3.4 Test Case Execution

At this stage the test cases are almost executable, but we still need to define the location for execution of the test case. The test cases are described using abstract locations, such as “Signal S1” or “second switch after entry” (hence the suffix “ABSTRACT” in the test case layer), which need to be instantiated. Actually, we could include categories for these locations with the appropriate choices and let TDE/UML generate the possible combinations as well. But this is not a viable solution. First, we need to execute test cases only at one or two locations, and second we need to be flexible to change the location on short notice. We therefore introduced another layer, the **test data layer**, to make it easy to execute the same test case at different locations. In order to keep the set of parameters consistent for different test environments, namely the laboratory and field, the parameter sets were also placed under the control of the test case generator. At this point we can finally execute the test script by creating an instance of the class and running it.

### 4 Lessons Learned and Conclusions

The orientation towards the Siemens industrial context imposes certain extra requirements and constraints on the MBT approach adopted. First, it is not wise to assume that a formal (complete and consistent) model of the software system already exists. Instead, it is reasonable to assume that an informal description (i.e. one in natural language or an informal drawing) documenting the system-required functionalities (or part of them) is available and that this should be the basis for creating the models. Modeling is thus a fundamental prerequisite for applying MBT successfully in a typical industrial context. Second, an MBT
approach can be effective only if the total effort it requires is affordable. This implies that the approach: (a) is simple enough for it to be easily learned, (b) is powerful enough to produce an effective set of test cases (thereby saving time/effort), and (c) also facilitates the management of system modifications (e.g. changing requirements due to product evolution).

The approach described is currently evaluated in an ETCS-related project by Siemens. The results from the application of the described approach in an ETCS project have so far been very encouraging, but also show that it is important to examine the details of the implementation and quickly eliminate any obstacles. Minor details may otherwise bring the testing process to a standstill, which is not acceptable in an industrial environment and lead to rejection of the MBT approach. Experience has shown that MBT does not necessarily offer immediate benefits with respect to traditional approaches. In the beginning, we need more time before we can start generating test cases than is needed with the manual approach, due to the development of a formal model. Nevertheless, we were able to save time in the test case specification phase, and expect that we will be able to save more time in the test case execution phase.

Further work could focus on checking whether the test cases can be reused in other projects. The framework was designed so as to allow its reuse in other projects, but this has not yet been put to the test. There are other issues relating to test case execution that also need to be addressed when a testing project is planned: the generation of test cases that meet specific coverage criteria, but also the minimization of the number of test cases while still ensuring the same specific coverage criteria. Another challenge is the provision of support for the selection of regression tests after changes have been made to the model. Since it should be easy to find out which test cases are affected by the change of an activity, it should also be possible to determine the appropriate regression tests with tool support.

References