Delft University of Technology
Software Engineering Research Group
Technical Report Series

Observation-Based Modeling for Model-Based Testing

Teemu Kanstrén, Éric Piel and Hans-Gerhard Gross

Report TUD-SERG-2009-012

TUD-SERG-2009-012

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Observation-Based Modeling for Model-Based Testing *

Teemu Kanstrén
VTT
Kaitoväylä 1, 90571 Oulu, Finland
teemu.kanstren@vtt.fi

Éric Piel Hans-Gerhard Gross
Software Engineering Research Group
Delft University of Technology
Mekelweg 4, 2628CD Delft, The Netherlands
{e.a.b.piel,h.g.gross}@tudelft.nl

Abstract

One of the single most important reasons that modeling and model-based testing are not yet common practice in industry is the perceived difficulty of making the models up to the level of detail and quality required for their automated processing. Models unleash their full potential only through sufficient formality, and after being processed by tools. However, making sufficiently formal models is regarded by industry as laborious, expensive, and overall daunting.

This article presents a solution for circumventing the manual modeling process by bootstrapping and devising the model automatically from observations, and using that model for test case generation. It describes a combination of existing techniques and tools, some of which are readily used in industry, to provide an iterative process with which test engineers can create a formal model of a component, use it to generate test stimuli, compare the observed component behaviour with the modeled behaviour, and refine the code of the model or the component quickly and easily, according to the issues detected. Moreover, this process turns modeling more into an activity akin to programming, favoured by developers. The technique is demonstrated on one case study, in which it finds issues on both the implementation and the specification.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: testing and debugging—test execution, test management

1. Introduction

Testing is the most commonly used approach in industry for verification and validation of software, and it can be regarded as the ultimate review of a system's specification, design, and implementation [7]. Model-Based Testing (MBT) refers to all automated test case generation techniques that are based on formalized descriptions of the System Under Test (SUT), in contrast to handcrafting test cases from other available (non-formal) documents, or the source code [20]. Since testing can, often, consume up to half of the overall development cost for a software project, while it adds nothing in terms of functionality to the software, there is a strong incentive towards test automation with MBT. Once the models are made and appropriate tools are available, model-based testing is a push-button solution. Unfortunately, making sufficiently formal descriptions, i.e., the models of the SUT that can be used for automated processing and test case generation, can be seen as the most significant cause for MBT not having penetrated industrial practice yet. Making the models is an activity that does not add any immediate additional value to the final product, and it is typically perceived by practitioners as being difficult, expensive, and overall daunting.

One solution for circumventing the difficult and costly manual design and construction process to obtain models for MBT is to generate them out of observations automatically [8], e.g., with the aid of a process mining technique [22]. Obviously, this method of observation-based modeling can only be "boot-strapped" from existing runtime scenarios and their executions. Examples of this type of scenarios are demo applications, user sessions and existing test cases (VIITE). Because most typical software projects in practice exhibit such properties, observation-based modeling can be adopted easily by practitioners, and can, eventually, offer automated support for constructing system specification models to be used for various purposes, including MBT.

This paper presents a compilation of techniques and tools that have been combined and integrated in order to devise an iterative and automated method to support the creation of behavioural models out of execution traces (observations). Those models are especially purposed for model-based testing. The contributions of the paper are

- a technique with a concrete tool chain for generating behavioural models out of system execution traces,
- a method with guidelines for manual refinement of the generated behavioural model,
- guidelines for applying the aforementioned method for devising test cases based on the models, and
- a concrete example to illustrate and evaluate the application of the method and tools.

Sect. 2 briefly outlines related work on the methods and techniques relevant to this article. Sect. 3 describes the tools and techniques used for OBM, and how they were integrated to generate a behavioural model from execution traces semi-automatically. Sect. 4 summarizes a concrete application of OBM for testing part of a maritime surveillance system, and discusses experiences with the techniques. Finally, Sect. 5 concludes the paper with directions for further research.

2. Background and Related Work

The observation-based modeling approach presented in this paper uses dynamic analysis as primary underlying model extraction technique. It means the running system is observed, and traces of its behaviour are captured. In this respect, the approach presented shares the basic properties of a number of techniques in fields relying on dynamic analysis, such as reverse engineering and program comprehension.

These techniques require models to be built at different levels of abstraction, based on information collected from program artifacts.

* The work presented in this paper has been carried out partially under the Poseidon project in cooperation with the Embedded Systems Institute (ESI), Eindhoven, The Netherlands, and supported by the Dutch Ministry of Economic Affairs (BSIK03021 program). This work has been supported by the Nokia Foundation.

Observation-based modeling follows the same principles, although, with the goal of using the obtained models for testing and related activities. A common application in dynamic analysis is to trace the method and function calls and use this data for building models of the system [9]. (VIITE?) Since most modern programming languages use methods as basic building blocks of programs, the techniques can be applied extensively. When external tracing mechanisms such as aspects are used [15], they have the added benefit of not requiring access to, or modifications in the source code. Various tools provide models and visualizations from this data, such as sequence and scenario diagrams, graphs and custom visualizations [9]. (VIITE?) Of these, we focus on tools most related to our approach.

Bertolino et al. [1] provide the basics of the concept we apply here, using the term anti-model-based testing. They describe how a set of program executions based on usage profiles can be devised, and how traces can be used, which are collected from these executions as a basis to synthesize a behavioural model. Besides describing the basic concept of generating a model and analysing it for unexpected behaviour, they do not take the concept further.

Ducasse et al. [4] use logic-based queries on SUT execution traces to test legacy systems. Their traces include events and object states, recorded from program execution. Events are messages sent between objects, including parameters and return values. The states of objects are recorded through their interfaces, including states of nested objects. To validate the assumptions about the SUT, they use logic queries on the traces and define a set of trace-based logic testing patterns. They use these tests to check for possible software regressions after updates, and for supporting the understanding of a program, by creating and validating assumptions about it. Their work is closely related to this paper w.r.t. to applying similar traces for testing and facilitating the understanding of legacy systems.

Whereas our focus is on dynamic analysis, related tools and techniques also exist in the field of static analysis. For example, Walkinshaw et al. [23] use symbolic execution to mine state-machines from the source code, including the paths that lead to these transitions. Their work is related to ours in mining for state-machines and using method calls as a basis for the work. They describe using their models for such activities as supporting inspections and design activities, and program comprehension. Although our focus is on MBT, our models could be used for these activities as well.

D'Amorim et al. [3] apply symbolic execution and random sequence generation in order to mine method invocation sequences. They use these techniques to generate unit tests. The test oracle is devised based on catching yet un-caught exceptions, plus monitoring the results of executions that violate the program's operational profile. The operational profile is described by an invariant model, built with Daikon[1], and is based on existing test suites. We follow this approach in that we generate a test oracle as part of our model, and in that the test oracle supports the same properties and the same type of verification for interaction sequences.

Lorenzoli et al. [13] present an algorithm called "GK-tail" that can be used to generate an extended finite state machine (EFSM) from execution traces. This is based on finite state machines (FSM) and Daikon-invariants similar to our approach. They use the EFSM for test case selection and for building an optimal test suite from existing test cases in order to increase coverage of the model.

They also compare the interactions of a component in a new context in order to observe changed behaviour [14], and suggest to use this information to update existing test suites. Our approach uses similar building blocks to partially automate the generation of the EFSM and we also share the application domain of test automation. However, the model representation and application are different. In contrast to our approach, Lorenzoli et al. do not generate tests from the model, and they do not use the model as an executable specification to verify the SUT against its specification.

[1] `http://groups.csail.mit.edu/pag/daikon/`

3. Observation-Based Modeling

This section describes our approach of Observation-Based Modeling which will be demonstrated with a concrete example in Sect. 4, to illustrate how it can be applied to MBT. Observation-Based Modeling (OBM) can also be referred to as Test-Based Modeling (TBM), a concept used sometimes in Test-Driven Development (TDD) and Agile Development [19].

The term Model-Based Testing has many definitions, and we use a definition by Utting and Legeard [20] who describe it as "Generation of test cases with oracles from a behavioural model". The model describes the expected behaviour of the SUT, and is used to generate sequences of method invocations and data as SUT stimulus. In order to validate the correctness of the responses from the SUT, test oracles check the expected output data and interaction sequences. The basic constituents of an MBT system include the system specification that is used as a basis to create the test model, the test tool required to generate tests based on this model, and the test harness (for online-testing) or test script generator (for offline-testing).

The OBM approach presented in this paper generates the test model, the test harness and the test oracle, all as one object, by using the execution traces of the SUT as an initial specification. Using an MBT tool, our approach can generate and execute tests based on this model in order to assess the implementation of the SUT. EFSM, which are commonly used for behavioural modeling and model-based testing [20, 18], describe the system or component in terms of control states and transitions between these states. States are externally visible abstract representations of the component's internal variable combinations. They are modified by the effects of transitions, and initiated through stimuli sent to the component under test. Specification items of a model can be translated into test programs that stimulate the SUT, and the outputs from the SUT executions are compared to the outputs defined by the model.

Observation-based modeling turns this approach around as described in [1]. It turns traces captured through executing the SUT into a model. This is done through the execution of existing test cases while monitoring the stimuli sent and retrieved, and the analysis of the captured information with process mining and invariant analysis tools. The outcome of these tools can be turned into a rudimentary model, to be refined and completed both manually and by further test cases. This approach is outlined in the following subsections.

3.1 Case Example

For the rest of the paper, we use a running example based on components (`Merger`) of a maritime surveillance system. Here we present the basic concepts of `Merger`, and in the following sections we use this to illustrate the different concepts related to our OBM method. In Sect. 4 we discuss our experiences in using OBM for testing the `Merger` component.

The `Merger` system receives information broadcasts from ships called *AIS messages* [10] and processes them in order to form a situational picture of the coastal waters. The (simplified) architecture of this system is displayed in Figure 2. The system comes with a specification in plain English defining behaviour and communication protocols of its components. The components are implemented in Java specifically crafted to be executed under Fractal [2], a component middleware platform. The Merger component was selected as SUT for our case study because it exhibits complex interaction with the other components. It acts as a temporary database of AIS

messages, and client components can consult it for track information of a ship. It can also be asked by clients to be notified of certain ship events, and it is key to displaying ship tracks on the screen of the command and control centre.

3.2 Requirements for OBM

There are a number of initial requirements that an observation-based modeling approach should abide to. First, it should work with black-box components and systems, for which source code is not necessarily available, so that the external interface of an SUT is the only required documentation to start with. Second, it should be easy to use by practitioners, meaning that the user of the observation-based modeling approach is not required to have extensive knowledge or experience in modeling and formal specifications.

The only requirement we set to the user is knowledge about setting up the SUT for testing, and experience with respect to the domain concepts of the SUT, i.e., its context. The domain concepts comprise creating and setting up initial testing states for the SUT, creating objects that the SUT uses (runtime context), and some basic understanding of its function in the context, i.e., its required functional specification. These are all basic requirements of what one could expect from a developer or tester in any arbitrary software project.

While OBM is able to provide a model of what the SUT does, it is essential that the user understands whether the observed behaviour is indeed also its expected behaviour, or whether the SUT deviates from what is expected. Typically this information is available in the form of some specification, i.e., the required specification of the context in which the SUT is going to be integrated. A final requirement is that the user should be able to work with and amend the model, i.e. without having modeling experience or knowledge of formal methods (an earlier requirement). In order to accommodate this requirement, we use Java (in ModelJUnit format) for the modeling part, so that Java programming skills are sufficient to master the modeling part. The remainder of this section describes the OBM approach in detail.

3.3 Turning Observations into Models

In order to derive models from observations, we make use of many existing techniques and tools. During model generation we use Daikon, a tool for dynamic detection of invariants, and PROM[2], a process mining tool. To represent the model in Java for MBT, we use ModelJUnit[3], JUnit[4] and EasyMock[5].

Daikon describes the SUT in terms of a set of invariants over its data values. For example an invariant could be a client name parameter value always being present in a list of subscribed clients, i.e., `ClientName` is a variable and `Subscriptions` a list of the names of subscribed clients. These invariants are inferred from a set of program executions. Thus, they are only as good (generic) as the initial test cases used, and are only meaningful for the scenarios used in the test executions. In general, they can be described as properties that hold at certain points of the SUT execution [6].

PROM is intended to mine process models from event logs [22]. It can produce a variety of models, such as petri-nets and FSM. The FSM model is used as a basic model in our approach. The FSM is generated through PROM's in-built transition system miner plugin, which is described in more detail in [21]. It is augmented with the invariant model from Daikon in order to turn it into a full EFSM.

ModelJUnit is an MBT tool that uses executable EFSM models expressed in Java for generating test cases. JUnit is a commonly used unit testing framework for Java, employed to execute test cases, and to provide the test oracle for asserting the correctness of output data received from the SUT. EasyMock is a mock-object-framework or a test-stub-framework for Java. It supports the creation of programmable test stubs that can be used to isolate the tested part from the rest of the system, and for verifying the correctness of the interactions between the SUT and its environment. EasyMock is used for creating a test harness and a test oracle representing the expectations towards the SUT.

3.4 Capturing a Trace

Observation-based modeling starts by capturing a trace of the SUT behaviour which is coming from a set of executions. This is depicted as Step 1 in Fig. **??**. These runtime scenarios may be based on any available real executions, e.g. from performing tests, or from nominal field data captured while using the component under consideration, as discussed in [5]. Since we generate a test model including test oracles based on program traces, we expect these traces to describe the *correct* nominal behaviour of the SUT. Test cases focusing on error handling properties of the SUT will expect exceptions or error codes to be thrown, and they should not be part of this set of executions. Thus, separation of nominal and exceptional behaviour helps generating a better, more concrete model. A simple way to retrieve the right stimuli is to use an existing test suite with a categorization of scenarios for the SUT, such as nominal behaviour, or exceptional behaviour. For example, this can also come from usage scenarios defined by use cases. We acknowledge the fact that in real systems with thousands of test cases and realistic field data, such classification is often difficult to obtain, and therefore is not strictly required. However, such test categories are "nice-to-have" for ease of model generation. Not having them simply means that the more "error state"-related behaviour will be represented in the traces, leading to more refinement effort to be performed by the integration test engineer, in order to retrieve a good model, eventually.

The information required to be captured in the trace includes the external messages passed through the input- and output-interfaces of the SUT and its global state when each message was passed. The messages are captured at the interface of the SUT, if used as mere black-box modeling approach. Adding global state information requires design for testability support by the SUT, such as additional test interfaces, following [7], or serialization interfaces. In case the SUT does not support such a test interface, it is also possible to maintain an "artificial" state within the component that monitors the SUT external interfaces by observing the inputs and outputs of the component and classifying them by type. This technique is again a full black-box approach.

For Step 1 of the process, the user must define what should be part of the trace. Typically, tracing comprises a list of methods (function points) or messages representing the external interface(s) of the SUT. Also fine-grained (white-box) tracing information can be used if needed but we have focused on keeping to a fully black-box approach. The instrumentation tool must then be connected with our Tracer (or other similar) component to produce the input logs compatible with our model generator. Note that when a component framework is used as a middleware, this framework can be instrumented to capture all component interactions, thus avoiding the need to separately instrument each component. More discussion on these different approaches can be found in [11].

[2] `http://www.processmining.org`

[3] `http://www.cs.waikato.ac.nz/~marku/mbt/modeljunit/`

[4] `http://www.junit.org`

[5] `http://www.easymock.org`

3.5 Model Generation

Our tool[6] for EFSM generation takes as input the trace captured in the previous step, the list of the monitored SUT input and output interface classes, and the SUT implementation class for the input interface. As output the tool produces the EFSM in ModelJUnit format.

As a first phase in Step 2, PROM is used to produce an FSM from the trace. This FSM consists of all the input and output messages of the SUT represented by states, and the transitions between these states describing the discovered message sequences. For example, Figure 1 shows the FSM produced by PROM for our `Merger` component example, used as a running example, and described in more detail in Sect. 4. The states in this model represent the messages (method invocations) sent to and received from the external input- and output-interfaces of the SUT. The transitions represent the order in which these messages occurred in the trace. The simple behavioural model shown in Figure 1 can be used already by an engineer as useful indicator of the kind of interactions performed when using an SUT in its real execution context and to see if the model seems correct and complete w.r.t. what is being tested. At this stage, it is not necessary for the user to amend this model manually, since a model generator will run the required algorithms automatically. Amendment of the model will only be required later on, in the model refinement phase, described in Subsection 3.6, in order to assess desired and faulty behaviour.

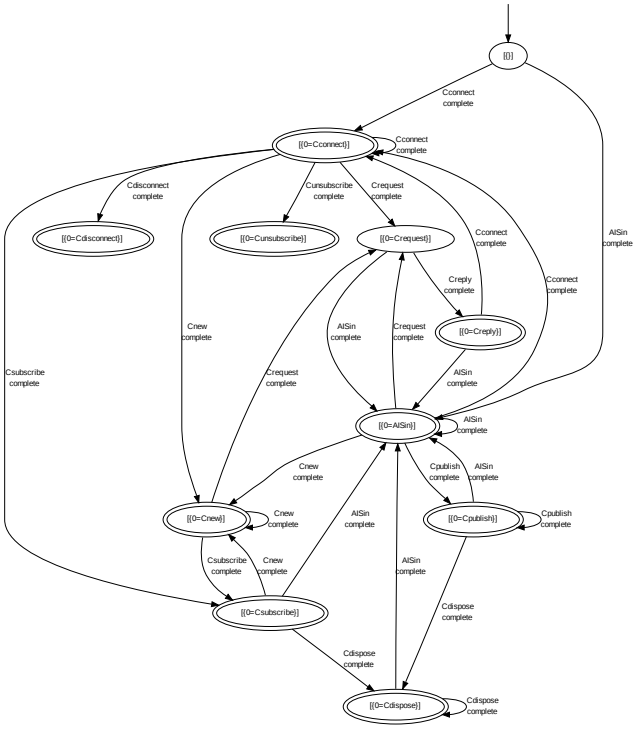

Figure 1. Merger component FSM produced by PROM.

In the second phase, Daikon is used to produce an invariant model from the trace. This model describes the invariants for data values related to both states and transitions. For example, Daikon produces invariants for the `Crequest` state, shown in the centre of Figure 1, as well as for any of its outgoing transitions, i.e., the transitions to `Creply` (`Creply_complete`) and `AISin` (`AISin_complete`). This is done for all states and all transitions of the FSM.

In the third phase, the names of the FSM states are mapped to respective method names of the input- and output-interfaces. The input- and output-interface definitions are part of the input provided by the user to the model generator, which is able to parse these definitions from the given class files. Since method names are used to describe the states in the FSM, making this mapping is straightforward.

The final phase of this step combines the FSM, invariants, and interface-mapping data into one single EFSM in ModelJUnit format. A sample listing with different parts of the model generated for the `Merger` component is shown in Listing 1. The generation of the `reset()` and `getState()` methods for ModelJUnit is relatively simple to perform. The reset method is always generated. It resets all existing mock objects, re-creates the SUT object, and sets the model state to its initial state. The generated `getState()` method returns a string that is updated by the `@Action` transition methods to describe the current state of the model. The three cleared variables `Messages`, `Subscriptions`, and `Clients` in the reset method in Listing 1 are examples of global state variables generated by our model generator. This is done by examining all invariants produced by Daikon, and, based on a chosen set of invariants such as "Client always in Subscriptions" ("Client" parameter value is always part of the global Subscriptions list), matching Java Collection objects are generated in order to store the global state of the EFSM. The `reset` method in Listing 1 clears and resets the global state after each completed test. One more item visible here is the object for the SUT that is being tested. In this particular instance it is the `aISMerger` object, reference to which is also stored globally in the model, to permit the different EFSM methods to access it as a part of the test harness. In addition, an empty template is generated for setting up the SUT object, `createAISMerger` in Listing 1. The SUT data type is provided by the user as part of the EFSM generator configuration along with SUT input- and output-interface definitions. The user must fill in the code skeleton for the `createAISMerger` method in order to set up the SUT object correctly. The mock objects for all output interfaces are provided by the model generator as parameters to this method call in order to enable the user to write the required setup code for the SUT.

The transition `@Action` methods and their related guard methods represent the largest and most complex element of the EFSM generation. They are generated for all methods that belong to the input interfaces for the SUT. As an example we will consider the `Crequest` (input) method of the Merger component referred to above. The first `@Action` method generated is matched to the state for the method in the FSM displayed in Figure 1. This `@Action` method is called `Crequest`, corresponding to the state and method name. Subsequently generated `@Action` methods represent the transitions from this state (method) to states matching the names of methods from output interfaces of the SUT. `Crequest` in Figure 1 has outgoing transitions to `AISin` and `Creply`. Since `AISin` belongs to the input interface of the Merger component, an `@Action` method is not generated for the transition from `Crequest` to `AISin`. Instead, `AISin` will get its own `@Action` methods, similar to `Crequest`, as they are both input methods. In contrast, an `@Action` method is generated for the transition from `Crequest` to `Creply`, because `Creply` is an output method of the Merger component. The code for this `@Action` method is shown in Listing 1. Each of these generated `@Action` methods for the transitions is named by prefixing the input method with the output method name(s) that comes next in the transition. For example, the `Crequest_Creply` represents the transition from the `Crequest` input method to `Creply` output method.

[6] `http://sourceforge.net/projects/noen/`

```
...
public void reset(boolean b) {
  state = "";
  System.out.println("- TEST "+testIndex+" -");
  testIndex++;
  Messages.clear();
  Subscriptions.clear();
  Clients.clear();
  EasyMock.reset(mockClientRcv2);
  try {
    aISMerger = createAISMerger(mockClientRcv2);
  } catch (Exception e) {
    throw new RuntimeException(e);
  }
}

...

@Action
public void Crequest_Creply() throws Exception {
  this.state = "Crequest->Creply";
  expect(mockClientRcv2.Creply((AISMessage)anyObject()))
                              .andReturn("ok");
  replay(mockClientRcv2);
  ReturnStatus rv5 = aISMerger.Crequest(Crequest_p0(),
                                        Crequest_p1(),
                                        Crequest_p2());
  assertEquals("ok", rv5);
  verify(mockClientRcv2);
  EasyMock.reset(mockClientRcv2);
}

public boolean Crequest_CreplyGuard() {
  if(ClientsIsNot_myclient()) return false;
  if(ClientsAreDifferentFrom_myclient_()) return false;
  if(SubscriptionsIsNotEmpty()) return false;
  if(ClientsSizeDoesNotEqual1()) return false;
  return true;
}
...
private String Crequest_p0() {
  return (String) randomItemFrom(Clients);
}

private int Crequest_p1() {
  return (int)1.0;
}

private byte Crequest_p2() {
  return (byte)1.0;
}
...
private String Cdisconnect_p0() {
  return (String) randomItemFrom(Clients);
}
```

Listing 1. Generated reset method and sample transition (@Action), guard and parameter value generation methods for Merger.

Following these principles, the ModelJUnitTest-code for the test harness and test oracle parts of the model are generated inside the `@Action` methods. Generating the test harness element is simple. The SUT object has been created as described earlier, and the model contains a global reference to this object. All that needs to be done is to generate a call to the input method on the SUT object that the `@Action` method is related to. This only requires the provision of the parameter values for the method invocation as follows.

Specific procedures are generated to provide input parameter values for method invocations. Listing 1 shows these procedures as `Crequest_pX()`, in which `X` denotes the parameter index plus the name matching the input method. The procedures employ the invariant model (from Daikon) to generate possible values. For instance, `Crequest_p0` in Listing 1 is generated from the invariant "this parameter value is always to be found in the global array named `Clients`", i.e., a successful request is only possible for a connected client which is stored in the respective array (list) `Clients`. Further, according to observations from the traces, parameters 1 and 2 have always been constant, therefore `p1==1.0`. That way, reasonable input values can be generated according to prior observations deduced from the traces. Alternatively, if no suitable invariant is found, a null value or a random number will be generated. In that case, it is the responsibility of the user to refine these missing parts of the model manually. This example also illustrates the importance of separating nominal from exceptional behaviour in the model. If the categories are not separated, e.g. executions with exceptional values such as clients that are not connected, it would not be possible to infer this kind of invariants (thus leaving more manual work).

The test oracle element of the generated EFSM consists of two parts, the so-called *interaction oracle* and the *return value oracle*. The return value oracle is similar to parameter values in that they are based on the invariant model. Return values in our invariant model are described separately from global state and parameter values, which results in a set of simple invariants that are turned into JUnit assertions. For the `@Action` method in Listing 1, this can be written as `assertEquals("ok",rv5);` This assertion ensures that the correct value is returned by the SUT. In this case, Daikon has inferred that the return value should always be `"ok"`, because only nominal behaviour has been executed and traced. Automatic generation of non-primitive objects through invariants is currently not supported by the tools, so that the mapping to such more elaborate objects, currently, has to be done manually in order to augment the final model. In this particular case, the user would have to change the line of code under consideration in the model to `ReturnStatus.ok`, as will be discussed below in subsection 3.6.

Generation of the interaction oracle is based on the FSM and its mapping to the input-output interfaces. Before a call is made to the actual input method of the SUT, the expected interactions are defined with the help of mock objects that were generated earlier. The example in Listing 1 shows one such expectation in that a call to the input method `Crequest` causes a another call from the SUT to the output method of `Creply`. This is expressed as `expect(mockClientRcv2.Creply((AISMessage) anyObject())).andReturn("ok");` in the model as shown in the listing. Once again, the return value for this call by the SUT is inferred and must be refined by the user similar to the return value assertion. Similar expressions are deduced for all expected interactions, i.e., the output method calls. Finally, after making a call to the SUT input method, it is verified that correct interactions took place, and all mock objects are reset for the next transition.

Generation of guard methods is based on invariants for the global state variables according to the state or transition under consideration. For example, the guard method `Crequest_CreplyGuard` in Listing 1 is derived from the invariants of the global state if `Creply` has been executed after `Crequest`. For the `Creply` state alone, the guards would be based on all invocations of `Creply`, regardless of previous or next FSM states. Only the global state is considered for the guards as that is the only state available during their evaluation.

3.6 Refining the Model

The final element of completing the model is the manual refinement phase. This is a process in which the model is iteratively executed and refined to match the user's expectation of the model, according to observations coming from the implementation (either the model or the implementation have to be amended). This process is depicted as Step 3 in Figure **??**. Tests generated from the model are executed and asserted with the MBT tool against the implementation to assess the correctness of the model and the implementation.

```
...
@Action
public void Crequest_Creply()  throws Exception {
  this.state = "Crequest->Creply";
  expect(mockClientRcv2.Creply((AISMessage)anyObject()))
                              .andReturn(ReturnStatus.ok);
  replay(mockClientRcv2);
  ReturnStatus rv5 = aISMerger.Crequest(Crequest_p0(),
                                        Crequest_p1(),
                                        Crequest_p2());
  assertEquals(ReturnStatus.ok, rv5);
  verify(mockClientRcv2);
  EasyMock.reset(mockClientRcv2);
}

public boolean Crequest_CreplyGuard() {
  if(Clients.size() < 1) return false;
  if(Messages.size() < 1) return false;
  return true;
}

...

private String Crequest_p0() {
  return (String) randomItemFrom(Clients);
}

private int Crequest_p1() {
  AISMessage msg = (AISMessage) randomItemFrom(Messages);
  return msg.getUserID();
}

private byte Crequest_p2() {
  return (byte)1.0;
}

...

private String Cdisconnect_p0() {
  String client = (String) randomItemFrom(Clients);
  Clients.remove(client);
  Subscriptions.remove(client);
  return client;
}

...
```

Listing 2. Refined versions of methods in listing 1.

Finally, any inconsistencies found in either the SUT or the model during this execution are fixed. Since the model for any non-trivial SUT will likely contain a large number of states and transitions, we propose to iteratively process and check one issue at a time, in order to keep the model under control, and to facilitate the understanding of the tests and the implementation. Listing 2 shows a refined version of the generated model depicted in Listing 1.

For a stepwise approach of looking at one state at a time, the user progresses by enabling the guards for the `@Action` methods one at a time. When starting with the refinement, the model is in its initial state. From here, the process goes to transitions, i.e., the `@Action` methods, that can be taken once the initial state transition has been made. The sequence follows the nominal behaviour of the SUT as defined by its protocol. This process is repeated until all the states and transitions in the model have been enabled and all errors in test generation and assertion have been fixed either in the model or the implementation. Typically and quite likely, during this process, there will be a number of issues or errors reported, indicating either the requirement to refine the EFSM that was generated, or the requirement to amend the implementation of the SUT.

This can be seen as the most critical part of observation-based modeling, requiring careful consideration from the user. The model is based on observations of what the SUT is doing and not on what it is supposed to do according to a specification, i.e., the model that one would be looking for. Thus the user has to pay specific attention that it actually describes the correct behaviour.

Issues related to the model include the refinement of the interaction sequence because of multiple invocations that are not visible in the FSM, missing object or parameter value creation errors and missing updates to global state. The number of output method invocations can vary depending on various properties such as originating input method and global state. The model is currently generated to expect a single invocation of each output method only. During refinement it can turn out that a method can be invoked multiple times, which is manifested as an error in the model execution. In this case as in any model refinement, the user has to check whether this is indeed the (nominal) expected behaviour of the SUT and amend the model accordingly if it is. In this case, this change in the model is achieved through adding `.anyTime()` to the end of the expected call, relaxing the requirement of a single call only.

As discussed earlier, creating non-primitive objects and maintaining global state falls under the responsibility of the user. Listing 2 shows how the `"ok"` objects from Listing 1 have been amended to `ReturnStatus.ok` in order to create these non-primitive objects. For global state, the collection of objects to store the global state are provided, but maintaining the contents of these global state objects is the responsibility of the user. Listing 2 displays this in the `Cdisconnect_p0()` method in terms of the statements `Clients.remove(client)` and `Subscriptions.remove(client)`, which means that once a client is disconnected, it should be removed from the list of connected and subscribed clients. These are concrete examples of domain knowledge necessary to apply our OBM method successfully.

Any errors in the model will be reported to the user as failures during the execution of the model, along with the line number and error message of the failed assertion. This helps in making the model refinement process easier and faster.

3.7 Debugging Errors

Sometimes the model will have residual defects that are difficult to identify, even though the model execution reports an issue. Such error could be located in the model, or in the SUT. An effective means of debugging and identifying such errors is to take the failing test case that has been automatically generated by ModelJUnit and turn it into a separate JUnit test case on its own and execute it. This will reveal all the hidden assumptions in data generation, interactions and similar properties, and it will allow the user to experiment with different settings of the test case. Through analysis of the results and comparison with the SUT specification, the fault can be located more easily. Currently, these tests have to be created manually, but the information required to automate their generation is already available in the ModelJUnit test case and with this information, this type of debugging support could be built into the testing environment.

4. Merger Case Study and Experiences

Goal of this experiment was the evaluation of the previously described OBM method with the target of generating suitable test cases for the Merger component.

4.1 Applying the Method

In order to capture a suitable initial trace for `Merger`, we used seven tests executing the component in situ of the encompassing system and having it process actual AIS field data. One of the tests consisted in running the entire system of 25 components over the course of 5 minutes of real AIS data, summing up to around 20,000 AIS messages, and about 60,000 recorded events of the system such

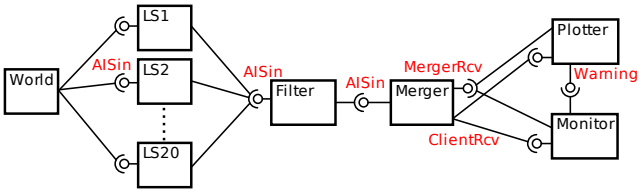

Figure 2. Architecture of the surveillance system used as example.

as invocations to `Merger`. This somewhat more elaborate test was used to capture the general interactions of `Merger` for building the FSM and capturing data for the invariant model. The other, less extensive tests were applied to extend the FSM of `Merger` with less common states and transitions. Detailed discussion on different types of tests and their coverage can be found in our earlier work [12].

AspectJ[7] was used to instrument the system to produce the required traces, including logs for every method invocation, return values, parameter values, and the global state of the component at the time an event occurred. Adequate logging formats for PROM and Daikon were provided through our `Tracer` component described earlier. The `Merger` component was extended with a test interface exhibiting its global state, including the list of connected clients, the list of subscriptions, and a list of AIS messages that it received. A more general approach for tracing could have been achieved through the message passing mechanisms of the used middleware. More discussion on the required testability and execution tracing infrastructure can be found in [11]. After capturing the traces, the initial FSM, shown in Figure 1, could be evaluated in PROM to check for compliance with the original specification. The EFSM model for ModelJUnit was generated by feeding the trace log files to the model generator component. After the final step of refining the generated model as described earlier lead to the final EFSM displayed in Figure 3. This model is a representation created by ModelJUnit, which can also be used during the model refinement process to get a view of the EFSM at any time.

In the following subsections, we summarize our experiences in performing these experiments.

4.2 Errors Discovered

The generated model must be refined manually to get the final model. In this process, errors were found in the generated model such as missing object creation, and global state updates, but also in its application against the SUT. The generated guard methods based on Daikon invariants were also overly strict, and all of them had to be amended.

A number of errors were found in the implementation of the `Merger` component, i.e., mismatches between implementation and specification, ambiguities, and problems in the design that cause errors under certain conditions. Problems in the system design were related to assumptions the SUT made about its environment. In one instance it made the assumption that one client component would never have several connections at the same time with the `Merger` component. Although the specification did not forbid such case, and the EFSM did test for this, the SUT developers considered supporting such corner case would make the implementation much more complicated without real advantages. However, this assumption and the resulting optimization also produced a tight coupling with the used middleware, which needed to be undone later, in order to port the component to another middleware. These findings provided useful insights into the implementation.

Mismatches identified between implementation and model were wrong return values, discovered by the return value assert oracles, and incorrect or missing transitions discovered by the mock object interaction oracles or by inspection of the FSM vs. the specification. In the case of return values, making a connection would always return "OK", regardless of the parameter provided, and whether a connection was successful or not. This was a clear violation of the specification, which states that `Merger` should return error codes.

Another issue detected was a missing specification item, about queries on ships that do not exist. The generated model issued an "OK", but the implementation returned an error code. The specification made no statement how this should be handled by `Merger`. In this case, this highlighted a need to update the specification and then re-evaluate the model vs. the implementation.

Protocol issues of the `Merger` were discovered through the interaction oracles and through inspection of the FSM. The specification states that "if a client is subscribed to a ship for which data exists, the Merger should immediately publish this data to the client," which it did not, initially, and this case was also not present in the trace, nor in the generated model. This problem was found when comparing the model and specification and could be verified by inspecting the FSM's shown in Figures 1 and 3.

The subscription code contained another problem that was discovered by the interaction oracles. Subscribing to a ship for which no messages had been received so far, caused the loss of data through a missing output message. This was an error in the implementation.

These issues could be resolved, eventually, by amending the `Merger` code following the specification and the refined model. Overall, in terms of identifying previously unknown errors of a component that had been used for some time in this context, this can be regarded as a very successful model-based testing experiment with real value to the quality of the system.

4.3 Model Evolution

Once the errors had been amended, the next step was to re-generate the model in order to evaluate to which extent the existing model could be updated with this new information. A direct patch of the initially generated version of the model with the newly re-generated model failed, due to scalability problems with the Unix `patch`-tool. Instead, a visual `diff`-tool[8] was utilized to compare the two versions and copy the relevant changes, which was easy to perform. Finally, the model was further amended with additional error handling behaviour, initially not considered in the tracing and, thus, in the model. Error handling behaviour was incorporated for client connection failures and erroneous request ID values.

4.4 Discussion of the Experiment

Generating EFSM from traces and using them together with input- and output-method information is a feasible option to retrieve models for model-based testing. This subsection presents a brief critical evaluation of the process.

Using the Daikon invariant model in order to generate transition guard statements turned out to be difficult. None of the generated transition guards were usable as such. They could be used as mere basis for defining a guard statement, in particular, for identifying participating variables. Overall, they were found too restrictive and had to be amended.

Our presented OBM approach requires global state information of the SUT, either through built-in query interfaces, or by monitoring the SUT's environment while executing it. If global state information is not available, most of the techniques presented will be

[7] `http://www.aspectj.org`

[8] http://meld.sourceforge.net/

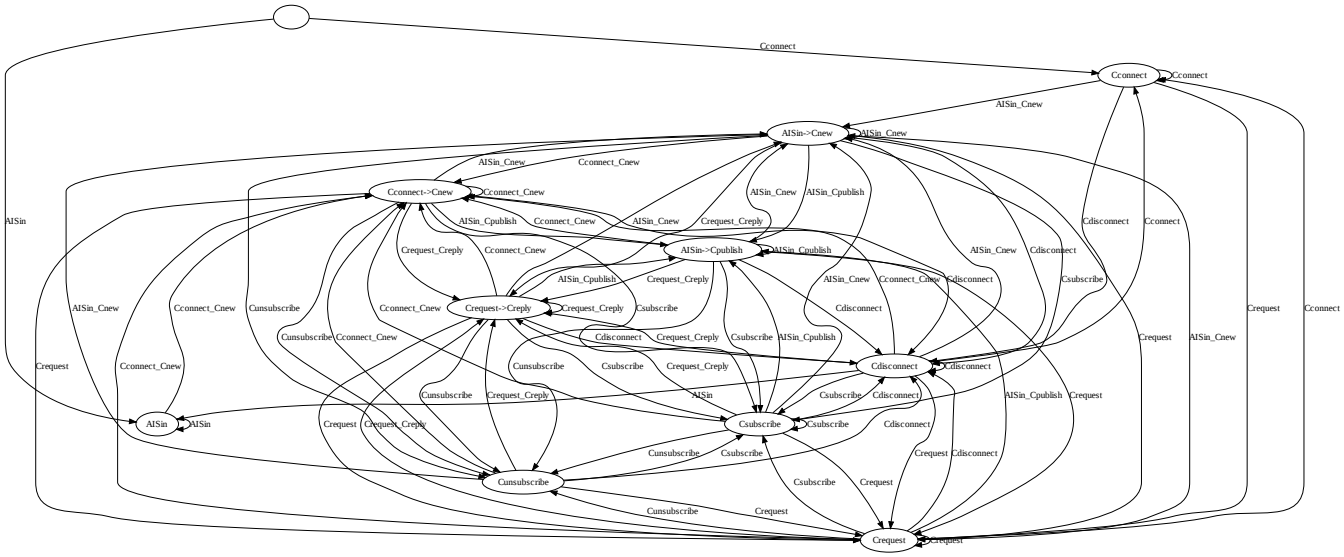

Figure 3. Final Merger EFSM produced by ModelJUnit.

applicable, such as capturing interactions at input and output interfaces, in order to come up with a rudimentary EFSM. However, the invariant model from Daikon will not be available, omitting information about return values, proposed basic guard statements, and parameter values. It would be sufficient to generate basic states and the test oracles, though, more elaborate modeling would have to be done manually by the user, which would decrease the value of this method considerably. This emphasizes the significance of having a built-in testability and testing infrastructure along the lines presented in [7] as part of the system architecture.

Bootstrapping and extending a full behavioural model based on runtime observations is challenging. Generating the most significant parts of the model by exercising test cases according to the nominal behaviour of the (informal) specification is straightforward. Any behaviour that is not stated in the specification, as discussed earlier, is difficult to discover. More elaborate testing techniques, including white box testing, and employing powerful search heuristics such as evolutionary algorithms [16] may be feasible solutions for behaviour exploration. However, this might run contrary to our initial requirement of using black-box testing only.

5. Conclusions and Future Work

This article describes a method and a collection of tools that help test engineers derive and refine behavioural models in a semi-automatic way to be used for model-based test generation. A rudimentary model is generated automatically based on observations from execution traces. This can be augmented with additional information derived according to further test executions, and assessed according to non-formal descriptions of the subject under test. Following this method makes the process of defining the models more akin to programming and code refinement, something that, in our opinion, is much more amenable to practitioners in industry, than writing formal specifications in an abstract mathematical notation, and it ensures seamless integration into a typical development and testing environment. Some of the tools employed are readily used in industry, in particular, the JUnit test framework is applied extensively in practice.

However, deriving models from observations also bears inherent dangers. That is, it is easy to generate and accept a model representing the behaviour of what an SUT does, rather than what it is supposed to do, thereby generating test cases assessing whether an SUT behaves the way it behaves, and not the way it *should* behave. Future work is, therefore, directed towards better support of the user in terms of (1) better behaviour exploration through more powerful search heuristics than random [3], (2) further automation of the debugging support, (3) improved invariant generation through relaxation of the Daikon rules along the lines described in [17], and (4) built-in tracing support by the middleware platform. Eventually, these improvements will not take the responsibility of the user to carefully refine and explore the behavioural state space of the SUT, but they will help to direct the user's attention to the essential tasks.

References

[1] A. Bertolino, A. Polini, P. Inverardi, and H. Muccini. Towards anti-model-based testing. In *Fast Abstract in The Int'l. Conf. on Dependable Systems and Networks, DSN 2004*, Florence, 2004.

[2] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006.

[3] M. d'Amorim, C. Pacheco, D. Marinov, T. Xie, and M. D. Ernst. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In *Proc. of the 21st Int'l. Conf. on Automated Software Engineering (ASE'06)*, pages 59–68, Tokyo, Japan, Sept. 2006.

[4] S. Ducasse, T. Girba, and R. Wuyts. Object-oriented legacy system trace-based logic testing. In *Proc. Conf. on Softw. Maintenance and Reengineering (CSME'06)*, pages 37–46, 2006.

[5] S. Elbaum and M. Diep. Profiling deployed software: Assessing strategies and testing opportunities. *IEEE Transactions on Softw. Eng.*, 31(4):312–327, Apr. 2005.

[6] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, Dec. 2007.

[7] H.-G. Gross. *Component-Based Software Testing with UML*. Springer, Heidelberg, 2005.

[8] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge. Recovering behavioral design models from execution traces. In *9th European Conf. on Software Maintenance and Reengineering (CSMR'2005)*,

pages 112–121, 2005.

[9] A. Hamou-Lhadj and T. C. Lethbridge. A survey of trace exploration tools and techniques. In *Proc. conf. of the Centre for Advanced Studies on Collaborative research (CASCON'04)*, pages 42–55, 2004.

[10] International Telecommunication Union. Recommendation ITU-R M.1371-1, 2001.

[11] T. Kanstrén. A study on design for testability in component-based embedded software. In *Proc. 6th Int'l. Conf. on Softw. Eng. Research, Management and Applications (SERA'08)*, pages 31–38, Prague, Czech Republic, 2008.

[12] T. Kanstrén. Towards a deeper understanding of test coverage. *J. Softw. Maint. Evol.*, 20(1):59–76, 2008.

[13] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *Proc. 30th Int'l. Conf. on Softw. Eng. (ICSE'08)*, pages 501–510, Leipzig, Germany, May 2008.

[14] L. Mariani and M. Pezzè. Dynamic detection of COTS component incompatibility. *IEEE Software*, 24(5):76–85, Sept. 2007.

[15] M. Marin, L. Moonen, and A. van Deursen. Fint: Tool support for aspect mining. In *13th Working Conf. on Reverse Eng. (WCRE'06)*, pages 299–300, 2006.

[16] P. McMinn. Search-based software test data generation: a survey. *Softw. Test., Verif. Reliab.*, 14(2):105–156, 2004.

[17] A. Mesbah and A. v. Deursen. Invariant-based automatic testing of ajax user interfaces. In *31st Int'l. Conf. on Softw. Eng. (ICSE'09)*, Vancouver, 2009.

[18] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One evaluation of model-based testing and its automation. In *Proc. 27th int'l. conf. on Softw. Eng.(ICSE'05)*, pages 392–401, 2005.

[19] B. Rumpe. Agile test-based modeling. In *Software Engineering Research and Practice*, pages 10–15, 2006.

[20] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 1 edition, 2006.

[21] W. M. P. van der Aalst, V. Rubin, H. M. W. Verbeek, B. F. van Dongen, E. Kindler, and C. W. Günther. Process mining: A two-step approach to balance between underfitting and overfitting. *Software and Systems Modeling (SoSyM)*, 2009.

[22] W. M. P. van der Aalst, B. F. van Dongen, C. W. Günther, R. S. Mans, A. K. A. de Medeiros, A.Rozinat, V. Rubin, M.Song, H. M. W. Verbeek, and A. J. M. M. Weijters. Prom 4.0: Comprehensive support for real process analysis. In *Application and Theory of Petri nets and Other Models of Concurrency 2007*, volume 4546, pages 484–494. Springer, Berlin, Germany, 2007.

[23] N. Walkinshaw, K. Bogdanov, S. Ali, and M. Holcombe. Automated discovery of state transitions and their functions in source code. *Softw. Test., Verif. Reliab.*, 18(2):99–121, 2008.

TUD-SERG-2009-012
ISSN 1872-5392