Delft University of Technology
Software Engineering Research Group
Technical Report Series

# Trace-Based Code Generation for Model-Based Testing

Teemu Kanstrén, Éric Piel and Hans-Gerhard Gross

Report TUD-SERG-2009-017


TUD-SERG-2009-017

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/

Note: Submitted for review at the Eighth International Conference on Generative Programming and Component Engineering

# Trace-Based Code Generation 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

Model-based testing can be a powerful means to generate test cases for the system under test. However, creating a useful model for model-based testing requires expertise in the (formal) modeling language of the used tool and the general concept of modeling the system under test for effective test generation. A commonly used modeling notation is to describe the model through an existing programming language.

This paper presents a technique to automatically generate an initial model describing the system from execution traces, using a common programming language notation. Turning this initial model into a full model to be used for model-based testing of the system under test then requires minimal effort compared to writing a model from scratch. This is illustrated by a case study application to a software component, which revealed real faults in its implementation.

***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 (SW), and it can be regarded as the ultimate review of a system's specification, design, and implementation [9]. 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. For this reason there is a strong incentive towards test automation solutions that make the testing process more effective.

One such solution is Model-Based Testing (MBT), which refers to modeling the System Under Test (SUT), using a formalized notation at a suitable abstraction level for what is being tested, and using an MBT tool to analyze these models and automatically generate tests based on them [19]. 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., models of the SUT that can be used for automated processing and test case generation, requires special expertise in the (formal) modeling language of the used MBT tool, and in the method of modeling the SUT for effective test generation. Even with the required expertise, creating and maintaining the models is a non-trivial task, making the cost-benefit trade-off hard to assess. These factors can significantly raise the threshold for industrial adoption of MBT tools and techniques.

This paper presents a technique to automatically generate a suitable model for MBT based on a set of captured execution traces of the SUT. This model describes the SUT as an Extended Finite State-Machine (EFSM), in which the SUT is described in terms of its external interfaces and global state attributes as a set of states, transitions between these states, and constraints describing when each transition can be taken. EFSM models are commonly used for behavioral modeling and model-based testing [19, 16]. With suitable traces as input, the technique can generate a complete model including the states, transitions, transition guards, test oracles to assess errors while executing the generated tests, and a test harness to connect the tests to the SUT implementation. Manual effort is still needed to refine the generated model, and in order to finalize certain properties of this model. Although, this work is minimal compared to writing the complete model from scratch.

Obviously, this method of trace-based model generation can only be "boot-strapped" from existing runtime scenarios, requiring an initial set of SUT executions to produce the input traces. It is, therefore, mainly suitable for software development projects for which an initial set of sample executions (e.g. test cases or field data) is readily available, such as integration of existing components and services, re-engineering/reconfiguration of systems, and the like. Because most typical software projects in practice exhibit such properties, the presented approach can be applied to most existing software projects, and offers a semi-automated way to constructing system specification models suitable for MBT.

The rest of the paper is structured as follows. Sect. 2 briefly outlines related work. Sect. 3 describes the tools and techniques used in our approach for model code generation, and how they were integrated to generate a behavioral model from execution traces semi-automatically. Sect. 4 discusses the presented approach and its limitations, presenting possibilities to address these limitations in future work. Finally, conclusions summarize the paper.

## 2. Background and Related Work

Many different techniques have been developed that generate state-based behavior models based on software execution traces. Existing work has also addressed the automated generation of test harness code to isolate a component from its environment, and the use of invariant-based models in test generation. This section gives an overview of previous research in these related areas.

Daikon[1] is an invariant inference engine used to infer likely invariants based on execution traces [8]. These invariants are described as likely invariants, as they hold for all the observations in the trace, which may or may not contain a representative sample

* 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.

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

of the SUT behavior. Example invariants include `x < 100` (value of x is always observed to be less than 100), and `x in Clients` (value of x is always observed to be included in the array Clients). Many trace analysis and model generation techniques make use of Daikon and invariants in modeling behavior. In general, the inferred invariants can be described as properties that hold at certain points of the SUT execution [8]. Invariants have also been proposed for testing [8, 14].

Test generation techniques based on program invariants include Agitator [4], Eclat [15] and the technique proposed by Xie and Notkin [22]. Each provides a tool that generates test input data, and based on the captured execution trace, presents a set of invariants describing the SUT behavior to the user. The user can analyze the proposed invariants to see if the SUT is working according to specification, and turn the invariants into assertions with related test input to form new test cases for the test suite. The approach presented in this paper makes use of similar assertions to define the state transition constraints that determine when a transition can be made, and to suggest possible test data values for the generated model. This is based on the execution trace and the relations of parameter values, return values, and global state values.

Lorenzoli et al. [11] model a system based on a captured trace including method invocations, parameter values, and global state. Similar to our approach, they use Finite State Machines (FSM) and Daikon-invariants to create the EFSM. These EFSM are used for test case selection and test suite optimization with the goal of increasing the coverage of the model. The approach presented in this paper uses similar means to generate the EFSM, but with different algorithms more suitable for MBT, and we also generate model source code from these models, whereas Lorenzoli et al. generate no tests nor code.

Mesbah and van Deursen [14] build an FSM for web-application user interfaces with the help of an automated crawler tool that is used to exercise the user interface and capture interaction sequences that cause changes in the interface's DOM tree representation. A change in the DOM tree constitutes a new state, and this information is used to model the FSM. Transitions are the clicks (input) to the SUT that caused these changes in the DOM tree. They use a set of their own invariants specifically built for web-applications to describe the expected changes in the DOM tree in response to input as test oracles. We also use an FSM and invariants as a basis for our model generation. However, our generated models are different. We target specifically MBT and generate the model code based on the EFSM. We also generate more specific oracles from the traces, whereas their focus is more on generic and user defined oracles.

Process mining is a technique developed to mine models for business processes from event logs [20]. Support for process mining has been implemented in a tool called ProM[2], which can produce various types of models, such as petri-nets and FSM [21] from the event logs. Process mining concepts have also been applied in the software testing domain, to help in validation of service-oriented applications [20]. We use the ProM tool to create the FSM model as a basis in our model generation tool.

In order to generate a model for MBT of a component, we must also generate the code that isolates it from its environment and verifies the correctness of its interactions with the environment. This is commonly achieved with the help of (component) test stubs that emulate the environment. When the stubs are made programmable, they are often referred to as mock objects [12]. This usually means that a component library provides interfaces to create these stubs, and that for each stub it is possible to define the expected interactions with the SUT and the values that should be returned in each case.

Tillmann and Schulte [18], and Saff et al. [17] provide means to automatically generate mock objects for the SUT. Tillmann and Schulte use static analysis (symbolic execution) and Saff et al. use dynamic analysis to capture the behavior expectations and return values for the mock objects. Both focus on one test at a time, to allow the generation of mock objects for exactly the purposes of this test. The test for which mock objects are defined is determined by factoring a larger test to smaller tests [17], or based on static analysis of code with symbolic execution [18].

A more specific test harness generation method for service-oriented mobile applications is presented by Bertolino et al. [3]. They assume the SUT is described using formal web-service description languages, such as WSDL and WS-Agreement. Based on these specifications, they generate test stubs for components with which the SUT is interacting. WSDL is used to define the stub interfaces, and WS-Agreement to define the expected behaviour of the SUT for the stubs. Additionally, using simulators, they generate data to test the SUT in different situations.

## 3. Trace-Based Model Code Generation

This section describes our approach of generating model code for MBT. We use the term Model-Based Testing similar to that of Utting and Legeard [19] who describe it as "Generation of test cases with oracles from a behavioral model". The model describes the expected behavior 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). In order to generate a usable model for MBT, we create all these parts from the execution trace.

The idea of turning the MBT approach around, and using execution traces from the implementation to provide the model was described by Bertolino et al. [2] as anti-model-based testing, although they never took it further than describing the concept. The produced model can then be used to verify the implementation against the specification and to generate additional tests for the SUT. This is what we do in our approach of generating the model code based on execution traces.

In our approach, the SUT is first exercised as guided by existing program execution definitions, according to a usage profile. Typical examples in the context of dynamic analysis, as we apply it, include existing test cases and example applications [6]. Useful input data for this can also be captured from the field data of actual application uses [7]. MBT is generally considered to be a black-box approach, based on the SUT's external interfaces and related specifications [19]. Similarly, our approach to model code generation is a black-box approach, and only data from the external interfaces of the components is captured.

Based on the captured traces, we produce the EFSM model. This process includes using ProM to generate the FSM, Daikon to generate the invariant model, combining these together to form the EFSM, and finally generating the model code from this EFSM in the format of the ModelJUnit[3] MBT tool. The generated model code includes the state transitions, guard constraints, test input, the test harness, and the test oracle. This model can be executed with

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

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

ModelJUnit to generate and execute tests in order to assess the implementation of the SUT.

Although, the process of generating the model code from the traces is completely automated, the generated model still requires some manual refinement. The user is required to have some knowledge of the SUT. Basically the user must know how to set up the SUT for testing, and how to create any non-primitive objects and data-structures it requires. This is due to limitations of the invariant model, which is used as a basis to generate input data for the model code. These limitations will be described in more detail in the following subsections. The user should also have a specification available to tell what is the correct expected behavior of the SUT, in order to be able to assess the correctness of the generated model.

We generate also the test oracle for the test model coming from the trace. This oracle is based on the assumption that the trace represents the correct behavior of the SUT. Excluding, for example, tests that exercise error revealing inputs on the SUT and validate its error handling functionality. This is important because the traces are used in generating the test oracles. When the trace includes erroneous inputs and behavior, it cannot be used as a basis for creating expectations for the test oracle to assess when the response to the model execution is correct. However, we realize that in practice it is not always possible to come up with a set of executions that exhibit this property of correctness perfectly, so this is not a strict requirement. Not having them, simply means that as more "error state"-related behavior will be represented in the traces, this leads to more refinement effort needing to be performed by the model engineer, in order to retrieve a good model, eventually.

The code generation process described in this paper makes use of several existing tools and techniques, including Daikon, ProM, ModelJUnit, JUnit[4] and EasyMock[5]. JUnit is a commonly used unit testing framework for Java, employed to execute test cases, and to provide the test oracle assertion language for verifying the correctness of output data received from the SUT. EasyMock is a mock-object-framework for Java. The approach presented here has been implemented in an automated tool and is available as open source[6].

### 3.1 ModelJUnit Notation

In order to provide required background information for the model code generation technique, this subsection presents the notation of the ModelJUnit tool for which the code is generated. Listing 1 shows a model for a simple vending machine in the ModelJUnit notation, adapted from [1]. This vending machine accepts 25 cent and 50 cent coins and allows the user to get the product when a total of 100 cents has been received. It does not allow inserting more than 100 cents, and once this limit is reached, the only action available is the "vend" action. When this is done, the machine goes back in the initial state and requires another 100 cents to vend. Figure 1 is an FSM visualization of the same model as provided by the ModelJUnit visualization support.

As shown in Listing 1, ModelJUnit uses the Java programming language notation for its models. The model code is a standard Java class implementing the `FsmModel` interface. This interface defines that the class must implement the `getState()` and `reset()` methods. The `getState()` method is used by ModelJUnit to query the current state of the model, and it uses this information as feedback to the test generation algorithms. The `reset()` method is invoked by ModelJUnit when it starts the generation of a new test case. Typically, several test cases are generated from a model, with a given goal, such as satisfying a chosen coverage criterion. The `reset()` method must set the model into its initial state for the next test case (transition sequence) that is to be generated.

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

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

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

```
public class VendingMachineModel implements FsmModel {
  private int money = 0;

  public Object getState() { return money; }

  public void reset(boolean b) { money = 0; }

  @Action public void vend() {money = 0;}
  public boolean vendGuard() {return money == 100;}

  @Action public void coin25() {money += 25;}
  public boolean coin25Guard() {return money <= 75;}

  @Action public void coin50() {money += 50;}
  public boolean coin50Guard() {return money <= 50;}
}
```

**Listing 1.** Example EFSM adapted from [1].

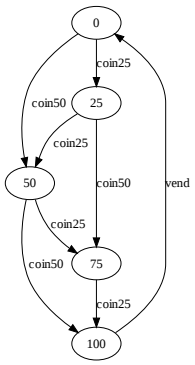


**Figure 1.** FSM for Listing 1.

A second part of the model is defined using Java annotations and naming conditions. This part defines the actual states, transitions and constraints for allowing the transitions to happen. As shown in Figure 1, the vending machine has five different states. These are the values returned by the `getState()` method. This value is updated by the transition methods, which in Listing 1 are `vend()`, `coin25()`, and `coin50()`. These symbolize the different state transitions of the state-machine, in this case, either inserting one of the allowed coins or using the vend-functionality. They all update the `money` state variable accordingly. All such transitions methods are identified by ModelJUnit through the `@Action` annotation.

Each transition is only allowed to happen when the amount of money inserted into the machine is below or above a certain sum. Vending is only possible when 100 cents are inserted. Inserting 25 or 50 cents is not possible when the total amount of money inserted would go over the maximum of 100. These constraints are defined in listing 1 in the methods `vendGuard()`, `coin25Guard()`, and `coin50Guard()`. ModelJUnit identifies and associates these constraint functions with their corresponding transitions by matching the method names. Each `@Action`-tagged transition method is expected to have a guard method with the same name but with `Guard` appended to the name. When this method returns true, the transition is permitted, and the related `@Action`-tagged method can be called. When the guard method returns false, the transition is not permitted, and the related `@Action`-tagged method is not called.

There are two essential elements missing from the model code in Listing 1. The transition methods update the global state of the model, but do not actually generate or execute any tests for these updates. For example, in case of online testing, the `coin25()` tran-

sition method could make a call to the actual SUT (assumed here to be represented by an object name `sut`) as `sut.insert25()`. This would also make the model state transition manifest in the SUT, causing in effect a test step to be executed. We call this the test harness, as it binds the MBT tool to the SUT. Another option would be to print out test scripts for an external tool, but we concentrate on online testing in this paper.

The second element missing from Listing 1 is the test oracle. This is the part that assesses whether the SUT provides the correct response to the input provided by the MBT tool. For example, in the `coin25()` transition method this could be a simple assertion `assert(money == sut.getInsertedCoins())` statement inserted after the `money += 25` statement that updates the model state, to verify the (expected) state in the model vs the (actual) state of the implementation. These parts are further illustrated in our case example in the following subsections.

### 3.2 Case Example

To illustrate our model generation approach, throughout the rest of the paper we show its application to one of the components (`Merger`) of a maritime surveillance system. This 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 behavior and communication protocols of its components. The components are implemented in Java specifically crafted to be executed under Fractal [5], a component middleware platform.

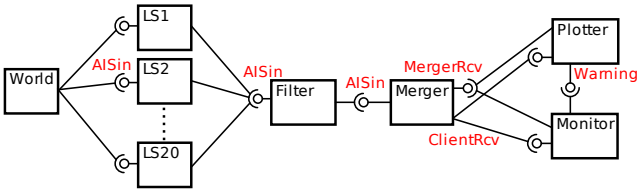


**Figure 2.** Architecture of the surveillance system used as example.

The `Merger` component was selected as SUT because it exhibits complex interaction with the other components. It acts as a temporary database of AIS messages, and client components can consult it to track information for 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 center. `Merger` was used as target component for experiments evaluating the model code generation approach. The goal is to generate a suitable test model for `Merger`, and to validate the implementation against the specification by executing the model produced and generating test cases based on it. When we applied the technique during the case study presented here, we were able to expose five previously unknown faults in the SUT and one ambiguity in the specification. This demonstrates the usefulness of our approach in practice. However, in this paper we focus on describing the code generation process and related properties, and do not discuss the details of the discovered faults or other properties not related to model generation.

As mentioned earlier, the trace used for model code generation is based on the external interface and the global state of the SUT. Different systems can have various kinds of external interfaces, and capturing the passed messages and representing them may require different approaches. In our case, the SUT external interfaces are represented by Java interfaces, implemented by `Merger` according to the Fractal middleware. Captured data related to its external interfaces includes method calls made into `Merger`, made by other components (its input interactions), and calls to other components made by `Merger` (its output interactions). Captured data related to its global state are lists of connected clients, subscribed clients, and received AIS messages. The global state is accessed by a test data interface implemented by `Merger`, although the same information could be constructed to provide a "mock" global state at the same time as its external interfaces are monitored, as the state is modified through these messages passed through the component external interfaces. This information is available in the component specification. Although, our tracing mechanism is tuned for the needs of this case study, we believe it can be adjusted to most other systems with relative ease, as most software components and systems provide external interfaces with messaging mechanisms.

The completeness of the provided trace is also important. It determines the completeness of the generated model. To start with, we run the complete system shown in Figure 2, with about 20000 AIS messages, captured from actual ship interactions. This produces the FSM as visualized by ProM in Figure 3. Since this is not a complete description of the SUT with respect to its non-formal specification, it is augmented with six additional stimuli, producing a more elaborate FSM (shown in Figure 4). Combining all executions into one single set, the final FSM is generated (shown in Figure 5). This new FSM is considered a good and representative set of executions for generating the code of the model.

The huge set of field data, passed through the system, is found sufficient as basis for building the invariant model. However, the specific part of the system, which is only exposed through the six previously mentioned additional test stimuli, is not well covered through field data. This results in fewer, and less useful invariants for this parts of the system. Since, here, our focus is on model generation, and not on input data generation, there is no requirement for crafting further test cases and input data as input stimuli. With the resulting test set we can proceed to the generation of the model code for the SUT under consideration, in the next sub-section.

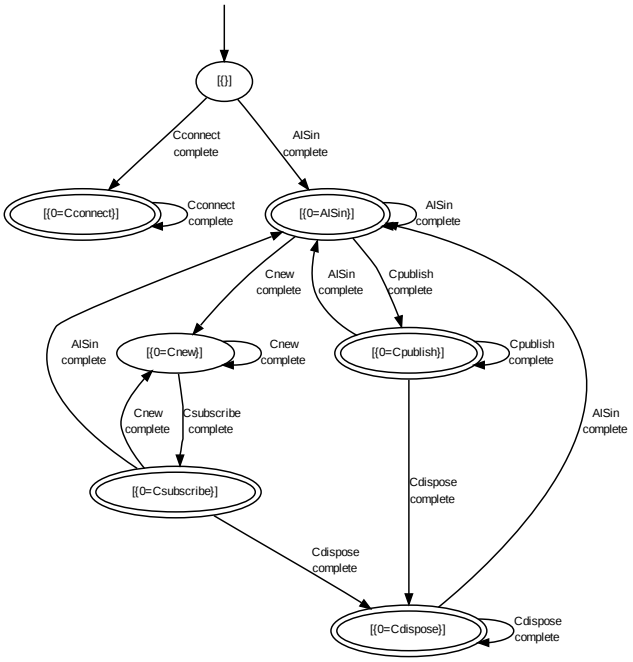


**Figure 3.** Merger FSM produced by ProM for the field data.

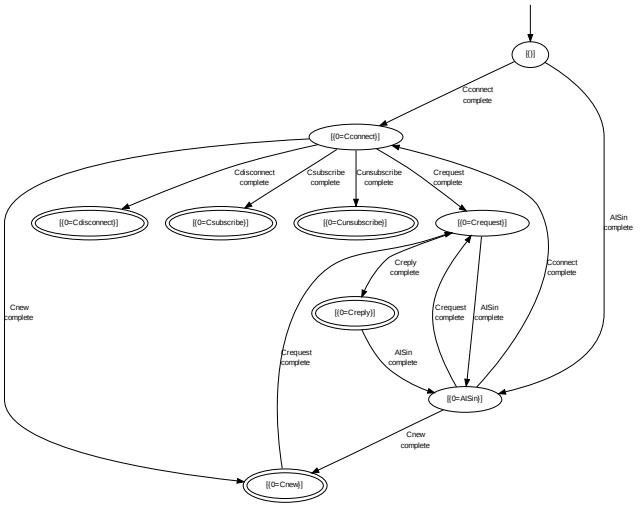


**Figure 4.** Merger FSM produced by ProM for the focused tests.

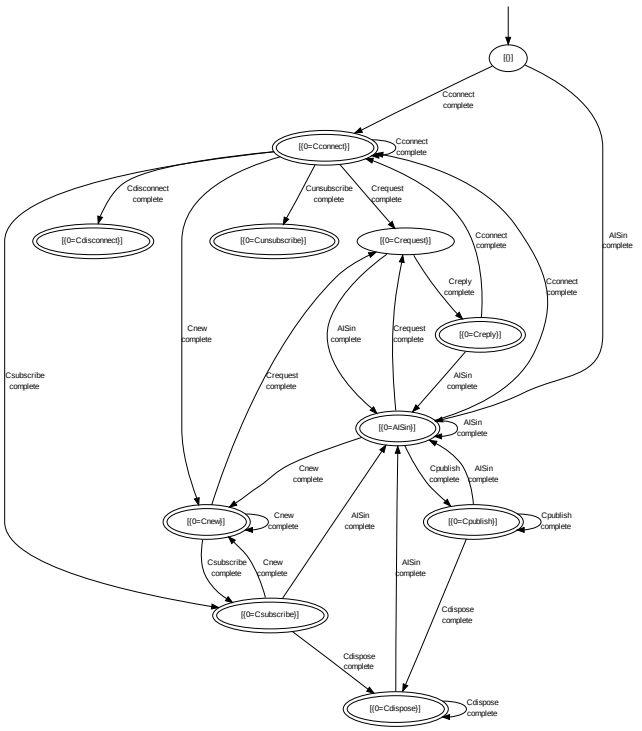


**Figure 5.** Combined Merger FSM produced by ProM.

### 3.3 Transforming the FSM into MBT model code

The execution trace of the SUT is transformed into an FSM with ProM's transition system miner component (described in detail in [20]). The tool can be used to visualize the trace in the form of an FSM. It can also be used as an algorithm library to create and access the FSM in terms of a set of data structures, bypassing the GUI. These data structures represent the basis for the code generation from the FSM. Here, an essential concept is the *state*.

The execution trace is based on input- and output-method invocations, made through the SUT's external interfaces. The FSM describes the SUT in terms of these method calls, where each message passed through one of the interfaces matches a state in the FSM. However, the states of the FSM cannot be used directly to describe the states in the generated EFSM. Instead, the differences between the input- and output-methods comprised in the trace have to be considered.

The ModelJUnit EFSM notation describes the SUT in terms of state transitions. We consider a state transition to be triggered by an invocation of an input-method to the SUT. Thus, for each input-method in the FSM, a matching `@Action`-method is generated in the model code. This is illustrated in Listing 2, showing examples of generated `reset()`, `@Action` transition, and transition guard methods for the `Merger` component. Here, the `Crequest` is an input message for the `Merger`, and thus it has its own state transition generated in the model code.

The basic components generated for each `@Action` transition method are also shown in Listing 2 for `Crequest`. The transition starts by setting the state of the model to a name matching the taken transition (`this.state = "Crequest"`). This allows the MBT tool to use its model coverage algorithms to cover different combinations of the interaction sequences. The second line in each generated transition method always prints out the name of the state transition taken (`System.out.println("CREQUEST")`) in order to make it easier to follow the paths that the MBT tool takes while it generates tests from the model. This is especially useful for debugging errors that it discovers.

The next step is the generation of the expected interactions within a transition. They are generated based on the FSM and the categorization of each message in the component interface into an input or output message. This classification is provided by the user, through determining the names of the Java classes defining the input and output message. These definitions (from the class files) are automatically parsed to create a list of messages (method names) that belong to either input- or output-interfaces. The FSM is then analyzed with this information and each input-state (message) is associated with outgoing transitions to any output-states (messages).

For each input message in the FSM, a number of `@Action` transition methods are generated. One for the input message alone, and one for each possible output message to which it has an outgoing transition. For example, the combined FSM has a state Crequest, which can either go to Creply or to AISin. As only Creply has been classified as an output message, we obtain in the Listing 2 two `@Action` methods: `Crequest()` corresponding to the input message itself, and `Crequest_Creply()` corresponding to the input message followed by a Creply output message.

In the `@Action` transition methods that include an expected output transition from the input transition, also expectations for this output transition are generated. This makes use of mock objects generated with EasyMock. An example is the `Crequest_Creply()` method. The EasyMock mock object framework is used to set up and verify the expected interactions with the SUT. All mock objects are automatically generated in the JUnit test method, which is used to execute this test through the JUnit testing framework. It also allows integration of the model execution with most modern IDEs, as they provide JUnit integration and means to report and analyze the test results. The generation is shown in Listing 2 in the `modelJUnitTest()` method, which is always the name of the generated JUnit test execution method. It starts with creating mock objects for the model, in this case `mockClientRcv2`. These are stored globally in the model to allow for all the transition methods to access them. The generated mock object is called `mockClientRcv2` according to the related output interface to which it belongs (ClientRcv2). In the `Crequest_Creply()` method the expectations for the output method interaction are set as `expect(mockClientRcv2.Creply((AISMessage)`

```
@Test public void modelJUnitTest() throws Exception {
  mockClientRcv2 = createMock(ClientRcv2.class);
  Tester tester = new RandomTester(this);
  ...
}
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() throws Exception {
  this.state = "Crequest";
  System.out.println("CREQUEST");
  replay(mockClientRcv2);
  ReturnStatus rv4 = aISMerger.Crequest(Crequest_p0(),
                                        Crequest_p1(),
                                        Crequest_p2());
  assertEquals("ok", rv4);
  verify(mockClientRcv2);
  EasyMock.reset(mockClientRcv2);
}
public boolean CrequestGuard() {
  if(SubscriptionsIsNotEmpty()) return false;
  if(MessagesAreDifferentFrom_ship2_()) return false;
  if(ClientsSizeDoesNotEqual1()) return false;
  if(MessagesSizeDoesNotEquals_0_1()) return false;
  return true;
}
@Action public void Crequest_Creply() throws Exception {
  this.state = "Crequest->Creply";
  System.out.println("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(SubscriptionsIsNotEmpty()) return false;
  if(MessagesSizeDoesNotEqual1()) 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 AISMessage AISin_p0() {
  return null;
}
...
private String Cdisconnect_p0() {
  return (String) randomItemFrom(Clients);
}
```

**Listing 2.** Generated reset method and sample transition (@Action), guard and parameter value generation methods for Merger.

```
=====================================================
Crequest:::ENTER
shipID?1 == AISType?2
shipID?1 == size(Clients?g[])
size(Clients?g[])-1 == size(Subscriptions?g[])
clientName?0 == "myclient"
shipID?1 == 1
Clients?g[] == [myclient]
Clients?g[] elements == "myclient"
Subscriptions?g[] == []
Messages?g[] elements == "ship2"
Messages?g[] one of { [], [ship2] }
size(Clients?g[]) == 1
size(Messages?g[]) one of { 0, 1 }
clientName?0 in Clients?g[]
=====================================================
Crequest_EXIT:::ENTER
ReturnStatus?r == "ok"
=====================================================
```

**Listing 3.** Sample Daikon output for Crequest.

`anyObject())).andReturn("ok");`. This means that EasyMock will expect the SUT (`Merger`) object to call the `Creply()` method of `mockClientRcv2` with any parameter of type `AISMessage`, and when this happens the mock object should return the value `"ok"` to the SUT. Generation of the return values will be discussed in more detail in the subsection 3.4.

Once the expectations are set, a call is made to the input method of the SUT that corresponds to the state transition method being executed. In the case of `Crequest_Creply()` it is the `Crequest()` method. To provide parameter values for these method invocations, template methods are generated which the user will have to fill. The types for the return value from the input method and for the input method parameters are parsed from the input-interface class files.

Finally the results are verified, i.e., the test oracles are invoked. For interactions, this is always of the form `verify(mockClientRcv2)`, with the name of the mock object replaced with the correct name. This causes the EasyMock framework to verify that all the expectations set for the mock object are met, and no additional extra interactions are performed. The return value oracles are discussed in the next subsection.

### 3.4 Transforming the invariants into model code for MBT

The second model used in the generation of the EFSM code is the set of invariants provided by Daikon. These invariants describe the properties of the parameters and return values of the input- and output-interface method invocations for the SUT, as well as their relations to the global states of the SUT. They are used to generate possible return values for the mocked output message sequences, parameter values for input messages, and guard conditions for transitions. Daikon can output the invariant information in many different formats for testing, and also in the form of Java assertions that check whether the invariants hold [8]. However, none of these formats is directly usable for our purpose. Therefore, we use the basic textual output, parse it, and generate code out of it. An example of this output for `Crequest` is shown in Listing 3.

Further, Listing 3 displays how we can identify which invariants are related to global state, and which are related to return values or parameter values. The postfix of the variable name represents the identifier: `?g` refers to a global state-related value, `?N`, with N as a positive integer, refers to a parameter value at the given index for a method, and `?r` refers to a return value. This formatting is automated in our trace component when it produces the Daikon input.

In addition, Listing 3 displays how we handle return values differently from parameter values (postfix `_EXIT` of the method name

Crequest_EXIT). This is simply a naming convention used to allow separation of these values according to how we use Daikon. Associating the invariant values to related parts of the FSM is based on the method names in both the Daikon invariant output and in the FSM. In case of Crequest in Listing 3, invariants related to global state are used to generate the guard method for Crequest state, the return value to create return value oracles and mock object return values, and parameter invariants to generate the parameter values as identified by the postfix in the variable name.

We do not generate non-primitive objects automatically, as it is not possible to know how the primitive values in the invariant model have to be mapped to previously unknown objects and their constructors. Instead, where such objects are needed, the value in the invariant model is provided to the user as a basis for manual refinement. In Listing 2 this is shown as "ok" in both transition methods Crequest() and Crequest_Creply(), where rv4 holds the return value for Crequest(), and this value is verified with assertEquals("ok", rv4). The same applies to the return value given to the SUT when it invokes the mockClientRcv2 mock object in Crequest_Creply(), which is shown as .andReturn("ok") in Listing 2. This "ok" must be changed to create an actual domain object matching this invariant value (shown in section 3.5 on model refinement). This illustrates the domain knowledge required of the user for creating non-primitive objects. The values provided, when matching more complex objects, have been defined by the person who created the instrumentation, and, thus, the representation of these objects in the trace. The user should also know how to turn the provided data into objects, e.g., in this case "ok". In addition to providing a single value, the code generator supports provision of several value options, and value ranges, based on the Daikon output. These are generated into a matching test oracle assertion for the return value, through comparison with a list of allowed values, or value ranges.

The second part of the generated model (based on the invariants) concerns the guard conditions. For example, the guard of the Crequest_Creply() transition method is the Crequest_CreplyGuard() method. It comprises four constraints on the Crequest_Creply() transition, each one being a call to a method generated at the end of the model code (Listing 2).

Examples are shown in Listing 4. SubscriptionsIsNotEmpty() matches the invariant Subscriptions?g[] == [] in Listing 3. MessagesAreDifferentFrom_ship2_() matches the invariant Messages?g[] elements == "ship2" in Listing 3. These names are generated in a way that they are human-readable, e.g., if(SubscriptionsIsNotEmpty()) return false in listing 2 means that "if the subscriptions list is not empty this transition is not allowed".

Since each constraint is modeled as a separate object in our code generation tool, each constraint objects also contains a specially crafted template for the related invariant. This template describes how to use the provided parameter name and invariant values in a combination to create a human-readable expression. The guard conditions are based on the global state only, as this is the only state available during the evaluation of the guard constraints (execution of the transition guard methods). In order to make the generated code more user readable, additional helper methods are generated in the model, e.g., randomItemFrom() method in Crequest_p0() and Cdisconnect_p0() methods in Listing 2. Making sure the generated code is human-readable permits to ease the manual refinement process that will be described in the following subsection. The easier it is to understand the code, the easier it is to compare it to the specification, and to modify it as needed.

```
...
public boolean SubscriptionsIsNotEmpty() {
  Collection requiredValues = new ArrayList();
  if (Subscriptions.equals(requiredValues)) {
    return false;
  }
  return true;
}
...
public boolean MessagesAreDifferentFrom_ship2_() {
  Object expected = "ship2";
  for (Object o : Messages) {
    if (expected.equals(o)) {
      return false;
    }
  }
  return true;
}
...
```

**Listing 4.** Sample generated invariant checks for Listing 3.

The generation of guards out of the invariants has, however, one major weakness: it often leads to overly constraining guards. The previous examples illustrate well this drawback:

- The guards check specifically for myclient (Listing 3) and ship2 (Listing 2) are derived from using those objects in the focused unit tests that produced the trace, although they are actually entirely irrelevant for the transition.
- The requirement for "no subscriptions" comes from the fact that the request/reply functionality is executed with a set of focused tests that never subscribe (since subscribing is not required for this particular functionality). It is irrelevant whether there are subscriptions or not.

Similarly, the (test) execution applied to produce the trace made only one single connection to the client in order to exercise the request/reply functionality. Consequently, a guard was generated stating that there must always be exactly one single client connected (ClientsSizeDoesNotEqual1, as this is the suggested setting from the executions. However, according to the specification there can be any number of clients connected, as long as there is at least one.

These examples show that the invariants for guard conditions can be overly constraining but, nevertheless, they still provide useful information for the creation of the correct guard constraints. In most cases, they grasp the correct variables influencing the transitions, and the correct type of test but they are too pessimistic. Therefore the generated guards can be considered as a good starting point for manual refinement.

The global state variables are identified in the trace itself by the used identifiers as described earlier. However, we do not process the trace but the invariant model that is the output of Daikon, and the FSM model that is the output of ProM. The global state variables for the EFSM are then based on the Daikon output as this includes the required information to infer the names and types of global state variables that should be in the generated model. When an invariant related to a global state variable refers to a primitive value, a similar variable is generated but with a suitable data type to match the value in Daikon output. For example, if the value is numeric, an integer variable is generated to describe it in the global state of the generated EFSM model. In case the global state refers to an array value, such as Subscriptions?g[] == [] in Listing 3, a Java List object is generated for this invariant to represent state in the model, using the name that it is being referred to in the trace. These can be identified in the Daikon output by the [] notation it uses both in its input and output. In the generated reset() method, all

state variables are cleared for the next test to be generated. The part related to adding and removing objects from the generated `List` objects is not updated automatically. This is up to the user to refine the model to maintain the global model state while also creating and destroying all non-primitive objects.

The final part of the generated code coming from invariants are the parameter values for the calls to the input methods of the SUT. These are illustrated in Listing 2 by `Crequest_p0()`, `Crequest_p1()`, `Crequest_p2()`, and `Cdisconnect_p0()`. Their provided values are similar to those provided for the return values, with the exception that for return values, only the return values are considered, whereas for parameters the parameter values are considered plus their relations to global state. `Crequest_p1()` and `Crequest_p2()` show the creation of parameter values for the `Crequest()` input method. Here, the invariant model has determined this to be a constant value, i.e., `1`. For `Crequest_p0()` and `Cdisconnect_p0()`, the invariants suggest that the value should always be coming from the global list of state variables, i.e., `Clients`. In this case, the generated model chooses one item randomly from this list. This code has been generated fully automatically based on the invariants.

The requirement to describe "correct" SUT behavior for providing a basis for test oracles was noted earlier. The parameter values generated from the invariants are another reason for only considering nominal behavior during tracing of the SUT execution. As long as there are no error producing inputs, the oracles can be generated with correct expectations, and the parameter value generation is more powerful. If also erroneous input is included in the trace, the invariant model would not be able to determine that a value should be from the `Clients` list, or that it should be a constant of `1`, for example. The trace would contain other values as well, as this would be a property of the erroneous input, which would make both the oracle generation, as discussed, and parameter generation less powerful.

The generated code starts with the required model code for ModelJUnit, the transition methods and their related guard methods. Then follow the template methods to be filled by the user: for creating the SUT object, and for creating parameter values. Finally, the guard condition checking methods are generated, which often have to be refined by the user. Separating the parts that will likely require manual effort helps the user concentrate on where his expertise is needed. This also ease the patching process used to keep the model up to date when the SUT evolves.

```
...
@Action public void Crequest_Creply() throws Exception {
  this.state = "Crequest->Creply";
  expect(mockClientRcv2.Creply((AISMessage)anyObject()))
                       .andReturn(ReturnStatus.ok).anyTimes();
  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;
}
...
long msgTime = 0;
int nextMsgId = 1;
private AISMessage AISin_p0() {
  AISMessage message = new AISMessage((byte) 1, 0,
                              nextMsgId, new Date(msgTime));
  nextMsgId++;
  msgTime += 1000;
  Messages.add(message);
  return message;
}
...
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 5.** Refined versions of methods in listing 2.

### 3.5 Refining the Model

Once the model code has been generated, it has to be amended manually and validated. In this case, the generated model code is executed with the MBT tool and any errors found are fixed in either the SUT or the model. In our experience, this step works best when the states are enabled one at a time, because it helps the user focus on any issues related to given state. In such iterative approach, it is possible to keep the model complexity under control, and find the causes of failures more easily.

Properties of the model that require manual refinement include multiplicity of invocation sequences not visible in the FSM, creation of non-primitive objects, updates to global state and updating the transition guard methods. These were already briefly discussed in previous sections and in this section we illustrate these properties using the code shown in Listing 2 as an example. Listing 5 shows a refined version of this model code.

On default, the generated model for an input method expects that a single message is received for each expected output. This is a restriction of using an FSM as a basis for generating these expectations. From a given state, the FSM only expresses the possible transitions, but not how often they may be invoked. For example, the transition method `Crequest_Creply()` in Listing 2 may provide several replies when it has received several messages matching the requested criteria. However, based on the FSM, only one reply can be expected. Therefore, the model has to be refined manually for all the reply messages to be delivered correctly. In Listing 5 this has been amended through adding `.anyTimes()` to the end of the EasyMock expectation for the `Crequest_Creply()` transition.

A second limitation due to the FSM is related to one input message producing a varying number of output messages. Figure 6 shows an example FSM for this issue. There are two input messages, A and D, and two output messages, B and C. We know that C can happen after B, and that B can happen after both A and D. However, since the FSM has now abstracted the trace to this level, we do not know if these interactions happen in this order always, sometimes, or never. It may be that after A, both B and C always follow. Similarly, it may be that after D, only B always follows but C never does. Since it is not possible to make this distinction from the FSM, all the possibilities are generated. Thus the generated model code will contain states `A`, `A_B`, `A_B_C`, `D`, `D_B`, and `D_B_C`. However, in the minimal case, it should actually only

contain states `A_B_C`, and `D_B`. The other states have to be deleted manually from the generated code. This will be highlighted by executing the model and observing the specification, which will show these states as "unreachable". This helps in amending the model, but requires additional effort.

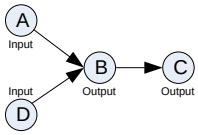


**Figure 6.** Example FSM.

The issues of creating non-primitive objects and updating global state are closely related. These are illustrated in Listing 5 in the parameter value creation methods `AISin_p0()` and `Cdisconnect_p0()`. The `AISin_p0()` is responsible for creating new AISMessage (domain) objects that the `Merger` component processes. It shows how the user must understand the domain concepts and know how to create the domain objects for the SUT. In this case, each of the messages needs a unique id value and a unique timestamp. As they are created and passed to the SUT implementation, the model's internal global state must also be updated to match the implementation: the created object is added to the `Messages` global state list. The `Cdisconnect_p0()` shows an example of when an object needs to be removed from the model global state in order to match the expected global state of the implementation. All these modifications must be carried out by the user. The global state lists are automatically generated in the model, i.e., `Messages`, `Clients`, and `Subscriptions`.

Examples of simpler refinements for non-primitive objects are the changes from `"ok"` to `ReturnStatus.ok` (shown in Listing 2 and Listing 5). In this case, it is a straight mapping to an enumeration with the name of the invariant value. An example of refining the generation of a primitive value is shown in Listing 5, where the `Crequest_p1()` method must return an id value which has been received previously by the SUT. It must be refined manually to take one of the messages from the global state list object `Messages` and return the id values of this message. This typically happens in case the value must depend on the global state. Since Daikon works with a set of primitive values from a trace, it does not usually infer this type of invariants related to composite objects, unless the trace is especially crafted for this, which it is normally not.

The generated transition guard methods need to be completely re-written for the final version of the model. As discussed earlier, the generated versions are overly strict because they are based on the invariants provided. For example, the guard method `Crequest_CreplyGuard()` shown in Listing 2 has been completely modified in the refined version shown in Listing 5. The original four generated ones have been changed to two specific ones. Comparing the generated and the refined versions, it is easy to see that the basic information is there in the generated copy, but it is not useful as such. Thus it provides good basis for refinement but is not usable without changes.

The refinements described in this section will be "requested" by the MBT tool while it executes the model against the implementation: any problem with matching the two together gives an error message. The user can then compare the error condition message with the specification, and make a decision to refine the model with the required fix, or to fix the implementation if it proves to contain a bug. In any case, these errors in the model vs. the implementation become apparent when the model is executed, and there is no need to do heavyweight manual inspections of the model to find them.

## 4. Discussion

In this section we discuss the weak points of our approach and how they could be addressed in future work. The discussion is mainly focused on the weaknesses of the code generation based on separate invariant and FSM models, as described in the earlier sections.

As raised in previous sections, the transition guard method generation based on the invariants provided by Daikon is not very effective. It is useful in providing insight into which variable affects the transition and should be considered in the guard statements, as well as their values. However, many excess conditions are generated, and even the ones hinting to the correct guard statements are overly strict and need to be analyzed and refined. We believe it would be more useful to focus on a few key invariants, make them more specific, and tune them specifically for our purposes. For example, when global state is represented in the form of a list, and the size of the list is either 1 or 3 in the trace, one option would be to make a more optimistic assumption and generate a guard statement to require that this list always contains some items (`size > 0`). However, identifying a good set of candidates and making them more specific for this purpose, would require more extensive studies with various components, state representations and input data sets.

Currently, global state updates have to be added manually to the generated model. However, invariant detection could be extended to automatically cover both pre- and post-conditions in the form of providing invariants over the global state, both before and after a message is processed by the SUT. For nor now, we focus on the pre-conditions, which means that, for example, it is not possible to infer an invariant stating whether a parameter value should become a part of a global state list variable, after a state transition. Daikon already provides some support for invariants based on pre- and post-conditions, but as it is based on assumptions of having a white-box trace of same values available both before and after each method call, we did not find effective means to tailor it to our black-box process.

Concerning the FSM code generation, a set of issues was described in relation to multiplicity of state transitions and the abstraction provided by the FSM, as illustrated in Figure 6. The use of a separate step for the FSM creation leads to the generation of too many and too few assumptions with regards to how the interactions are presented by the trace. If the current FSM related code generation was performed directly from the trace, or from an extended FSM model including this information, without any tool overly abstracting the required information in between, this issue could be addressed without any manual effort.

With respect to the tools we used, it can be summarized that having more specific and effective means of FSM and invariant generation would be useful. One option would be to extend the models and tools to support the additional information needed. Another option would be to replace those intermediary models by simpler models representing solely the information needed for the generation of the code, and to build our own "FSM" and "invariant" inference engines. However this second option would restrain the possibilities for the user to observe the intermediary models. For example, the usage of ProM permits the user to assess the completeness of the trace, via many different types of models and visualizations. Another enhancement for the tools would be to integrate them more closely with the tracing mechanism so that the models can be built at runtime. This avoids the storage and batch processing issues for the potentially huge traces that are generated for large input sets.

Although, in most cases, the root-causes for errors reported by the MBT tool are clear, sometimes they are difficult to identify. The cause of an error may be located in the model, or in the SUT. An effective approach for finding these causes is to create a separate test case with a specific testing tool, such as JUnit, based on the

generated test case. This separate test case will reveal all the hidden assumptions in data generation, interactions and similar properties, and allow the user to experiment with different settings. A separate test case permits to do more focused analysis of the failure cause. Currently, these tests have to be created manually. However, the information required for their generation is already available in the test case generated by the MBT tool. With this information, the separate test scripts with related data values and other generated input could be automatically generated, saving considerable effort for these difficult debugging cases.

Finally, sometimes, it might be problematic to have an extensive set of test executions available for the SUT. In these cases, an interesting research approach would be to apply input generation mechanisms, such as search-based heuristics [13] for behavior exploration. This would however, require means to sort out the "correct" behavior from the "incorrect" in the generated traces, or at least starting with the assumption that the trace will contain as much erroneous as correct behavior. In this case, the model becomes more of a representation of all the possibilities with the SUT, and less a representation of what is the expected behavior. This has effects on the test oracle generation as described in Section 3. An extensive study is needed to see how this could be made effective.

## 5. Conclusions and Future Work

This paper described a method and a collection of tools that help test engineers derive and refine behavioral models in a semi-automatic way to be used for model-based test generation. We showed how execution traces can first be turned into FSM and invariant models to be used later as basis to generate model code usable for MBT. An automated way to turn these models into an EFSM in a MBT tool notation was presented, along with describing the algorithms needed. The presented technique for generating the test model has some limitations. These limitations were discussed, and we have detailed how they could be addressed in future work. However, even with these improvements it is still important to remember that the generated model alone is not a description of what should be *expected* from the SUT, but rather what it actually *provides*. As the model for MBT must be a description of what is expected of the SUT, it is especially important that the user verifies the model vs. the specification of the system.

Future work will comprise an application of search heuristics, i.e., evolutionary testing techniques, for the generation of test stimuli in order to obtain the traces, and improvement on the generation of transition guards in order to make them more generic.

## References


[1] Groovy - model-based testing with modeljunit. http://groovy.codehaus.org/Model-based+testing+using+ModelJUnit, 2009.

[2] 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.

[3] Antonia Bertolino, Guglielmo De Angelis, Francesca Lonetti, and Antonino Sabetta. Let the puppets move! automated testbed generation for service-oriented mobile applications. In *Proc. of the 34th Euromicro Conf. on Softw. Eng. and Advanced Applications (SEAA2008)*, pages 321–328, Parma, Italy, 2008.

[4] Marat Boshernitsan, R. Doong, and A. Savoia. From daikon to agitator: Lessons and challenges in building a commercial tool for developer testing. In *Proc. of the Int'l. Symposium on Software Testing and Analysis (ISSTA2006)*, pages 169–179, Portland, Maine, USA, 2006.

[5] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006.

[6] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 2009.

[7] Sebastian Elbaum and Madeline Diep. Profiling deployed software: Assessing strategies and testing opportunities. *IEEE Transactions on Softw. Eng.*, 31(4):312–327, April 2005.

[8] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007.

[9] Hans-Gerhard Gross. *Component-Based Software Testing with UML*. Springer, Heidelberg, 2005.

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

[11] Davide Lorenzoli, Leonardo Mariani, and Mauro 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.

[12] Tim Mackinnon, Steve Freeman, and Philip Craig. Endo-testing: Unit testing with mock objects. In *Proc. of eXtreme Programming and Flexible Processes in Software Engineering (XP2000)*, Cagliari, Sardinia, Italy, 2000.

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

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

[15] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, pages 504–527, Glasgow, Scotland, July 2005.

[16] 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.

[17] David Saff, S. Artzi, J.H. Perkins, and M.D. Ernst. Automated test factoring for java. In *Proc. of the 20th Int'l. Conf. on Automated Softw. Eng. (ASE2005)*, pages 114–123, 2005.

[18] N. Tillman and W. Schulte. Mock-object generation with behaviour. In *Proc. of the 21st Int'l. Conf. on Automated Softw. Eng. (ASE2006)*, pages 365–368, Tokyo, Japan, 2006.

[19] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 1 edition, 2006.

[20] 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.

[21] W. M. P. van der Aalst, B. F. van Dongen, C. W. Günther, R. S. Mans, A. K. Alves 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.

[22] Tao Xie and David Notkin. Tool-assisted unit test generation and selection based on operational abstractions. *Automated Software Engineering Journal*, 13(3):345–371, July 2006.

TUD-SERG-2009-017
ISSN 1872-5392