Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

SEN

Software Engineering

Software ENgineering

Adaptive test case execution in practice

J.R. Calamé

REPORT SEN-R0703 JUNE 2007

Centrum voor Wiskunde en Informatica (CWI) is the national research institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organisation for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

Adaptive test case execution in practice

ABSTRACT
Behavior-oriented Adaptation in Testing (BAiT) is a toolset, which supports test generation and execution for deterministic and nondeterministic systems with data. It covers the generation of test cases from a (formal) system specification and test purposes, the identification and selection of test data during test execution and, where necessary and possible, the dynamic adaptation of a test run to the reactions of the implementation under test. The test generation part of the toolset is based on the tool Test Generation with Verification Techniques (TGV) from the Caesar/Aldébaran Development Package (CADP).

1998 ACM Computing Classification System: D.2.5 [Testing and Debugging]
Keywords and Phrases: conformance testing, test case generation, data abstraction, constraint-solving, test execution, Java
Note: Part of this work was funded by the Dutch Bsik/BRICKS project.

Adaptive Test Case Execution in Practice

Jens R. Calamé
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT
Behavior-oriented Adaptation in Testing (BAiT) is a toolset, which supports test generation and execution for deterministic and nondeterministic systems with data. It covers the generation of test cases from a (formal) system specification and test purposes, the identification and selection of test data during test execution and, where necessary and possible, the dynamic adaptation of a test run to the reactions of the implementation under test. The test generation part of the toolset is based on the tool Test Generation with Verification Techniques (TGV) from the Caesar/Aldébaran Development Package (CADP).

1998 ACM Computing Classification System: D.2.5 [Testing and Debugging].
Keywords and Phrases: conformance testing, test case generation, data abstraction, constraint-solving, test execution, Java.
Note: Part of this work was funded by the Dutch Bsik/BRICKS project.

1. Introduction

Motivation Software failures can have severe, sometimes even fatal, consequences. Testing a software product is thus one of the crucial aspects of software development. However, even though mature test approaches have been developed over the years, testing is still very time-consuming and requires lots of resources. It is therefore desirable, to automate the testing process as far as possible.

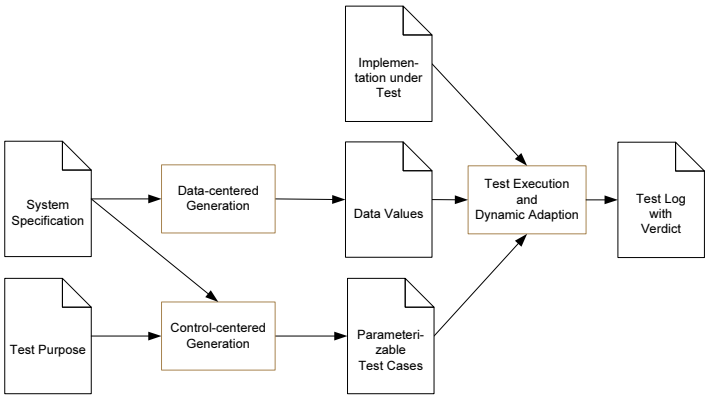

Figure 1: Test Generation and Execution Process

In [3], we developed the theory of an automated test generation and execution process. The process, shown in Figure 1, covers the generation of test cases from a system specification and the sketch of a

test scenario, named a test purpose. The control and data flow of the system are considered separately; test cases are instantiated with actual test data only during test execution. This allows us to reuse once generated test code for different test runs. We differentiate between the tester and the *implementation under test* (IUT).

Behavior-oriented Adaptation in Testing (BAiT) is based on the theory of *conformance testing* [12], one of the most rigorous existing testing techniques. The purpose of conformance testing is to validate, whether an IUT conforms its specification. Generally speaking, this means that, given input allowed by the specification, the IUT may only produce output, which is also allowed by the specification. After a test has been executed, a verdict is assigned, which states whether the system seems to conform to its specification (verdict Pass), whether it does not conform (Fail) or whether the test was not sufficient to come to a final conclusion regarding the conformance of the system with its specification (Inconc).

A specification of a system, however, is a model of the real system and thus in many cases a simplification. This means, that the specification can leave choices for particular implementations open. Such a *nondeterministic* specification can be the basis for a really nondeterministic system, but also for a *deterministic* one, in which the developer has in each instance chosen for one of the possible choices. The test generator selects sequences of actions (*traces*) from the specification, which are then executed. However, if a particular functionality is not implemented as the trace selected by the test generator, but following a valid alternative trace, test execution can result in a wrong verdict.

BAiT minimizes this possibility. The toolset also selects a trace prior to test execution. However, if the IUT deviates from this trace, BAiT evaluates, whether this alternative is valid according to the system specification. If it is valid, test execution adapts to the alternative trace rather than sticking to the preselected one. If the alternative is invalid, a Fail verdict is assigned to the test. The foundation of BAiT has been developed and proven correct in [3]. In this paper, we will present the tool itself.

Outline This document forms several parts. The first part, consisting of this section and the following one, forms a user manual for the test generation and execution framework. While having given an overview in this section, Section 2 contains a tutorial to the general usage of the framework based on a small case study.

The second part, consisting of Sections 3 and 4, forms a reference manual for a deeper insight into the framework. In Section 3, we will regard the case study in more detail. In Section 4, we will give more profound descriptions of the steps a test engineer has to take in order to use this tool.

The third part, finally, consists of Section 5, positioning our framework towards two other widely-used test frameworks (*Testing and Test Control Notation, version 3* (TTCN-3), *Symbolic Test Generation* (STG), Conformiq *Qtronic* and unit test frameworks (xUnit)), and a conclusion in Section 6.

2. Tutorial Case Study

This section serves as a tutorial for the the test generation and execution process of BAiT. As an example, we present a simple automatic teller machine (ATM). We will first model the system and afterwards discuss in detail the inputs, outputs and general steps to be taken for test generation and execution.

Automatic Teller Machine

The system, which we use as our case study, consists of several components (Figure 2). The main component is the ATM itself. It is a reactive system in an environment, which contains a user of the machine, and the user's bank card. While the bank card communicates unidirectional with the ATM, the user interaction is a bidirectional communication. In the diagram, the method calls are given an order (1;2;3;...), also also considering alternative traces (a,b;aa;...).

The specification of the ATM is shown in Figure 3 as a UML state chart. Inputs to the system are attributed with a question mark, outputs from the system with an exclamation mark. When the user inserts his card, the user's personal identification number (PIN) is initialized. The user then

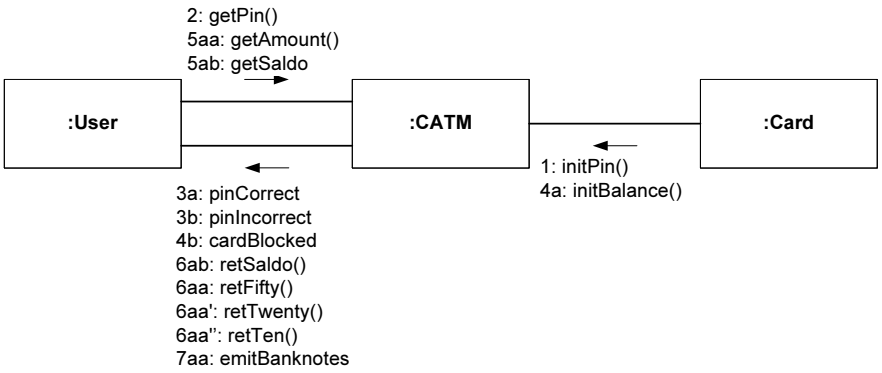

Figure 2: Components of the Automatic Teller Machine

has to enter the correct PIN. If he succeeds, the ATM send the message *pinCorrect* and the process can go on; otherwise the message *pinIncorrect* followed by *tryAgain* or – after the third mismatch – *cardBlocked* is sent.

If the user has entered the correct PIN code, his bank account balance is initialized and the user may choose, whether to review his saldo or to withdraw a particular amount of money. When he chooses the latter option, he either gets the message *retLowSaldo*, if he asked for more money than was actually on the account, or the machine starts paying out. Paying out money is left nondeterministic in this specification: The machine prepares a certain amount of 50 €, 20 € and 10 € bank notes for emission and emits them when they sum up to the requested amount of money. Hereby, it is left open in which order and how many of the single bank notes will be emitted. This decision is made later during the implementation of the ATM and will serve as the example for the adaptation of test execution.

Test Generation and Test Execution
In order to generate and execute a test, several artifacts have to be provided:

System Specification: The system specification defines both data- and control-related aspects, which are implemented in the IUT. It is the starting point for the data-centered test generation and – together with the test purpose – also for the control-centered generation.

Test Purpose: The test purpose is the specification of a test scenario. It guides the control-centered test generation by sketching out relevant actions in a particular order without having to be complete (i.e., naming all actions of a particular trace).

Test Oracle: A test oracle computes possible test input and output data using constraint solving techniques.

Proxy Classes: Proxy classes serve as the platform- and system-specific connector between the generated test cases and the actual IUT.

IUT: The IUT, finally, is the implementation of the software, which is tested.

We will now describe these artifacts in more detail. The details of their processing in the course of the generation and execution process will be described in Section 3. Section 4 will provide an insight into test customization issues.

2.1 The Test Purpose
In our test, we want to validate that after entering the correct PIN code, we will eventually get some money (even more precisely: we do receive some 10 € bank notes). This scenario is formulated as the

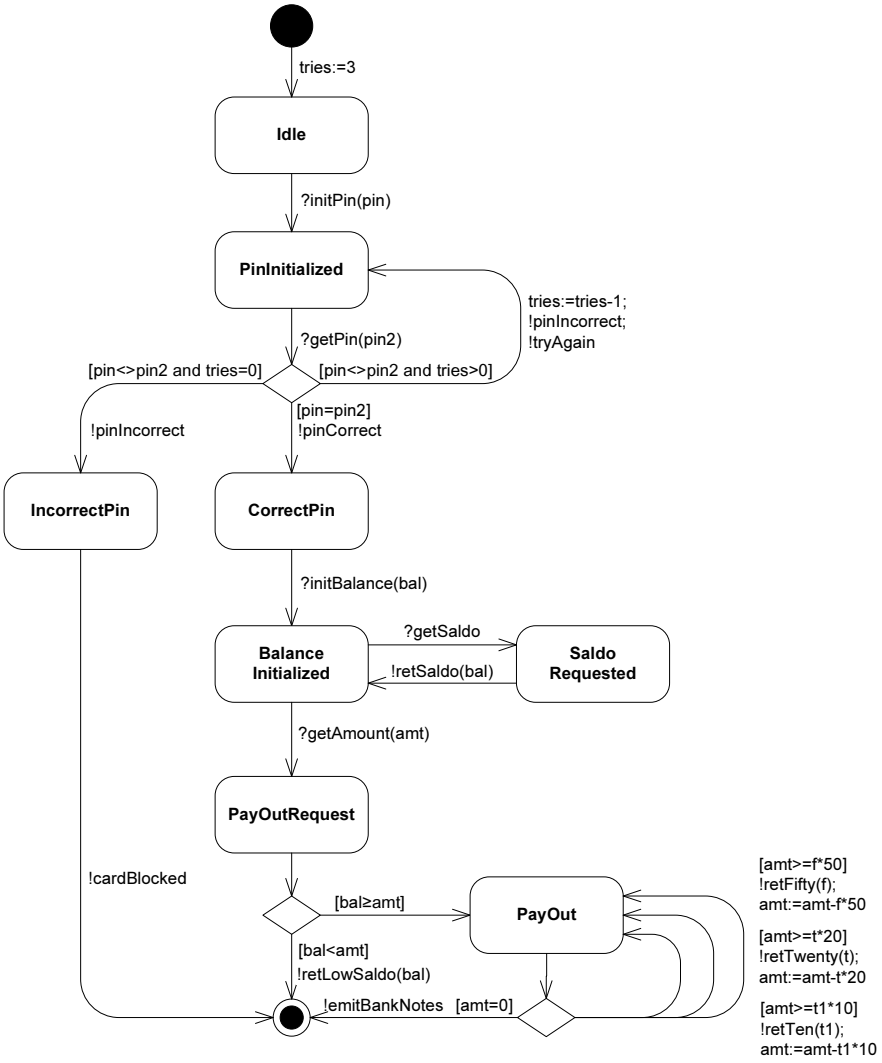

Figure 3: Specification of the Automatic Teller Machine

test purpose in Figure 4. It accepts the occurrence of *emitBankNotes* after entering a correct PIN code (sequence *getPin*, *pinCorrect*) and *retTen*, the preparation of 10 € bank notes for emission. The occurrence of *pinIncorrect* in test traces is refused, i.e. it is not in the focus of this test.

A test purpose does not define a complete trace, as it would be executed during test execution. It rather defines relevant actions in the order, in which they should appear during the test (in our case: *retTen* after *pinCorrect*). Those actions, which are missing in the test purpose, but are relevant during execution, like *getAmount* in our example, are automatically completed during test generation.

Test Generation After having specified the IUT and having set the test purpose, test generation can start. It is a process, which is mainly – in many cases fully – automated, and which we will thus not work out in full detail here (cf. Section 3). Test generation consists of two parallel activities, the generation of the parameterizable test cases (control-oriented generation) and that of the test oracle (data-oriented generation). The output of the generation process is then taken as input for actual test execution (see Figure 1).

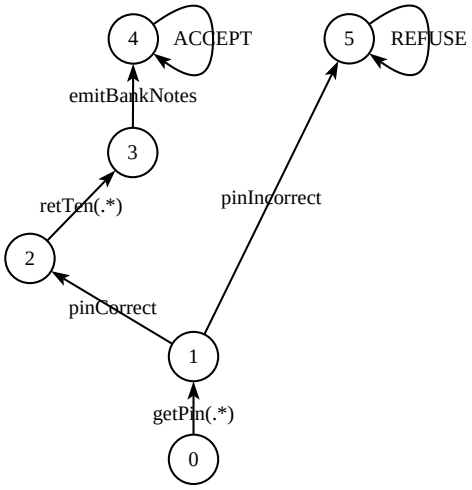

Figure 4: Test Purpose for the Automatic Teller Machine

2.2 Test Oracle
In parallel to generating the test cases, the test oracle can be generated. It is a constraint logic program (CLP), which represents the system specification. Queries to this CLP represent traces through the IUT. A query holds, if its represented trace is possible in the system. The query then results in test input and output data, for which the trace is possible in the system. In BAiT, CLPs are generated for ECL^iPS^e Prolog.

2.3 Proxy Classes
The last step before test execution is the implementation or generation of two proxy classes between the generic part of BAiT and the IUT. At this point, we will concentrate on the implementation of the ATM case study for a moment. Details on the implementation of the proxy classes will be given in Section 4 (test customization).

Implementation of the IUT The ATM is realized as the (procedure-call-based) Java classes `CATM` or `CATM_faulty`, resp. (Figure 5, package `atmv3`), which is based on two interfaces. The interface `IATM`, which is implemented by `CATM` and `CATM_faulty`, declares those actions which serve as *input* to the ATM, while `IATMUser` declares the *output actions*. In order to realize the bidirectional communication between the ATM and its environment on a procedural level, the ATM has been realized following the *Publisher Subscriber Pattern* [7], limiting the number of possible subscribers to emitted events to one. This explains the presence of two additional methods *attach()* and *detach()* in `IATM` to subscribe or unsubscribe a component (i.e. the tester) to or from, resp., the events emitted from the ATM.

The interface `IATMUser` must be implemented by the tester, who also must attach to the IUT. This leads us to the implementation of the test adapter `ATMProxy` from Figure 5. `ATMProxy` implements the interface `IATMUser`, i.e. an implementation is provided for all output actions of the IUT. This implementation does nothing more than logging the action name and all actual parameters received from the IUT.

With `CATM` and `ATMProxy` we have nearly all necessary ingredients to run a test. What we are still missing is the initialization of the IUT and the creation of a proxy object. Therefore, we have to implement or generate the class `TestCATM`, which initializes both the IUT and its proxy. Furthermore, it may define basic settings for the test, like a mapping between datatypes in the specification language and in Java.

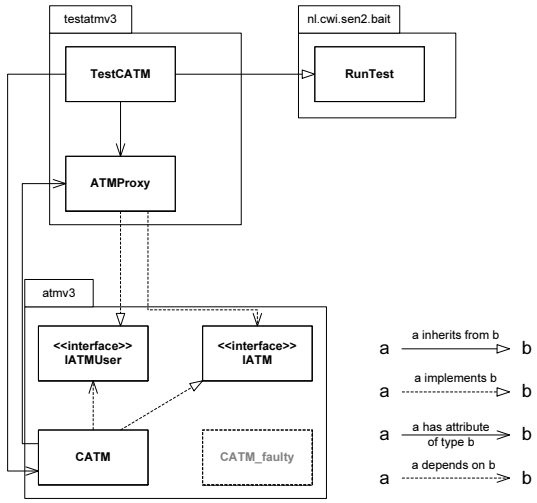

Figure 5: IUT (classes `CATM` and `CATM_faulty`) and tester

2.4 Test Execution

After having compiled the IUT and the two Java classes on the tester side, we can now execute the test cases. For this case study, we implemented two mutants of the ATM. `CATM` works correctly, returning 50 €, 20 € and 10 € bank notes, such that the customer not only receives notes of the largest denomination, but also of the smaller ones. This means, that a request for 100 € will result in one bank note at 50 € each, two at 20 € each and one at 10 € each. The faulty implementation of the ATM, `CATM_faulty`, does not return enough bank notes of the smallest denomination.

When we now execute the test for the correct ATM, we will first be asked to define the values for some variables:

```
Pin in {[-Infinity .. Infinity]} => 5
Bal in {[0 .. Infinity]} => 1000
Amt in {[0 .. 1000]} => 100
```

This happens, since in the default settings (more about that topic in Section 4) test data is instantiated interactively – this could be automated. After entering the values from above, the test will be executed:

```
May 25, 2007 3:09:56 PM nl.cwi.sen2.bait.RunTest executeTest
INFO: Trying: [init],initPin(5),getPin(5),pinCorrect,                \
    initBalance(1000),getAmount(100),[tau],retTen(10),emitBankNotes
May 25, 2007 3:09:56 PM nl.cwi.sen2.bait.RunTest executeTest
INFO: Expected: retTen(10), received: retFifty(1) -> Trace failed,   \
    trying alternative.
May 25, 2007 3:09:56 PM nl.cwi.sen2.bait.RunTest executeTest
INFO: Executed so far: [init],initPin(5),getPin(5),pinCorrect,       \
    initBalance(1000),getAmount(100),[tau],retFifty(1),retTwenty(2), \
    retTen(1),emitBankNotes
May 25, 2007 3:09:56 PM nl.cwi.sen2.bait.RunTest executeTest
INFO: Trying: [init],initPin(5),getPin(5),pinCorrect,                \
    initBalance(1000),getAmount(100),[tau],retFifty(1),retTwenty(2), \
    retTen(1),emitBankNotes
```

```
May 25, 2007 3:09:56 PM nl.cwi.sen2.bait.RunTest executeTest
INFO: Test finished; executed: [init],initPin(5),getPin(5),          \
    pinCorrect,initBalance(1000),getAmount(100),[tau],retFifty(1),   \
    retTwenty(2),retTen(1),emitBankNotes
May 25, 2007 3:09:56 PM nl.cwi.sen2.bait.RunTest executeTest
INFO: The test case ended with verdict PASS.
```

The first thing to remark is, that both the parameters for *initPin* and *getPin* have been instantiated, even though we only provided one value for them. The reason is, that the variable instantiator only tries to instantiate variables, which are not yet defined and whose values cannot be derived from other variables. Since the parameter for *getPin* could be derived from the parameter for *initPin* – they have to be equal for the test to result in a Pass verdict – it is silently instantiated and the test engineer is not asked again.

The second thing to remark is, that the tool first tries to execute the shortest trace to Pass in the test case (first *INFO* line). This fails (second *INFO* line), since the implementation returns one 50 € bank note as explained earlier, instead of ten 10 € notes. Thus test execution must be adapted. In the third *INFO* line, one can see the events sent to and received from the IUT up to now, including the yet unprocessed actions *retTwenty(2)*, *retTen(1)* and *emitBankNotes*. In the next *INFO* line, the alternative trace is shown, which will be executed further. Since it matches exactly with the results received from the IUT, the test ends with a Pass verdict (last line).

Adapting test execution to the output of the IUT does not necessarily lead to a passing test, as we can see below. Adaptation does not lead to a Pass here, since the constraint to receive enough money from the ATM is violated in the test of `CATM_faulty` and thus the test ends with a Fail verdict.

```
May 25, 2007 3:06:19 PM nl.cwi.sen2.bait.steps.TraceImpl merge
FINE: Pruning planned test trace.
May 25, 2007 3:06:19 PM nl.cwi.sen2.bait.RunTest findTrace
FINER: Examining: [init],initPin(Pin),getPin(PinUser),pinCorrect,    \
    initBalance(Bal),getAmount(Amt),[tau],retTen(Twe),emitBankNotes
May 25, 2007 3:06:19 PM nl.cwi.sen2.bait.steps.TraceImpl solve
FINEST: init(G1),initPin(G1,G2,lparam(nat(Pin))),[...]
[Pin in {[-Infinity .. Infinity]} => 5]
May 25, 2007 3:06:21 PM nl.cwi.sen2.bait.steps.TraceImpl solve
FINEST: init(G1),initPin(G1,G2,lparam(nat(5))),[...]
[...]
May 25, 2007 3:06:26 PM nl.cwi.sen2.bait.RunTest executeTest
INFO: Trying: [init],initPin(5),getPin(5),pinCorrect,                \
    initBalance(1000),getAmount(100),[tau],retTen(10),emitBankNotes
May 25, 2007 3:06:26 PM nl.cwi.sen2.bait.RunTest executeTest
FINE: initPin(5) -> OK
[...]
May 25, 2007 3:06:26 PM nl.cwi.sen2.bait.RunTest executeTest
FINE: retTen(10) -> NOT OK
May 25, 2007 3:06:26 PM nl.cwi.sen2.bait.RunTest executeTest
INFO: Expected: retTen(10), received: retFifty(1) -> Trace failed,   \
    trying alternative.
May 25, 2007 3:06:26 PM nl.cwi.sen2.bait.RunTest executeTest
INFO: Trace deviated; executed so far: [init],initPin(5),getPin(5),  \
    pinCorrect,initBalance(1000),getAmount(100),[tau],retFifty(1),   \
    retTwenty(2),retTen(0),emitBankNotes
May 25, 2007 3:06:26 PM nl.cwi.sen2.bait.steps.TraceImpl merge
```

```
FINE: Pruning planned test trace.
May 25, 2007 3:06:26 PM nl.cwi.sen2.bait.steps.TraceImpl merge
FINE: Adding retFifty(1) to trace stub.
[...]
May 25, 2007 3:06:26 PM nl.cwi.sen2.bait.steps.TraceImpl merge
FINE: Adding emitBankNotes to trace stub.
May 25, 2007 3:06:27 PM nl.cwi.sen2.bait.RunTest executeTest
INFO: Test finished; executed: [init],initPin(5),getPin(5),          \
    pinCorrect,initBalance(1000),getAmount(100),[tau],retFifty(1),   \
    retTwenty(2),retTen(0),emitBankNotes
May 25, 2007 3:06:27 PM nl.cwi.sen2.bait.RunTest executeTest
INFO: The test case ended with verdict FAIL.
```

3. A Deeper View at BAiT on the Case Study

Now, we can work out the test generation and execution process in more detail. We will therefore first describe the installation of BAiT and then the single steps taken during test generation and execution. Finally, we will explain some of the internals and options of test execution.

3.1 Installation of BAiT

Behavior-oriented Adaptation in Testing (BAiT) is a toolset for the conformance test of nondeterministic systems or systems based on nondeterminstic specifications, resp. It is itself based on two other toolsets, the micro Common Representation Language (μCRL) toolset and the CADP toolset. In order to install BAiT, the following software must have been downloaded, if necessary compiled, and installed (all version numbers are minimal requirements):

- Java SE JDK version 6 (`java.sun.com/javase/downloads/index.jsp`),
- *ECL^{i}PS^e* Prolog 5.10 (`www.eclipse-clp.org`),
- CADP (`www.inrialpes.fr/vasy/cadp/`),
- μCRL 2.17 (`www.cwi.nl/~mcrl/`).

In order to be able to use the graphical user interface BAiT-UI, you will need to use X-Window with Qt 3.3 and KDE 3.5 libraries installed. The system, BAiT has been tested on, was a standard Fedora Core 6 system.

The installation of the binary distribution of BAiT is relatively straight-forward. The files (`abstractor`, `tdcgenerator`, `testbyabstractionui` and `bait.jar`) must be copied to a directory in the system path. Furthermore, make sure that the file `bait.jar` is in the Java classpath of your system. After having finished the installation, the tools from this section can be used either from the command line or from the graphical user interface BAiT-UI, which is started by invoking `testbyabstractionui`.

3.2 The Test Generation and Execution Process in Detail

In Figure 6, the test generation and execution process is shown again, this time in more detail. The control-centered generation activities are divided into the abstraction of the specification and then the generation of parameterizable test cases. During abstraction, action parameters are replaced by a constant $\mathbb{T}_{\mathbb{D}}$ (named *chaos*) for a variable of domain $\mathbb{D}$. These constants are also present in the generated test cases and are transformed to variable names prior to execution.

The data-centered activity is mainly the generation of test oracle. While this CLP must be present prior to test execution, test data can be selected statically prior to or dynamically during the execution of a test.

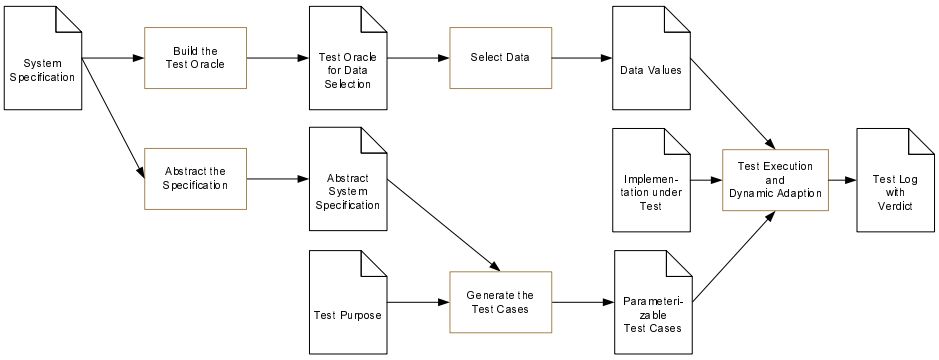

Figure 6: Test Generation and Execution Process, Detailed View

In this section, we describe these steps with the ATM case study as an example. Since there is no direct support for the Unified Modeling Language (UML) yet in the framework, the specification of the IUT must be given in μCRL. For the sake of completeness, this specification is shown in Appendix A.

In order to generate and run the test, several steps have to be taken (see Table 1). Most of them are automated, only a few need manual interaction of a test engineer. In the remainder of this section, we will go through each of the steps and explain it regarding the case study.

1.	Definition of a test purpose.	manually
2.	Data abstraction of the IUT's specification.	automatically
3.	Generation of test cases based on the test purpose.	automatically
4.	Re-introduction of variable names in the generated test case.	automatically or manually
5.	Generation of the test oracle for test data instantiation.	automatically
6.	Implementation of the proxy classes between the tester and the IUT.	automatically or manually
7.	Test execution.	automatically

Table 1: Test Generation and Execution Steps

The realization of the test scenario as well as the implementation of proxy classes has already been described in Section 2; in this section, we will concentrate on the remaining issues. We will use a number of tools from the μCRL toolset [11], the CADP toolset [6] and our own framework. For a complete reference for the named tools, please refer to the appropriate manuals.

The Role of BAiT Test generation and execution is automated by BAiT and is accessible by command line tools and a GUI. After invoking `testbyabstractionui`, this GUI will appear. It has three tabs, of which the first one covers the generation of test cases and test oracles, the second tab covers the generation of the test proxy and the third tab covers test execution. In the remainder of this section, we will explain the steps taken to test the case study on both the GUI level and the command line level.

3.3 Test Case and Oracle Generation

Data Abstraction Test cases are generated on the level of Input/Output Labeled Transition Systems (IOLTSs). IOLTSs are directed graphs, which define a set of states in a system as well as transitions from one state to one or more others. These transitions are labeled with actions, which are executed when the transition is taken during an execution of the system. IOLTSs distinguish actions as inputs, outputs and internal actions in IOLTSs.

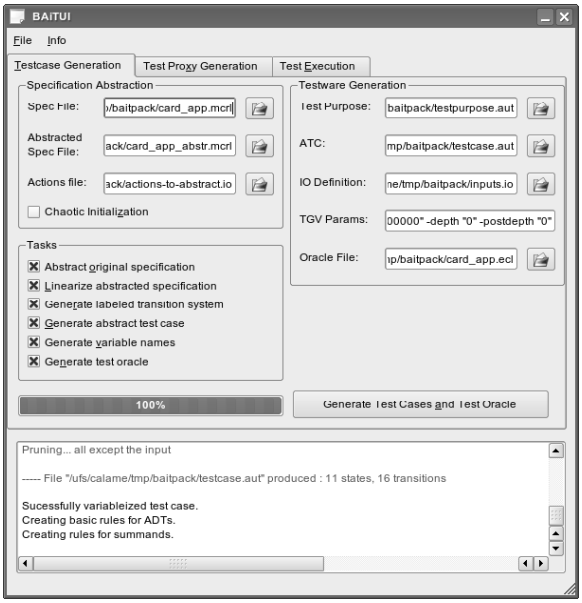

Figure 7: BAiT UI, Test Case and Oracle Generation

The test purpose as well as the system specification are thus given as IOLTSs. Test purposes are always of a quite handy size, while the IOLTS of the IUT's specification normally suffers from state space explosion induced by variable data from large or infinite domains.

Our case study is an example for the use of such data parameters. Having a look at the initialization of the PIN code, for instance, you will notice that the message *initPin* has a parameter from the natural numbers $\mathbb{N}$ (an arbitrary number). If we try to generate an IOLTS directly from this specification, the generator would try to parameterize *initPin* with all possible values from $\mathbb{N}$, what would immediately lead to an infinite number of outgoing transitions already from the system's initial state.

We thus apply *chaotic data abstraction* [3] to the specification. Doing so, all input and output variables are replaced by a constant value $\mathbb{T}$ (*chaos*). De facto, these variables are not replaced by $\mathbb{T}$ directly, but by $\mathbb{T}_{\mathbb{D}}$ for a variable of datatype $\mathbb{D}$ – this detail will become important later. When we now generate the state space of the system under test, we will only get one outgoing transition from the initial state, stating that the PIN is initialized with *any* value.

In order to apply data abstraction, we first have to *linearize* the specification, i.e. bring it into a normal form. Therefore, we use the command `mcrl` from the μCRL toolset. Then we apply data abstraction using the tool `abstractor` from BAiT. We provide a list of actions, which must be abstracted when appearing with an input or output variable (*actions-to-abstract.io*), and the linearized specification to the data abstraction tool, which produces the abstracted specification. This abstracted specification must be linearized again using `mcrl` before finally its state space can be generated with the tool `instantiator` (not related to the variable instantiators described in this document) from the μCRL toolset. In BAiT-UI, the necessary steps are configured in the group *Specification Abstraction* and they are related to the tasks *Abstract original specification*, *Linearize abstracted specification* and *Generate labeled transition system*. The tool invocations on the command line are listed below:

```
mcrl -regular -nocluster card_app.mcrl
abstractor -i actions-to-abstract.io -o card_app_abstr.mcrl card_app.tbf
mcrl -regular -nocluster card_app_abstr.mcrl
instantiator card_app_abstr
```

Generation of Test Cases Having generated the IOLTS of the specification, we can now generate the test cases. Therefore, we use the test generator *Test Generation with Verification Techniques* (TGV) from CADP with the commandline `bcg_open ... tgv` (TGV needs its input in the format *.bcg*, so we first have to convert our state space with `bcg_io`). This tool needs three files as input:

1. the IOLTS of the specification (*card_app_abstr.aut* in our example),
2. the test purpose (*testpurpose.aut*) and
3. a file listing those actions, which are inputs to the IUT (*inputs.io*).

The latter file is necessary to distinguish the set of input actions Act_{in} from that of output and internal actions $Act_{\text{out}} \cup \{\tau\}$ in the specification. Before we can use TGV, we first have to convert the IOLTS into a format, readable for TGV (*.bcg* format). In BAiT-UI, the configuration takes place in the group *Testware Generation* with the task *Generate abstract testcase* activated. On the command line, it looks as follows:

```
bcg_io card_app_abstr.aut tmp.bcg
bcg_open tmp.bcg tgv -io ./inputs.io -hash "100000" -depth "0" \
         -output "testcase.aut" -postdepth "0" ./testpurpose.aut
```

We do not want to go into detail regarding the parameters of TGV here (cf. [2]). TGV can in principle generate three different kinds of test cases: choice-free test cases with or without loops as well as the so-named complete test graphs (CTGs). The first two kinds of test cases are choice-free, that means even if the test purpose allows more than one trace to a Pass verdict, TGV will already select one of them during test generation. That makes them worthless for us, since we have to choose a trace ourselves during test execution. In order to do so, we generate CTGs with TGV.

3.4 Reintroduction of Variable Names

During data abstraction, the names of unbounded variables are replaced by a constant $\mathbb{T}_{\mathbb{D}}$. When now test cases are generated from this modified specification, you will find this constant back as a parameter in all parameterized actions. This has two main disadvantages: (1) you cannot derive the meaning of a variable from its name anymore, and (2) the constraint-solver would bind all variables to the same value during execution. Especially the latter would make a proper test execution impossible, so that unique variable names, or the original ones, resp., must be introduced.

In order to do so, either an automatic tool is used to give all variables a unique identifier, or the variables are renamed manually. The automatic tool *Variableizer* renames all parameters $\mathbb{T}_{\mathbb{D}}$ of an action $s(\mathbb{T}_{\mathbb{D}})$ to $VsnV_{\mathbb{D}}$ where $n \in \mathbb{N}$ is counted up over all parameters appearing in the specification to guarantee uniqueness of the variable names. For instance, a transition $a(\mathbb{T}_{\mathbb{D}_1}, \mathbb{T}_{\mathbb{D}_2})$ in the test case can be transformed to $a(Va1V_{\mathbb{D}_1}, Va2V_{\mathbb{D}_2})$ by the variableizer. When renaming variables, the data domain information must be preserved for test execution.

Variable names are automatically generated from BAiT-UI marking the task *Generate variable names*. The output file is then named `testcase_var.aut` for a test case file `testcase.aut`. From the command line, the tool is invoked as follows:

```
java nl.cwi.sen2.bait.variableizer.Variableizer testcase.aut testcase_var.aut
```

For the moment, we recommend to manually rename variables, since this allows to reintroduce the variables' original semantics. Multiple occurrences of the same variable in a loop during execution are handled automatically; variables then get a suffix $m \in \mathbb{N}$ with m being the number of reoccurrences of the particular variable within the loop. The third occurrence of action $a(Va1V_{\mathbb{D}_1}, Va2V_{\mathbb{D}_2})$ from above, for instance, would dynamically be transformed to $a(Va1V3_{\mathbb{D}_1}, Va2V3_{\mathbb{D}_2})$ to guarantee the uniqueness of the parameter variables over the test oracle.

In this case study, we provide manually named variables for each of the steps in the test case. The result is shown in Figure 8.

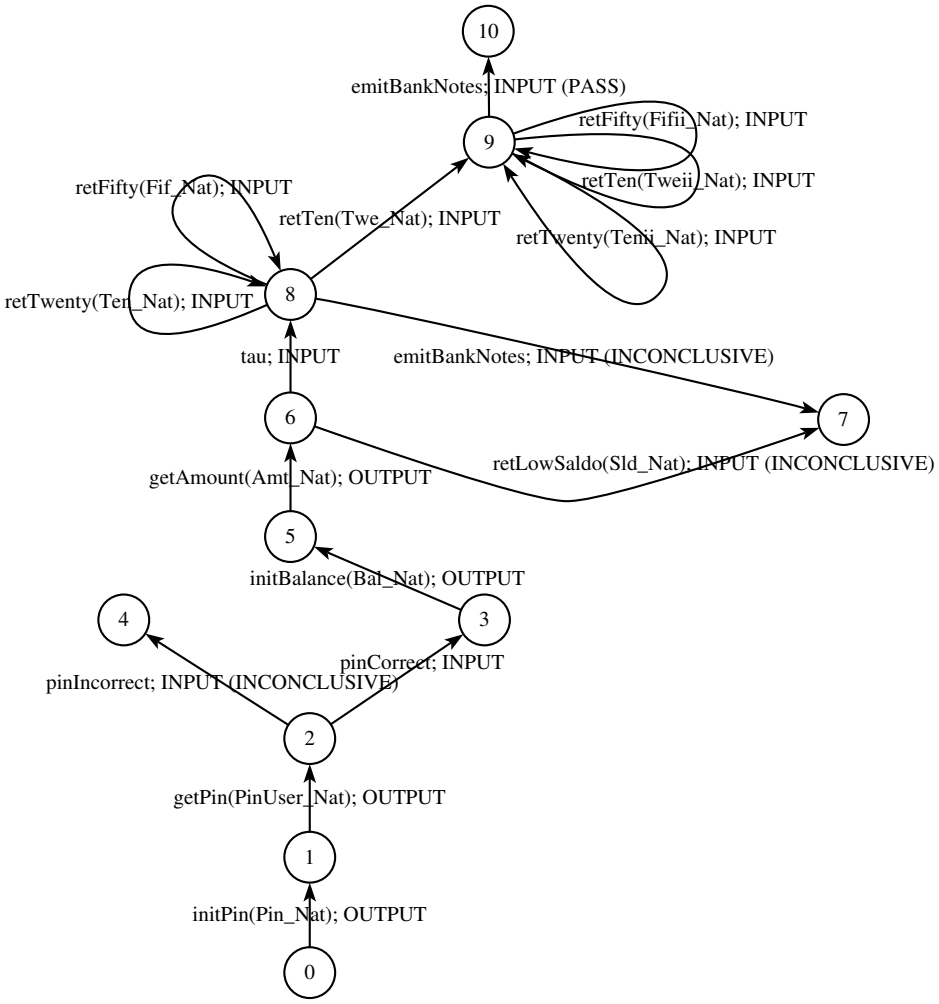

Figure 8: Complete Test Graph with Named Variables

Generation of the Test Oracle Before the test can be executed, the test oracle for test data selection must be generated. In principle, this happens fully automatically by using the *test data constraint generator*. In our example, we generate the constraint program from the linearized specification file `card_app.tbf` and save it in `card_app.ecl`, which is an ECL^iPS^e Prolog file:

```
tdcgenerator -o card_app.ecl -spec card_app.tbf
```

This step is in BAiT-UI also configured in the group *Testware Generation* and is activated by the task *Generate test oracle*. After having done these steps, the test can be executed as described in the previous section.

4. Running a Test

Now we have generated both the test cases and the test oracle at a system independent level. In order to execute the test on a concrete IUT, we have must provide a proxy between the tester and the IUT. In this proxy, we can also customize some implementation aspects of our test.

In this section, we will first introduce the implementation or generation, resp., of the proxy classes and then describe the steps necessary for test execution.

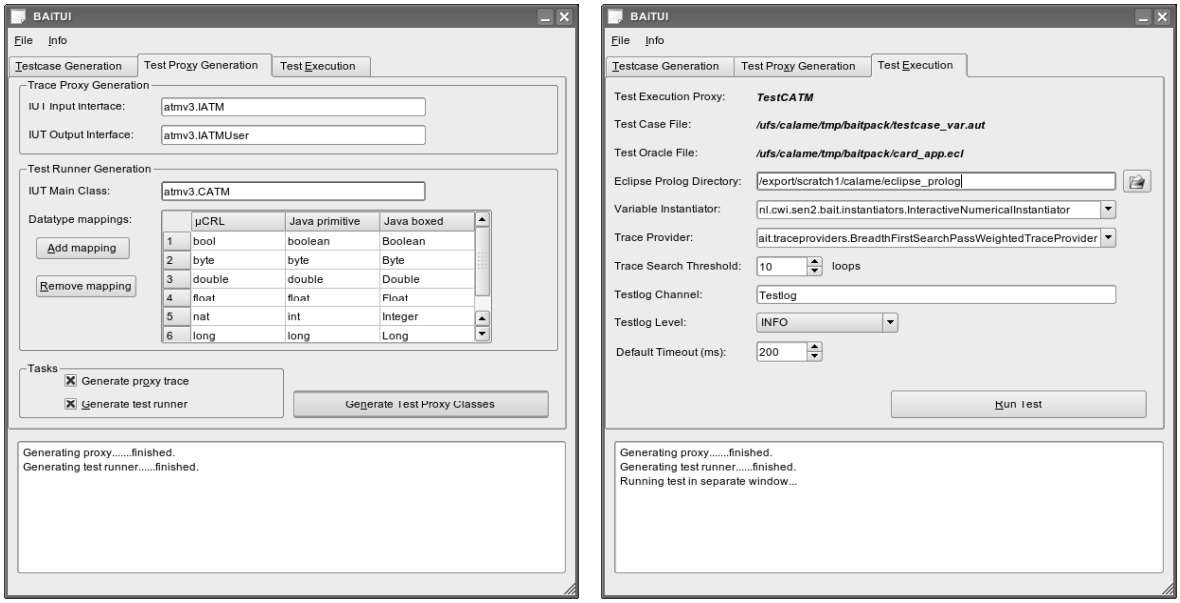

Figure 9: BAiT UI, Test Proxy Generation and Test Case Execution

4.1 Implementation of the Trace Proxy

A test proxy is that object on the tester side, that communicates with the IUT. In our case study, this object is an instance of `ATMProxy` (Listing 1). In a procedure-call-based system, this class is derived from `TraceImpl`, which is provided by the framework; in a system following another communication paradigm, like socket-based communication, it may not be necessary to implement this proxy. The proxy has a constructor, which gets initialized with a timeout and an instance of the IUT. The timeout defines the maximal time, the tester waits for input from the IUT. The instance of the IUT is used to attach the proxy to it (line 7).

The proxy implements the output interface of the IUT, in our case thus `IATMUser`. Each of the IUT's output actions must be implemented by the proxy. This implementation is, however, a rather simple one-liner, which formwards the action name and the set of parameters received from the IUT to the framework for logging (lines 10ff.). The log is kept by the base class of the proxy, `TraceImpl`, which also takes care for the rest of the communication between the tester and the IUT.

Listing 1: Test Runner of the Case Study

```
// [...] package and imports

public class ATMProxy extends TraceImpl implements IATMUser {

    public ATMProxy(int defaultTimeout, atmv3.IATM atm) {
        super(defaultTimeout);
        atm.attach(this);
    }

    public void retFifty(int no) {
        protocolInput("retFifty", new Object[]{no});
    }
    // [...] left out some identically looking methods
}
```

A test proxy can be generated using BAiT-UI by selecting the group *Test Proxy Generation/Trace Proxy Generation* and the task *Generate Proxy Trace*. In order to do the automatic generation, both the input interface of the IUT must be provided (`IATM` in our case) and its output interface (`IATMUser`) must be provided. The tool will then produce a Java class named after the output interface with the postfix `Proxy`; in our case, the output class will thus be named `IATMUserProxy`. On the command line, we can achieve the same using the tool *TraceProxyGenerator*:

```
java nl.cwi.sen2.bait.proxygenerator.TraceProxyGenerator atmv3.IATMUser atmv3.IATM
```

When producing the trace proxy, the two interfaces are loaded into the proxy generator and are analyzed with reflection techniques. For this reason, they must be accessible in the Java class path.

4.2 Implementation of the Test Runner

Automatic Generation BAiT-UI allows to automatically generate a test runner. In the tab *Test Proxy Generation*, the name of the IUT's main class is given together with a mapping of μCRL datatypes to the corresponding types in Java (both primitive types, if applicable, and object-oriented ones). Furthermore, the task *Generate test runner* must be activated. BAiT then automatically generates a template `TestIUT` for the test runner, if the Java class under test is named `IUT`. This template can be further customized, but also compiled and used immediately. To make the generation and customization process more transparent, we will also describe the manual implementation of the test runner.

Manual Implementation The project-specific test runner is a subclass of the framework's generic test runner `nl.cwi.sen2.bait.RunTest` and provides the following:

1. an overridden method *getIUT()*,
2. an overridden method *getAdapter()* and optionally
3. an overridden method *initializeTest()*.

The test runner for our case study is shown in Listing 2. In lines 10ff., you can see the method *getIUT()*, which returns an instance of the IUT. The method *getAdapter()*, which returns an instance of the test adapter, is shown in lines 17ff. The adapter is already initialized with a default timeout setting (see Section 4.3) and bound to the IUT.

In the method *initializeTest()* (lines 23ff.) test execution is configured. It first invokes the standard configuration method from its base class (line 24). This standard configuration initializes the system properties for the test (cf. Section 4.3). Furthermore, it sets up some standard type mappings from the specification language μCRL to Java datatypes as shown in Table 2.

μCRL type[1]	Java unboxed type	Java boxed type
bool ($\mathbb{B}$)	`boolean`	`Boolean`
byte	`byte`	`Byte`
double ($\mathbb{R}$)	`double`	`Double`
float	`float`	`Float`
nat ($\mathbb{N}$ or $\mathbb{Z}$)	`int`	`Integer`
long	`long`	`Long`
short	`short`	`Short`

Table 2: Mappings of Primitive Datatypes

[1] This mapping is the default for the test execution system. However, not all datatypes are necessarily specified in μCRL; this holds in particular for floating-point types.

Listing 2: Test Runner of the Case Study

```
// [...] package and imports

public class RunATMTest extends RunTest {

    private atmv3.IATM _atm = null;

    public RunATMTest() {super();}

    @Override
    protected Object getIUT() {
        if(_atm==null)
            _atm = new atmv3.CATM();
        //  _atm = new atmv3.CATM_faulty();
        return(_atm);
    }

    @Override
    protected Trace getAdapter() {
        return(new ATMProxy(_defaultTimeout,(atmv3.IATM)getIUT()));
    }

    @Override
    protected void initializeTest() {
        super.initializeTest();

        // optional file logging handler
        try {
            addLogHandler(new java.util.logging.FileHandler("testlog.log"));
        }
        catch(java.io.IOException e) {
            System.err.println("Could␣not␣open␣logfile.");
        }
    }
}
```

Afterwards, system-specific initialization can take place. Non-default dataypes can be mapped here by invoking

```
TypeMappingProvider.getInstance().addTypeMapping(<μCRL type>,
  <Java unboxed type>, <Java boxed type>, <type proxy>);
```

setting a μCRL datatype, its corresponding Java types and a type proxy. In case, the newly-mapped datatype maps to a primitive Java type, both the unboxed (primitive) and the boxed (class-wrapped) type have to be defined. If the corresponding Java type is a non-primitive type, the unboxed type is set to `null`. The type proxy, finally, is a class which can decode strings, ECL^iPS^e Prolog terms and Java objects of their corresponding type to set a variable to a particular value and which can also encode this value to both ECL^iPS^e Prolog terms in a string representation as well as Java objects. For any non-primitive datatype, such a type proxy must be implemented; for primitive types, the default implementation `nl.cwi.sen2.bait.variables.ConstrainedVariableImpl` is used. In this case, the third parameter of *addTypeMapping()* can be left out.

Finally, in lines 27ff., we add a file handler for test logs. Adding such a handler is done by invoking the method *addLogHandler()* of the class `RunTest` providing a standard Java logging handle or an own implementation. Adding a logging handler is optional.

The *main()* method, which actually runs the test, is implemented in the generic part of the execution tool and does the following steps:

1. Invoke *initializeTest()* to initialize the test.
2. Setup the ECL^iPS^e Prolog constraint solver and initialize it with the test oracle.
3. Start executing the test, i.e. searching for appropriate test traces and execute them stepwise.

4.3 Running the Test

Test execution is either started from the command line of from within *BAiT UI* (tab *Test Execution*), or from the command line with

```
java <java parameters> nl.cwi.sen2.bait.RunTest <test runner> <test case> \
  <test oracle file>
```

While the three latter parameters have already been explained in the previous section, we will now elaborate on the Java parameters. One of these is mandatory, `-Declipse.directory=<directory of eclipse-prolog>`, which sets the Java property `eclipse.directory` to the base directory of the local ECL^iPS^e Prolog installation. The optional properties are explained below.

Instantiator With the parameter `-Dinstantiator=<variable instantiator class>`, the test execution tool can be configured to use a project-specific variable instantiator. It evaluates, for each unbound parameter in a test trace, the boundaries of this variable and selects an appropriate value for instantiation. The default variable instantiator is `nl.cwi.sen2.bait.instantiators.InteractiveNumericalInstantiator`, which we used already in Section 2. The reference implementation of BAiT also provides a simple random variable instantiator (`RandomInstantiator`) for a fully automated test and an interactive lower/upper boundary variable instantiator (`InteractiveBoundaryInstantiator`). Both classes are situated in the same Java package as the default variable instantiator.

Trace Provider Given a CTG from TGV, single traces to Pass and Inconc states must be selected for execution. This is the task of a trace provider, configured with `-Dtraceprovider=<trace provider class>`. The default is `nl.cwi.sen2.bait.traceproviders.BreadthFirstSearchPassWeightedTraceProvider`. It selects traces from the state space of the CTG using a breadth first search strategy. Doing so, it always looks for the next trace to Pass and returns that one, storing all Inconc traces found in the meanwhile for later use. The reference implementation of the framework also provides the class `BreadthFirstSearchTraceProvider` in the same package, which returns the first trace found to a verdict, no matter whether it leads to Pass or Inconc.

Default Timeout While in a synchronous system, the input from the IUT is received immediately by the tester, this cannot be guaranteed in an asynchronous system. The property `defaulttimeout` defines, how long the tester should wait for the IUT to reply before stopping execution of the actual trace and assuming the IUT being in a quiescent state. This timeout is set by the parameter `-Ddefaulttimeout=<timeout in milliseconds>` and has a default value of 200.

Trace Search Threshold Imagine the test case from Figure 8. Imagine further, that *emitBankNotes* were an internal action of the IUT which does not result in a message sent around. Now imagine, that ordering 100 € from the ATM has yet resulted in one banknote of 50 € and two of 20 €. The test execution would now be in state 9 of the test case still waiting for the remaining 10 €, i.e. for a message *retTen(10)*. The verdict Fail cannot be assigned, since the trace so far has been valid as of the specification – there is just one more banknote missing. Pass, however, can also not be assigned, since we are still waiting for this one banknote.

If it does not receive any message from the IUT for a while, the test execution tool times out and tries another trace to the Pass state. A possible trace would, for instance, be *retTen(0).retTen(1).emitBankNotes*. This means, that an infinite number of traces could be generated, instantiated and tried to be executed, so that test execution could not terminate. One option to still terminate would be to prune transitions from the test case (like the messages *retFifty* or *retTwenty*), which seem not to be necessary. However, this might tamper the testing result, if these "unnecessary" messages are indeed sent before the one message, which we are waiting for. Even though, this is not a very likely scenario, it does not violate the system's specification such that pruning spare transitions is not allowed in this situation.

Instead of pruning transitions, the test execution tool uses a *trace search threshold* to define, how many traces it should generate and try out before giving up with an Inconc verdict. The default value is 10; it can be set by the command line parameter `-Dtracesearchthreshold=<threshold>`. The result of test execution can also be tampered in this case, if test execution gives up too early. However, in this case it can be dynamically configured to try out more traces, so that test execution can be more precise in this respect if this should be necessary.

Logging With the property `loglevel` (`-Dloglevel=<logging level>`), the default loglevel of logging handlers added to the logchannel in the test runner can be changed. This can be used, to make logging more fine-granular. As parameters, the property accepts all Java loglevels, i.e. *SEVERE*, *WARNING*, *INFO*, *CONFIG*, *FINE*, *FINER* and *FINEST*. The *INFO* level allows you to follow executed traces, trace mismatches and test verdicts. This is the level, the standard logging handler of Java (*root handler*) processes. According to this, you will always see the testlog at this level on your console, even without adding any own loghandlers. The level *FINE*, default level if `loglevel` has not been set, allows following test execution on a per-step-level; the level *FINER* also captures output from trace selection.

The testlog is by default sent to a Java logging channel named *Testrun*. If there is any reason to change this name, you can do so by setting the property `logchannel` to another name, using the parameter `-Dlogchannel=<logchannel>`.

5. Discussion

In this section, we want to position our tool towards other, well-known test execution frameworks.

5.1 TTCN-3

Testing and Test Control Notation, version 3 (TTCN-3) [10] is a system-independent test description language. It allows the separation of test code and test data to reuse once written code. TTCN-3 is used for the automatic execution of conformance tests, not for their generation.

TTCN-3 code is compiled prior to test execution. This is a difference to our approach, that interprets test cases during execution. The IUT is bound to the TTCN-3 runtime by platform and system

adapters, which have about the function of the proxy object in our (simpler) framework. Test data is instantiated statically prior to execution, while this happens dynamically during the execution phase in our approach.

Even though, binding a constraint-solver to the TTCN-3 runtime is principally possible, its sense would be rather limited. Since all the trace and data selection would happen in a constraint-solver proxy, TTCN-3 would just get a next step for execution and pass it on to the system adapter, receive an event from the IUT and pass that one on to the constraint-solver proxy. This is exactly, what our test execution tool is doing *without* TTCN-3 in between.

However, there would be possibilities to generate TTCN-3 code that is able to adapt its own execution. Since TTCN-3 is a rich language, all facilities necessary to do so (like loops and conditional branching) are available. There are two possibilities to generate TTCN-3. The first one is a generation of TTCN-3 from the TGV test cases and the test oracle. In the presence of loops in the test cases, this approach would quite likely not terminate. If those loops are unfolded during test execution, the IUT terminates mostly within finite time, so that also test execution terminates (even though this cannot be guaranteed, either). So unfolding these loops seems to be an option, however, if this happens prior to interaction with the IUT, it will limit the ability of test cases to adapt to unexpected system behavior induced by a nondeterministic specification.

Another idea would thus be to generate TTCN-3 code directly from the system specification, which defines the logical structure of the system. The result would be a *mirrored* system, that would serve as the tester of the real system. Since the *whole* system and not only parts of it would be mirrored, a limitation of test cases by test purposes would in this case not be possible anymore.

5.2 xUnit

xUnit (`www.xprogramming.com/testfram.htm`) is a class of very simple unit testing frameworks, which exist for a variety of different system platforms. The "x" in xUnit stands for an arbitrary prefix, depending on the particular framework. A collection of these frameworks has comparatively been described in [1].

xUnit is based on the ideas of agile software development and a test-first approach. Test-first means, that the specification of a system is given as a set of test cases. For this reason, Pass and Inconc verdicts are not distinguished.

There is, in most cases, no separation of test code and test data; however, some frameworks support this, e.g. *JXUnit* for Java as an extension to the standard *JUnit*. System requirements are stated in *assert...()* methods. Such a method fails immediately, if its requirement is not met by the IUT. In this case, the test case fails; an adaptation of the execution is not possible in this way.

This leads to the conclusion, that if we want to profit from the simplicity of xUnit test cases, we cannot provide test case adaptation in contrast to our framework. Since the basic language of a particular xUnit framework is rich enough to provide loops and conditional branching, the same holds as for the TTCN-3 test case generation discussed in the previous subsection. However, using an xUnit framework in this way would not have any advantages towards just using the framework's basic programming language.

5.3 STG

Symbolic Test Generation (STG) is a test generation approach described in [4] and more detailed in [9], implemented in the equally named toolset (`www.irisa.fr/prive/ployette/stg-doc/stg-web.html`), which consists of a test generator and an execution framework.

Like TGV, on whose ideas it is based, the input to STG are a system specification and a test purpose. Since STG works on a symbolic rather than an enumerative level, it is not necessary to generate their state spaces. Both the system specification and the test purpose are thus given as Input/Output Symbolic Transition Systems (IOSTSs). The specification language used for STG is a proprietary one, which is based on the ideas of *IF*. A difference to TGV and to our toolset is, that test purposes may contain guards.

When generating the test case, STG – like TGV – produces the synchronous product of both the system specification and the test purpose from which it then selects test cases. These test cases still contain guards and uninstantiated variables. As its output, STG generates a tester, which is executed in parallel to the IUT. The tester instantiates the variables on the selected test trace is based on output of the *Lucky* solver, taking into account nondeterminism of a system specification. To the best of our knowledge, test trace selection in STG takes place already before test execution, such that the adaption to the system's nondeterminism happens on the level of test data, but not on that of traces. BAiT, on the other hand, supports adaptation with respect to both behavior (i.e., trace) and data.

5.4 Qtronic

TTCN-3 is supported by several commercial tools, which can execute or generate test cases. One of these tools is Qtronic (`www.conformiq.com/qtronic.php`). This tool supports as well the on-the-fly execution of test cases as well as the generation of TTCN-3 code.

The input for Qtronic are UML state charts with their behavior specified in Java (the combination is named QML [5]). Additionally, models can also be specified in Lisp. There exist interfaces for test adapters to C++ and Java.

The tool examines a model and either immediately executes a test on the IUT or generates a TTCN-3 test case. On-the-fly testing can either be random or directled by coverage criteria (condition coverage, branch coverage as well as transition or state coverage on the level of UML). Testing with Qtronic (both on-the-fly and test case generation) does not necessarily terminate, but can be guided by Use Cases. The documentation of Qtronic leaves open, how test cases can be parameterized with custom data other than that needed for the aforementioned coverage criteria or boundary value analysis.

6. Conclusion

With BAiT, we developed a toolset for the generation and execution of conformance test cases of systems with data. It can be seen as an extension of automatic test case generation by TGV [8] or STG [4]. Embedded into the test generation and execution process proposed in this paper, it can provide a nearly fully automatic test of a system.

Automation starts at the specification level. The whole process covers data abstraction, test generation with TGV (requiring the manual preparation of a test purpose) and the generation of a test oracle, which represents the original system in a constraint logic program. After the reintroduction of variable names in the generated test cases and the manual implementation of the test runner and in a procedure-oriented setting of a proxy to the IUT, the test can be run automatically.

While doing so, the TGV test cases are interpreted, handling system control and test data separately. For a full trace, all action parameters are instantiated and the trace is executed step-wise. If the IUT returns an unexpected event, an appropriate test case is searched and test execution is adapted to the new situation. If no trace can successfully be executed, the test ends with the verdict Fail, otherwise it either ends with Pass when a test trace could successfully be executed, or Inconc in case the trace could be executed according to the system specification, even though, the test purpose did not intend this course of test execution.

$$
\begin{array}{lll}
\textbf{proc} & \multicolumn{2}{l}{\text{Idle}(\text{tries} : \mathbb{N})} \\
= & \multicolumn{2}{l}{\sum_{x \in \mathbb{N}} (\text{initPin}(x).\text{PinInitialized}(\text{tries}, x))} \\
\textbf{proc} & \multicolumn{2}{l}{\text{PinInitialized}(\text{tries} : \mathbb{N}, \text{pin} : \mathbb{N})} \\
= & \multicolumn{2}{l}{\sum_{x1 : \mathbb{N}} (\text{getPin}(x1).\text{VerifyPin}(\text{tries}, \text{pin}, x1))} \\
\textbf{proc} & \multicolumn{2}{l}{\text{VerifyPin}(\text{tries} : \mathbb{N}, \text{pin} : \mathbb{N}, x1 : \mathbb{N}) \ \textit{\% implicit in UML state chart}} \\
= & \multicolumn{2}{l}{\text{pinIncorrect.tryAgain.PinInitialized}(\text{tries} - 1, \text{pin}) \lhd \text{and}(\text{not}(\text{eq}(\text{pin}, x1)), \text{gt}(\text{tries}, 0)) \rhd} \\
 & \multicolumn{2}{l}{(\text{pinCorrect.CorrectPin} \lhd \text{eq}(\text{pin}, x1) \rhd \text{pinIncorrect.IncorrectPin})} \\
\textbf{proc} & \multicolumn{2}{l}{\text{IncorrectPin}} \\
= & \multicolumn{2}{l}{\text{cardBlocked.StopMachine}} \\
\textbf{proc} & \multicolumn{2}{l}{\text{CorrectPin}} \\
= & \multicolumn{2}{l}{\sum_{\text{bal} : \mathbb{N}} (\text{initBalance}(\text{bal}).\text{BalanceInitialized}(\text{bal}))} \\
\textbf{proc} & \multicolumn{2}{l}{\text{BalanceInitialized}(\text{bal} : \mathbb{N})} \\
= & \multicolumn{2}{l}{\text{getSaldo.SaldoRequested}(\text{bal})} \\
+ & \multicolumn{2}{l}{\sum_{\text{amt} : \mathbb{N}} (\text{getAmount}(\text{amt}).\text{PayOutRequest}(\text{bal}, \text{amt}))} \\
\textbf{proc} & \multicolumn{2}{l}{\text{SaldoRequested}(\text{bal} : \mathbb{N})} \\
= & \multicolumn{2}{l}{\text{retSaldo}(\text{bal}).\text{BalanceInitialized}(\text{bal})} \\
\textbf{proc} & \multicolumn{2}{l}{\text{PayOutRequest}(\text{bal} : \mathbb{N}, \text{amt} : \mathbb{N})} \\
= & \multicolumn{2}{l}{\text{retLowSaldo}(\text{bal}).\text{StopMachine} \lhd \text{lt}(\text{bal}, \text{amt}) \rhd \delta} \\
+ & \multicolumn{2}{l}{\tau.\text{PayOut}(\text{amt}) \lhd \text{ge}(\text{bal}, \text{amt}) \rhd \delta} \\
\textbf{proc} & \multicolumn{2}{l}{\text{PayOut}(\text{amt} : \mathbb{N})} \\
= & \multicolumn{2}{l}{\sum_{\text{no} : \mathbb{N}} (\text{retFifty}(\text{no}).\text{PayOut}(\text{amt} - (\text{no} \cdot 50)) \lhd \text{and}(\text{ge}(\text{amt}, \text{no} \cdot 50), \text{ge}(\text{no}, 0)) \rhd \delta)} \\
+ & \multicolumn{2}{l}{\sum_{\text{no} : \mathbb{N}} (\text{retTwenty}(\text{no}).\text{PayOut}(\text{amt} - (\text{no} \cdot 20)) \lhd \text{and}(\text{ge}(\text{amt}, \text{no} \cdot 20), \text{ge}(\text{no}, 0)) \rhd \delta)} \\
+ & \multicolumn{2}{l}{\sum_{\text{no} : \mathbb{N}} (\text{retTen}(\text{no}).\text{PayOut}(\text{amt} - (\text{no} \cdot 10)) \lhd \text{and}(\text{ge}(\text{amt}, \text{no} \cdot 10), \text{ge}(\text{no}, 0)) \rhd \delta)} \\
+ & \multicolumn{2}{l}{\text{emitBankNotes.StopMachine} \lhd \text{eq}(\text{amt}, 0) \rhd \delta} \\
\textbf{proc} & \multicolumn{2}{l}{\text{StopMachine} \ \textit{\% Final state in UML state chart}} \\
= & \multicolumn{2}{l}{\tau.\text{StopMachine}} \\
\textbf{init} & \multicolumn{2}{l}{\text{Idle}(3)}
\end{array}
$$

References

1. Jens R. Calamé. Considerations on Object Oriented Software Testing. Preprint 4/2003, Universität Potsdam, Institut für Informatik, May 2003.
2. Jens R. Calamé. Specification-based Test Generation with TGV. Technical Report SEN-R0508, Centrum voor Wiskunde en Informatica, May 2005.
3. Jens R. Calamé, Natalia Ioustinova, and Jaco van de Pol. Towards Automatic Generation of Parameterized Test Cases from Abstractions. Technical Report SEN-E0602, Centrum voor Wiskunde en Informatica, March 2006.
4. Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. STG: A Symbolic Test Generation Tool. In *Proc. of the 8th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, pages 470–475, London, UK, 2002. Springer.
5. *Conformiq Qtronic – Quick Start, Installation, Use, Reference.*
6. Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Laurent Mounier, Radu Mateescu, and Mihaela Sighireanu. CADP – A Protocol Validation and Verification Toolbox. In *Proc. of the 8th Intl. Conf. on Computer-Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 437–440. Springer, 1996.
7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman Publishing Co., Inc., 1995.
8. Claude Jard and Thierry Jéron. TGV: Theory, Principles and Algorithms. *Intl. Journ. on Software Tools for Technology Transfer*, 7(4):297–315, 2005.
9. Thierry Jéron. *Contribution à la génération automatique des tests pour les systèmes réactifs.* Habilitation thesis, L'Université de Rennes 1, Rennes, February 2004.
10. Colin Willcock, Thomas Deiß, Stephan Tobies, Stefan Keil, Frederico Engler, and Stephan Schulz. *An Introduction to TTCN-3.* Wiley, 2005.
11. Arno Wouters. Manual for the μCRL Toolset. Technical Report SEN-R0130, Centrum voor Wiskunde en Informatica, December 2001.
12. ITU-T Recommendation X.290-ISO/IEC 9646-1, Information Technology – Open Systems Interconnection – Conformance Testing Methodology and Framework – Part 1: General Concepts.