

Runtime testing generated systems from Rebel specifications

Thanusijan Tharumarajah

tthanusijan@gmail.com

October 4, 2017, 56 pages

Research Supervisor: Prof.dr J.J. Vinju
Host Supervisor: Jorryt-Jan Dijkstra
Host organisation: ING Bank, The Netherlands, www.ing.nl



UNIVERSITEIT VAN AMSTERDAM

FACULTEIT DER NATUURWETENSCHAPPEN, WISKUNDE EN INFORMATICA
MASTER SOFTWARE ENGINEERING

<http://www.software-engineering-amsterdam.nl>

Contents

Abstract	3
1 Introduction	5
1.1 Problem statement	5
1.1.1 Solution direction	5
1.1.2 Research questions	6
1.1.3 Research method	6
1.2 Contributions	7
1.3 Related Work	7
1.4 Outline	8
2 Background	9
2.1 Rebel	9
2.1.1 Example specification	9
2.1.2 Code generation	10
2.2 Simulation and Checking Specifications	11
2.2.1 Bounded checking	11
2.2.2 Simulation	12
3 Test mechanics	13
3.1 The account specification	13
3.2 Method	14
3.2.1 Evaluation criteria	16
3.3 Approach	16
3.4 Results	17
3.4.1 Codegen-Akka	17
3.5 Analyse	17
3.5.1 Codegen-Akka	17
3.6 Evaluation	18
3.7 Conclusion	18
4 Experiment 1: Invalid execution traces	19
4.1 Method	19
4.1.1 Evaluation criteria	19
4.2 Approach	20
4.2.1 Checking	21
4.2.2 Generated system	21
4.3 Results	22
4.3.1 Codegen-Akka	22
4.4 Analyse	22
4.4.1 Codegen-Akka	22
4.5 Evaluation	25
4.5.1 Bugs	25
4.5.2 Efficiency	25

4.5.3	Coverage	25
4.6	Conclusion	26
4.7	Threats to validity	26
5	Experiment 2: Valid execution traces	27
5.1	More complex specifications	27
5.2	Method	28
5.2.1	Evaluation criteria	28
5.3	Approach	29
5.3.1	Pre-transition check	31
5.3.2	Transition check	32
5.3.3	Post-transition check	33
5.4	Results	34
5.4.1	Codegen-Akka	34
5.4.2	Codegen-Javadatomic	34
5.4.3	Codegen-Scala-ES	35
5.4.4	Distributed Codegen-Akka	35
5.5	Analyse	36
5.5.1	Codegen-Akka	36
5.5.2	Codegen-Javadatomic	38
5.5.3	Codegen-Scala-ES	41
5.5.4	Distributed Codegen-Akka	44
5.6	Evaluation	46
5.6.1	Bugs	46
5.6.2	Efficiency	47
5.6.3	Coverage	47
5.7	Conclusion	48
5.8	Threats to validity	48
6	Discussion	50
6.1	RQ 1: How is the input/output of the generated system tested?	50
6.1.1	Experiment 1: Invalid execution traces	50
6.1.2	Experiment 2: Valid execution traces	51
6.2	RQ 2: Are there any false positives/negatives when the generated system has been implemented correctly?	51
6.2.1	Varying results from the SMT solver	51
6.2.2	Invalid current state	51
6.2.3	Identifiers for entities	51
6.3	RQ 3: What kind of bugs can be found and what are the factors?	52
6.3.1	Templating	52
6.3.2	Compilation	52
6.3.3	Distribution	52
7	Conclusion	53
7.1	Future work	53
7.1.1	mutation model-based testing	53
	Bibliography	55

Abstract

This section summarises the content of the thesis for potential readers who do not have time to read it whole, or for those undecided whether to read it at all. Sum up the following aspects:

- relevance and motivation for the research
- research question(s) and a brief description of the research method
- results, contributions and conclusions

Todo's

add background about smt/sat	9
SMT van nu doet dit niet?	11
paper is niet consistent met nu?	11
add reference for no model when unsat	18
what about expression as iban?	32
check all instances	33
what are the definitions of Rebel types?	51

Chapter 1

Introduction

Growing systems is a concern for large organisations. [1, p. 1] The continuity of systems becomes difficult and a single modification can result in unexpected behaviour of a larger part of the system.

Within the domain knowledge, reasoning about the expected behaviour of a system, changes and errors are hard. *Rebel* aims to solve these challenges by centralising the domain knowledge and relating it to the running systems. *Rebel* is a formal specification language to control the intrinsic complexity of software for financial enterprise systems. [1, p. 1]

Software testing is an important part of software projects. [2, p. 4] The testing process within large systems can be challenging, it entails not only defining and executing many test cases, solving thousands of errors, handling thousands of modules, but also enormous project management. To facilitate this process *Rebel* offers automated simulation and checking of specifications with the use of a Satisfiability Modulo Theories (SMT) solver. This solves to some extent the testing and reasoning of *Rebel* specifications, but this is only within in the *Rebel* domain.

Code generators generate code from the *Rebel* specifications. The problem with code generation is that the resulting product is leaving the *Rebel* domain, causing loss of testing and reasoning with the use of formal methods. The challenge is to regain the benefits from the *Rebel* domain to be able to test and reason about running systems.

1.1 Problem statement

According to the study [3, p. 3], it should be possible to generate running systems from *Rebel* specifications. Right now this is possible, and running systems are generated from *Rebel* specifications. It is not always straightforward to generate a correct system from *Rebel* specifications since *Rebel* is a declarative language. [3, p. 3] As mentioned before, the simulation and checking for the correctness of specifications is only in the *Rebel* domain.

The running systems which are generated from specifications need to be properly based on these specifications, it should be conform to these specifications. So additional work is necessary for the generation process to know that running systems are conform to the specifications.

The language *Rebel* promised to be deterministic, this also holds for the generated system. Thus, non-deterministic behaviour in the generated system should be identified.

Especially for ING Bank, it is important that there is no corrupted data within the runtime systems.

1.1.1 Solution direction

The research is about testing the implementation correctness of specifications. For the problems in [Section 1.1](#), the study [3, p.3] proposed a possible solution for these problems, which is to use SMT solvers. As before mentioned, the mapping of the *Rebel* language to the SMT formulas makes it possible to check and simulate specifications. As a result of this, there is an interpreter for *Rebel* specifications, which is the SMT solver. [1, p.5]

In the same study, an attempt of model-based testing is done to test real banking systems. According to the study, it is only possible to test interactively using the simulation. The steps made in the

simulation are executed in the system under test (SUT), any differences in behaviour are displayed in the simulator. The future work of this approach is to expand the functionality to work automatically with a given trace.

Due to all these reasons, it seems to be a good solution to use the SMT solver which holds the key in testing the generated system. Theoretically, with this approach, it is possible to regain the benefits from the *Rebel* domain, and again able to test and reason about *Rebel* specifications and generated system.

The main research question is as follows: *How to validate the generated code from a Rebel specification?* To research this, the SMT solver is used as an oracle for testing the generated code. So the SMT solver will be used to test the implementation correctness of specifications in the generated system. To clarify implementation correctness, we emphasise templating, compilation and distribution. This applies to the code generators and the generated code. First, the code generators use templating to generate code from the specifications. The generated code should correctly map to the input code from templates. If not, the generated code and *Rebel* specifications will have different meanings. Secondly, the generated code from the code generators needs to compile. Otherwise, it is not possible to run or test the generated system. Lastly, the implementation of the generated system must conform to the *Rebel* semantics, *e.g.*, synchronisation and distribution. For instance, transition atomicity should also be guaranteed in the generated systems. A transition is only allowed to be executed when the preconditions hold. No transitions should change the relevant values before the preconditions and during the execution of the transition. After the execution of the transition, the postconditions of the transition should hold. Concepts such as transactions [4, p. 6] and locking [4, p. 10] influence transition atomicity in the implementation of the specifications (generated system).

With the given approach, a few assumptions need to be made:

- **The specification is always correct.** The specifications are written correctly, *i.e.*, the specifications are correctly modelled from the business point of view. An incorrect specification will probably not pass the checking or simulation, and testing a generated system derived from an incorrect specification is not really effective.
- **The generated system can be compiled.** When the generated system cannot be compiled, it is possible that there might be a fault in the code generator. Although, with this approach testing a not compilable system is impossible.
- **The *Rebel* specifications are correctly interpreted by the SMT solver.** The SMT solver is used as an oracle/black box in the testing approach since it is an interpreter for *Rebel* specification. However, when something goes wrong with the mapping of the *Rebel* language to the SMT formulas, this will result into misbehaviour of the specification which may lead to incorrect results.

1.1.2 Research questions

The following questions are defined to achieve the research goal:

RQ How to validate the generated code from a Rebel specification?

SQ1 How is the input/output of the generated system tested?

SQ2 Are there any false positives/negatives when the generated system has been implemented correctly?

SQ3 What kind of bugs can be found and what are the factors?

1.1.3 Research method

We test generated systems by the code generators in two ways, invalid execution and valid execution. The first experiment tests invalid execution in the generated systems. Therefore, the test framework will use checking to check the satisfiability of a given specification. However, testing valid execution

can also provide valuable results. The second experiment tests valid execution in the generated systems with the use of checking and simulation.

At first, an initial lightweight version is expected, then it will be extended with motivated improvements with evaluation and validation. The proof of concept is a testing tool for testing the implementation correctness of a specification of SUT.

The approach is to start with the lightweight version which can trigger a bug and test it with the SMT solver. For the lightweight version, it is an easily reproducible bug. Then the lightweight version is improved with smarter testing techniques to generate tests automatically, and these improvements are done with evaluation and validation. For example, by using existing software testing techniques like Concolic testing [5], Fuzz testing [6] and Mutation testing [7].

1.2 Contributions

The research has the following contributions:

1. Methodologies to validate generated systems from *Rebel* specifications. These methodologies include an in-depth analysis and evaluation of the results.
2. Limitations in *Rebel* and SMT encoding as this an important part of the test approach. These limitations can lead to false positives when the generated system is generated correctly.
3. The bugs and factors encountered in the generated system that was found using the methodologies.

1.3 Related Work

Model-based testing

Model-based testing entails the process and techniques for automatic generation of test cases using abstract models. [8, 9, 10] Test cases are generated based on these models and then executed on the SUT. These models represent the behaviours of a SUT and/or its environment. [8, 9]

After defining the model, test selection criteria need to be defined to transform these criteria into test case specifications. Test case specifications describe on a high level the desired test case. Test cases are generated when the model and test case specifications are defined. [8] Then a test execution environment can be used to automatically execute test cases and record verdicts.

The main difference with our approach and model-based testing is that the model is already present. The model in *Rebel* is the *Rebel* specifications. *Rebel* specifications describe banking products, and also running systems are generated from it. The model in model-based testing is built from informal requirements or existing specification documents. [8, p. 2] This model shares the same characteristics as *Rebel* specifications.

In model-based testing, there exist several test generation technologies to generate test cases, such as random generation, (bounded) model checking, etc. [8, p. 8-9] As mentioned earlier, *Rebel* offers automated simulation and checking of specifications with the use of an SMT solver. For both simulation and checking, *Rebel* uses bounded model checking. Our approach is also using the bounded model checking to test the SUT.

Runtime verification

Runtime verification is a technique to ensure that a system at the time of execution meets the desired behaviour. [11, 12, 13] Runtime verification is seen as a lightweight verification in addition to verification techniques like model checking and testing. [11, p. 294] This gives the possibility to react when misbehaviour of a system is detected. The origins of runtime verification are in model checking, but a variant of linear temporal logic is often used. The main difference between runtime verification and other verification techniques is that runtime verification is performed at runtime. The focus of runtime verification is to detect satisfactions or violations of safety properties. [11, 13]

The monitor performs the checking whether an execution in the system meets the safety property. [11, p. 295] The device which reads a finite trace and gives a certain verdict is called monitor. The monitors are usually automatically generated from a high specification in runtime verification. [11, 13]

The main similarity of runtime verification with our approach is the ability to test systems at runtime. In our approach, the generated systems are being tested against the Application programming interface (API), which is at runtime. Runtime verification is only considered to detect satisfactions or violations of safety properties. [11, 13] In our approach, simulation and checking, which uses bounded model checking, will be used to test the generated systems. Bounded model checking is also used to check whether a safety property holds. [1, p. 4] Although, the approach we have chosen for is not only to check the safety property but also to check whether a certain execution is (not) possible in the generated system.

The main difference between runtime verification and model checking is the presence of a model of the system to be checked. Runtime verification refers only to executions observed as they are generated by the real system; thus there is no system model. [11, p. 295] However, with model checking, a model of the system to be checked needs to be build to check all possible executions.

As said before, in runtime verification are the monitors usually automatically generated from a high specification in runtime verification. In comparison to our approach, we are not going to generate monitors since we are going to test the generated systems with simulation and checking, whether the generated systems behaves conform to the specification.

Specification based testing/random testing

Testing distributed systems

1.4 Outline

This section outlines the structure of the thesis. [Chapter 2](#) contains the background of this thesis. As this research focuses on experiments that validate generated systems, each experiment is divided into its own chapter. The experiments test the generated systems in two ways, invalid execution and valid execution. The lightweight version which tests invalid execution is discussed in [Chapter 3](#). The invalid execution experiment and its results are discussed in [Chapter 4](#). The valid execution and its results are discussed in [Chapter 5](#). [Chapter 6](#) contains the answers for each research question, also containing a discussion of the conducted experiments, limitations and found bugs. Finally, a conclusion of this thesis is given in [Chapter 7](#).

Chapter 2

Background

add background
about smt/sat

2.1 Rebel

Rebel is a formal specification language written in the language workbench Rascal [14]. The specification language is developed by ING¹ and Centrum Wiskunde & Informatica (CWI)².

The language is used for controlling the intrinsic complexity of software for financial enterprise systems. [1, p. 1] The goal of *Rebel* is to develop applications based on verified specifications that are easy to write and understand. The formal specification language makes product descriptions more precise, and it removes the ambiguity. The simulation in the language is used as an early prototyping mechanism to verify the product with the user. *Rebel* can specify banking products like savings accounts.

The mapping of the *Rebel* language to the SMT formulas makes it possible to simulate and check these specifications. Simulation and checking specifications can be used for early fault detection.

2.1.1 Example specification

An example of a *Rebel* specification is given in Listing 2.1. The specification specifies a simple account where it is only possible to open an account with some balance. After opening an account, the state of the account goes to the opened state which is also the final state. When the account is in its final state, no further action is allowed. Notice also the fields of the specification; these are the account number of type *IBAN* and balance of type *Money*.

¹<https://www.ing.nl/>

²<https://www.cwi.nl/>

```
1 specification Account {
2   fields {
3     accountNumber: IBAN @key
4     balance: Money
5   }
6
7   events {
8     openAccount[]
9   }
10
11   lifeCycle {
12     initial init -> opened: openAccount
13     final opened
14   }
15 }
```

Listing 2.1: A simple account specification

As shown in the specification, it describes only what is possible with an account and not how. The specification does not contain the definition of the events. These definitions are specified somewhere else to promote reuse of events and invariants for other *Rebel* entities, and to make *Rebel* specifications more concise. [1, p. 4]

The definition of the transition *openAccount* is illustrated in Listing 2.2. The precondition of the transition is that the initial deposit should be equal or above 0 euro. To assign the value to the field *balance*, the keyword *new* is used in the postcondition. This refers to the value of the variable in the post-state after the execution of the transition. [1, p. 4]

```
1 event openAccount[minimalDeposit: Money = EUR 0.00](initialDeposit: Money) {
2   preconditions {
3     initialDeposit >= minimalDeposit;
4   }
5   postconditions {
6     new this.balance == initialDeposit;
7   }
8 }
```

Listing 2.2: *openAccount* event definition from specification

2.1.2 Code generation

Writing programs that write programs is called code generation. [15, p. 3] Code generation in software engineering projects can result in a valuable impact on productivity and quality. The volume of code generated by the code generator would take much longer when it is written manually. Generating code from templates preserves consistent code quality throughout the entire code base. Even when a bug is encountered or improvements are made in generated code, in short time can these errors be fixed through revising the templates and starting the code generation process. [15, p. 15-17] These fixes are applied consistently throughout the code base.

The code generators of ING Bank are capable of generating source code from a *Rebel* specification. These generators are a template-based generator which uses a Rascal (which has a page-template feature)[14] to build code. The following generators exist right now for *Rebel*:

- Codegen-Akka: The Codegen-Akka generator generates a Scala system from *Rebel* specifications. The generated system uses Akka [16, p. 4] as Actor Model and Cassandra [17] is used for storage.

- Codegen-Javatomic: This generator generates a Java system based on the *Rebel* specifications. The generated system uses Datomic [18, p. 170-172] for storage.
- Codegen-Scala-ES: The Codegen-Scala-ES generator also generates a Scala system. The implementation of the generated system uses Command Query Responsibility Segregation (CQRS) [19] and Event Sourcing [20].

The API's of the generated system from the code generators are not standardised. The request which is made for transitions are all implemented in the same way between the code generators. However, the response returned by the generated system may differ. For example a request for the transition given in Listing 2.2 looks as follows: `{ "OpenAccount": { "initialDeposit": "EUR 50.00" } }`. Since the interactions for transitions within the generated systems are the same, all three code generators can be used to test the implementation of *Rebel* specifications.

2.2 Simulation and Checking Specifications

The semantics of *Rebel* is defined as labelled transition systems. [1, p. 5] Thus the current state of a specification holds the state name with the current fields assignments and the event parameters which causes the current state. The labelled transitions map to the events and their preconditions and postconditions. *Rebel* has also support to specify invariants for a given specification. These are predicates which should always be true. These predicates are defined as external specifications of expected behaviour, that is why they are converted to additional formulas.

SMT van nu doet dit niet?

Bounded model checking can be used for *Rebel* specifications. Therefore, *Rebel* is defined as an SMT problem by encoding it to symbolic bounded model checking (with data). The goal of model checking is to find a state which is reachable with some properties which don't hold. [1, p. 5] For example, for the specification from Listing 2.1, an account within the state opened with a negative balance. *Rebel* uses SMT solver Z3 [21] for simulation and checking.

2.2.1 Bounded checking

Checking of *Rebel* specifications is used to check the consistency of a given specification. [1, p. 5] A specification is consistent when invariants hold in all reachable states. A state is reachable when it can be reached from the initial state via valid transitions.

paper is niet consistent met nu?

The bounded analysis tries to find the smallest (the least possible steps) possible counterexample; this is fully automatic and incremental. Thus the given computations by the SMT solver satisfies the route from pre-condition to post-condition for every transition. First, it tries to reach an invalid state in one step. If that did not succeed, then it tries to reach the invalid state in two steps. This process continues until a counterexample is found or configurable time-out is met. The bounded analysis tries to prove the following: $\theta(s_0) \wedge P(s_0) \wedge p(s_0, s_1) \wedge P(s_1) \wedge \dots \wedge p(s_{k-1}, s_k) \wedge \neg P(s_k)$.

An example of checking *Rebel* specifications is given in Listing 2.3. These checks can be defined in so-called tebl files. The SMT solver tries to reach the state opened of account with the balance above 0 euro. As configurable time-out is six used.

```
1 module simple_transaction.OpenAccountCheck
2
3 import simple_transaction.Account
4
5 state openAccountCheck {
6   opened Account with balance > EUR 0.00;
7 }
8
9 check openAccountCheck reachable in max 6 steps;
```

Listing 2.3: Checking opened account

2.2.2 Simulation

The purpose of simulation and checking differs. The purpose of the simulation is to check the external consistency, and checking is used for the internal consistency. [1, p. 5] As explained in the previous paragraph, checking is used to reason about all possible traces. Simulation focuses on individual steps to reason about. Thus with the simulator, the user can quickly check the specification behaves as expected. As for checking, the same strategy is used in the simulation, *i.e.*, using SMT solver and encoding for *Rebel* Specifications.

Chapter 3

Test mechanics

In this chapter, we explain how to implement the lightweight proof of concept. The intention of the lightweight version is to know that the test approach can find a bug in generated systems. Therefore, we need to understand how the existing foundations from *Rebel* can be reused.

3.1 The account specification

An extended account specification ¹ from [Listing 2.1](#) is used for this experiment, the implementation in *Rebel* is shown in [Listing 3.4](#). According to the specification, an account can be opened with a minimum deposit of 50 euro. When an account is opened, it is possible to withdraw or deposit money. Besides deposit and withdraw, the balance may increase or decrease by interest. To disable any account, the *block* event can be used to put the account to the state blocked. The final state of account is closed. Therefore there should not be any remaining balance. When an account is in the state closed, no further action is allowed since it is in the final state. The invariant is specified to keep every account with a positive balance.

¹https://github.com/cwi-swat/rebel/blob/e58590c7f51f59e7ee6443bb89ef09dff6febab6/rebel-core/examples/simple_transaction/Account.ebl

```
1 specification Account {
2   fields {
3     accountNumber: IBAN @key
4     balance: Money
5   }
6
7   events {
8     openAccount[minimalDeposit = EUR 50.00]
9     withdraw[]
10    deposit []
11    interest []
12    block []
13    unblock []
14    close []
15  }
16
17  invariants {
18    positiveBalance
19  }
20
21  lifeCycle {
22    initial init -> opened: openAccount
23
24    opened -> opened: withdraw, deposit, interest
25            -> blocked: block
26            -> closed: close
27
28    blocked -> opened: unblock
29
30    final closed
31  }
32 }
```

Listing 3.4: Account specification

The account specification has a *close* event² to close an account which is illustrated in Listing 3.5. The precondition of the *close* event is that the balance of the account should be equal to zero. There are no postconditions, this means that the postconditions are satisfied. So there are no properties changed of the account, but only the state is changed to closed.

```
1 event close() {
2   preconditions {
3     this.balance == EUR 0.00;
4   }
5 }
```

Listing 3.5: *close* event definition from account specification

3.2 Method

As mentioned earlier, an initial lightweight version is expected, then it will be extended with motivated improvements with evaluation and validation. The approach is to start with a lightweight version

²https://github.com/cwi-swat/rebel/blob/e58590c7f51f59e7ee6443bb89ef09dff6febab6/rebel-core/examples/simple_transaction/Library.ebl

which can trigger a bug and test it with the SMT solver. For the lightweight version, it is an easily reproducible bug. This lightweight version is then able to trigger and test one specific bug.

Listing 3.6 illustrates the code which is generated to check the precondition for the *close* event. From this code, we can see that the balance of an account should be zero before it is getting closed. So we can assume that the precondition is correctly generated.

```

1 case Close() => {
2
3   checkPreCondition(({
4     require(data.nonEmpty, s"data should be set, was: $data")
5     require(data.get.balance.nonEmpty, s"data.get.balance should be set, was: $data.get.balance")
6     data.get.balance.get
7   } == EUR(0.00)), "this.balance == EUR 0.00")
8
9 }

```

Listing 3.6: Generated Precondition for *close* event

The first bug to trigger is to close an account with some balance. To do this, the precondition of the *close* event should be changed in the generated system (see [Figure 3.1](#)). By manually making the changes in the generated system, the SUT, we know that there is definitely a bug in the generated system, assuming that the specification is correct.

The modified precondition looks as follows in [Listing 3.7](#). The precondition is changed to *RebelConditionCheck.success*, this means that the precondition is satisfied. Right now we have introduced a bug in the SUT. Thus the SUT is not conform to the specification.

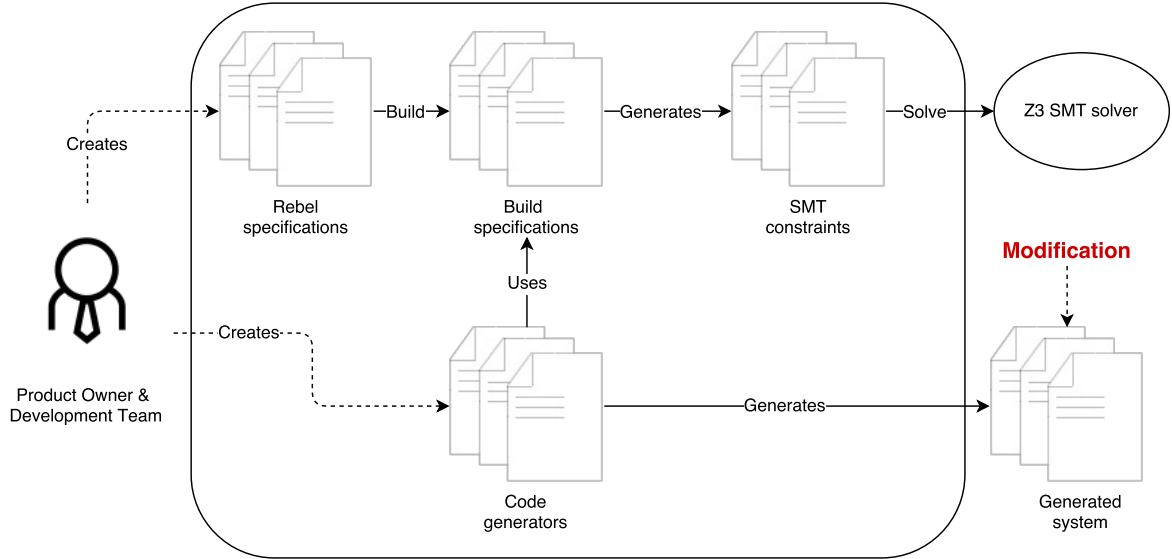


Figure 3.1: Modification in specification development

```

1 case Close() => {
2   RebelConditionCheck.success
3 }

```

Listing 3.7: Modified Precondition for *close* event

3.2.1 Evaluation criteria

Bugs

Since the precondition of the *close* transition is modified in the SUT, we know that the SUT contains the bug to close an account with some balance. For the lightweight version, it is expected to find this bug in the SUT.

Efficiency

The lightweight version will use checking to check whether it is possible to have a closed account with some balance. Therefore, it is expected to test the same transition in the SUT.

Coverage

With this lightweight version, we are going to trigger one single bug in the SUT, *i.e.* finding the bug for the *close* transition. Thus from the account specification, we are testing only the transition *close*.

3.3 Approach

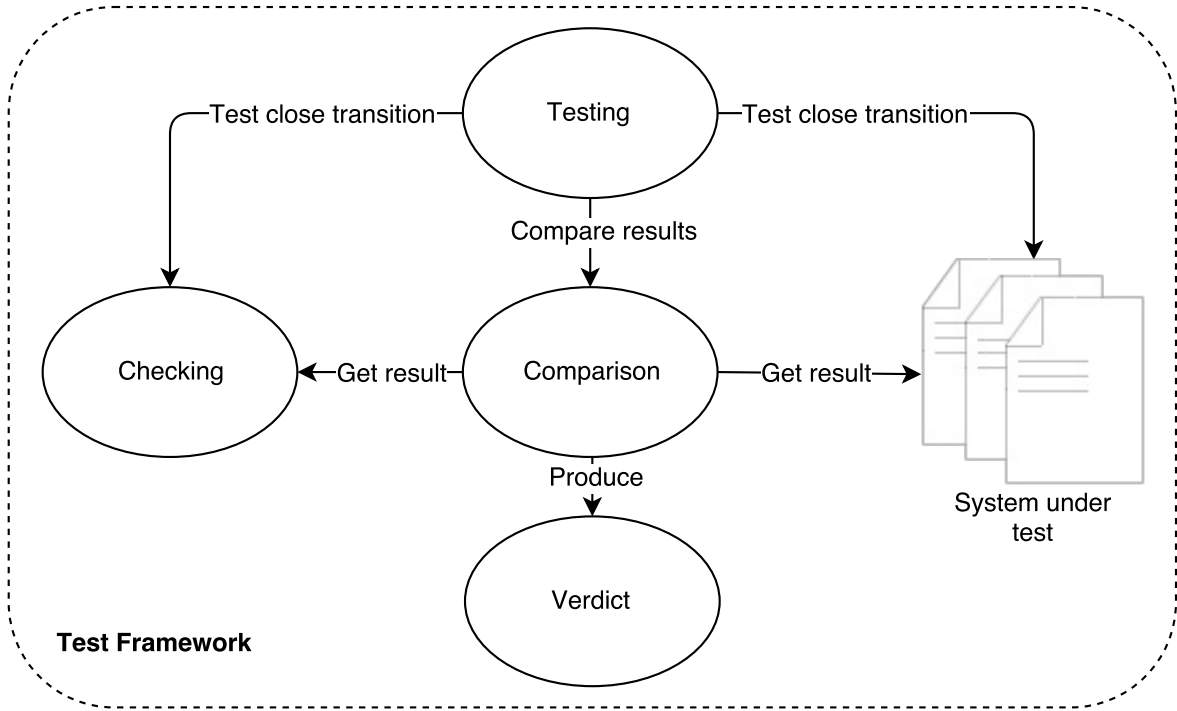


Figure 3.2: Testing approach for *close* transition

The testing approach is shown in [Figure 3.2](#). We discussed earlier that we are going to use the SMT solver to find bugs in the SUT. Having an account with some balance in the state closed should not be possible according to the specification. To let the SMT solver solve this situation, it is necessary to generate the appropriate SMT formulas. Therefore, checking can be used to check whether the given state with its properties is reachable. The test framework first uses checking to test whether the state is reachable.

To check the state is a tebl file created which is shown in [Listing 3.8](#). It defines the state of a closed account with the property balance, where the balance is not equal to zero. Also, here is six used as configurable time-out. The SMT solver tries to solve this problem in max six steps.

```

1 module simple_transaction.ClosedAccountWithBalance
2
3 import simple_transaction.Account
4
5 state closedAccountWithBalance {
6   closed Account with balance != EUR 0.00;
7 }
8
9 check closedAccountWithBalance reachable in max 6 steps;

```

Listing 3.8: Closed account test

The input for the SMT solver is now defined. Similar behaviour should be defined for the SUT. So an account needs to be opened and closed afterwards. Therefore, the testing framework performs both transitions in the SUT.

The tebl file is passed to the model checker, and it returns whether the given SMT problem is reachable or not. A state is reachable when it can be reached from the initial state via valid transitions. [1, p. 4] To check if the state is reachable in the SUT, the request made for the given transition contains afterwards a check whether the request is successful. Then the testing framework can compare the results from the SMT solver and the request made in the SUT.

3.4 Results

3.4.1 Codegen-Akka

Since we have defined the input for both systems and can compare it, we can trigger the bug and compare the results of it. The results of testing the *close* transition are shown in Table 3.1.

Transition to test	Reachability SMT solver	Reachability SUT	Test result
close	✗	✓	✗

Table 3.1: Results: testing *close* transition of account specification

3.5 Analyse

3.5.1 Codegen-Akka

According to the results from Table 3.1, the generated test for the *close* transition has failed. The results of the model checker state that the defined state in Listing 3.8 is not reachable. Although, the state in the SUT is reachable.

Looking at the account in the SUT, it looks as follows Listing 3.9. The state of the account is closed, and the balance is the same as when it was opened. To conclude, the *close* transition is performed in the SUT due to the modification of the generated precondition.

```

1 {
2   "state":{
3     "SpecificationState":{
4       "state":{
5         "Closed":{
6
7         }
8       }
9     }
10  },
11  "data":{
12    "Initialised ":{
13      "data":{
14        "accountNumber":null,
15        "balance":"EUR 50.00"
16      }
17    }
18  }
19 }

```

Listing 3.9: Account state after *close* transition

3.6 Evaluation

Bugs

The expectation for the bugs criteria is to find the bug in the *close* transition. As expected, we did find the bug for the *close* transition due to the manually modified precondition.

Efficiency

Checking is used in this lightweight version to check whether it is possible to have an opened account with some balance. According to Table 3.1, this state is not reachable. It is expected to test the same transition with checking in the SUT.

When the model checker returns that a given state is not reachable, traces are not available. Models (traces) are not available when "unsat" is returned by Z3. In this case, traces cannot be used since they are not provided. Although, to reach the given state, the *openAccount* and *close* transition are used.

add reference
for no model
when unsat

Coverage

The lightweight version is used only to trigger a single bug in the SUT. As expected, only the transition *close* tested. Although, to close an account the account needs to be first opened. Therefore, the transition *openAccount* performed, but this transition is not tested whether the request is successful.

3.7 Conclusion

As we have seen with this lightweight version, it can find one specific bug with the use of an SMT solver. Since the code generator is template based, it is possible to find faults in templating. There are two parts where there can occur faults during the generation parts. As described in [22, p. 274], the majority of the generated code is fixed, some isolated parts are dependent on the input of the model. So it is possible that there might be bugs in the fixed code. The second part is the template control code which is injected into the generated code. The manually introduced bug from the previous chapter belongs to this category.

Chapter 4

Experiment 1: Invalid execution traces

Discovering the unexpected is more important than confirming the known.

George E. P. Box

The lightweight proof of concept discussed in [Chapter 3](#) is only able to trigger one bug which is created manually. In this chapter, we discuss how the lightweight proof of concept is automated and the test results of the generated system.

4.1 Method

The lightweight version from the previous chapter is only able to test one specific bug. The bug itself is created manually by modifying the SUT. Now, this lightweight version needs to be automated to automatically generate a test for every transition from a specification.

With every transition, it is possible to reach a state or stay in the current state. To check *Rebel* specifications, the state to reach with a transition needs to be defined. As mentioned before, the goal of model checking is to find a state which is reachable with some properties which do not hold [1, p. 5]. Thus defining only the reachable state is not enough, the properties of interest for a transition needs to be specified. Each property is different per transition, so these properties should be different for the defined state. For example for the *close* transition, we want to check whether it is possible to have a closed account where the balance is not equal to zero (as in [Section 3.2](#)), for the transition *withdraw* we want to check whether a negative balance can be achieved with the transition. Again, with checking traces are not available when a state is not reachable. In this case, traces cannot be used with opposite preconditions since they are not provided.

To conclude, with this approach we are testing the opposite of the preconditions. Thus what is not possible according to the specification is tested.

4.1.1 Evaluation criteria

Bugs

Since we are testing with this approach the opposite of the preconditions, thus what should be not possible according to the specification. It is expected to find bugs in the SUT where it is possible to perform the opposite of a transition. So faults can be found like preconditions which are not properly generated. An example of this is the manually created bug ([Table 3.1](#)) for the lightweight version.

Efficiency

In this approach is checking used to check what is not possible according to the specification. Therefore, the same tested transition should be tested in the SUT. To test all transitions from the account specification, it may take longer since some transactions require an initial state for which transitions need to be performed to reach this state.

Coverage

The experiment is going to generate a test for all transitions. Therefore, it is expected to test all the transitions of a specification. With the criteria bugs, we discussed the expectation to find faults in not properly generated preconditions. This may lead to the inability to test transitions. For example, when a failure (incorrect preconditions) occurs during reaching the initial state of the *withdraw* transition. This leads to the inability to test the *withdraw* transition.

4.2 Approach

The testing approach is a well-known approach in mutation testing. Mutation testing is a fault-based testing technique, which generates faulty programs by syntactic changes to the original program. [7, p. 1] The set of faulty programs are called mutants, each mutant contains a different syntactic change. In our case, only one mutant is generated. A test suite for a program is used to determine whether the faulty programs are detected. A mutant is killed when it is detected by the test suite. The mutant is in our case killed when the result from the SMT solver and the SUT are the same. we are using the same approach from [Chapter 3](#) to compare the results of the SMT solver and SUT.

Mutation testing generates a mutant based on mutation operator, which is a transformation rule that generates a mutant from the original program. [7, p. 3-4] The mutation operator for our approach is Negate Conditionals Mutator [23], this operator belongs to the type relational operator replacement [24, p. 688].

The testing approach is illustrated in [Figure 4.1](#). The first step is to start with a *Rebel* specification, which is in our case the already existing account specification. When the specifications are defined, the specifications are being built, *i.e.*, Concrete Syntax Trees (CSTs) are produced of these specifications. Using these CSTs, the code generator generates the code, which is then the SUT.

The test case generator can be used to test the SUT when the SUT is generated from the CSTs. The CSTs of the specifications are traversed by the test case generator to generate a test for each transition. The test case generator generates tebl files for transitions to use checking.

To test the SUT, the test case generator performs a similar transition as used within checking in the SUT. Finally, the results from checking and the performed transition in the SUT are compared.

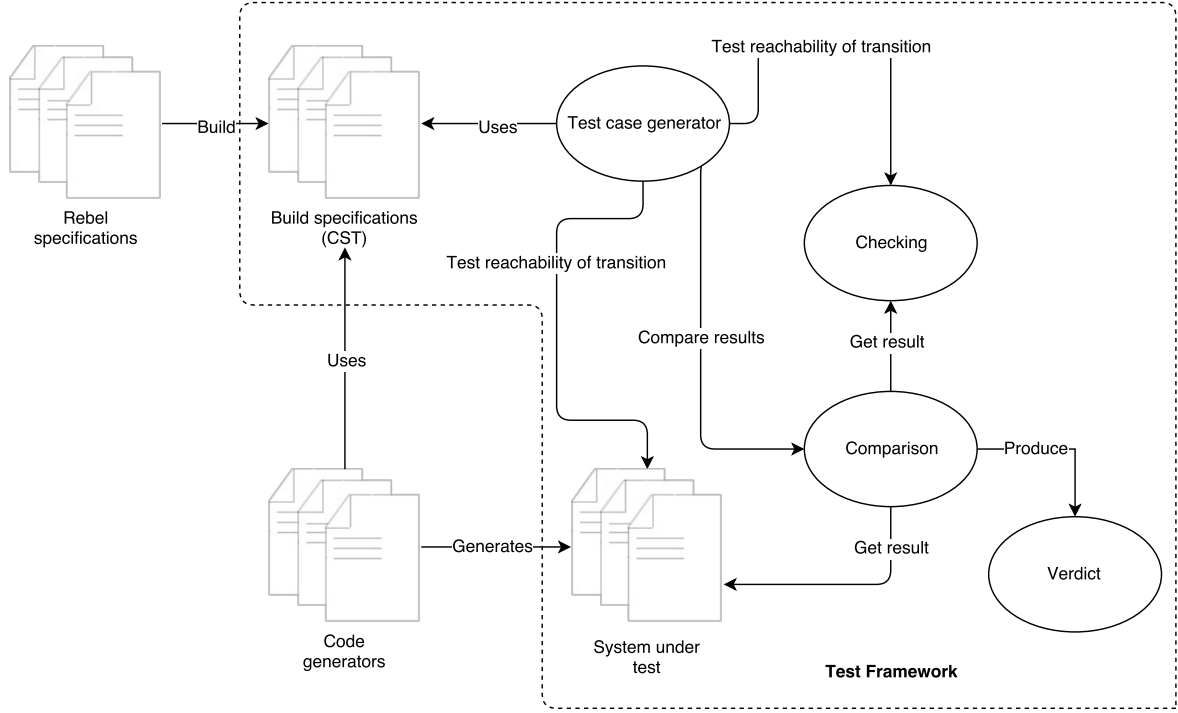


Figure 4.1: Testing approach with checking

4.2.1 Checking

Only expressions which contain a reference to the specification fields needs to be replaced since it is only possible in tebl to specify the reachable state with the properties of interest (these properties are not part of the transition).

Earlier the definition of the *close* event transition was given in Listing 3.5 which contains the following statement `this.balance == EUR 0.00; .` When this statement is translated to tebl with a negated conditional, it looks as follows `balance != EUR 0.00; .` Thus the replaced conditional is the opposite condition of the statement defined in the *close* transition.

Replacing conditionals to negated conditionals is done for all conditionals with relational operators. The chosen mutation operator Negate Conditionals Mutator will replace conditionals according to the replacement table in Table 4.1.

Actual expression	Translated expression
!=	==
==	!=
>	<=
>=	<
<	>=
<=	>

Table 4.1: Conditionals replacement [23]

4.2.2 Generated system

The conditionals for the transition in the SUT are also replaced. Although, it is not necessary to replace always the conditionals. For some transitions is an initial state required, *e.g.*, to execute the

transition unblock of the account specification, the account should be in the state blocked. So an initial state needs to be constructed for some transitions. Of course, the SMT solver can construct its initial state to reach a state.

The definition of the *deposit* transition is given in Listing 4.10 and contains the following statement in the preconditions `amount > EUR 0.00; .` First, the initial state needs to be constructed which is the state opened. Following is the replacement of the conditionals, the replaced precondition from the *deposit* transition looks as follows `amount > EUR 0.00; .` The *deposit* transition needs then to be performed in the SUT.

To perform the *deposit* transition, the transition parameters for this transition must be determined satisfying replaced conditionals. The transition parameter for the *deposit* transition, amount, should be less than or equal to 0 euro. Therefore, are generators implemented to generate values satisfying the negated conditionals. For example, the following transition parameter is generated to be used in the *deposit* transition `"amount": "EUR -2.00" .`

```

1 event deposit(amount: Money) {
2   preconditions {
3     amount > EUR 0.00;
4   }
5   postconditions {
6     new this.balance == this.balance + amount;
7   }
8 }
```

Listing 4.10: *deposit* event definition from specification

4.3 Results

4.3.1 Codegen-Akka

For this experiment, we are testing the generator Codegen-Akka. The results of this test run are shown in Table 5.1. As shown in this table, the tests for four transitions are successful and the tests for the other three transitions are failed.

Transition to test	Transition
openAccount	✓
withdraw	✗
deposit	✗
interest	✓
block	✓
unblock	✓
close	✗

Table 4.2: Results: testing account specification transitions

4.4 Analyse

4.4.1 Codegen-Akka

Bug: closing an account with balance

When this automated version of checking is executed, it produces some false positives. After investigating the tests for the transitions, the test for the *close* transition seems not be successful (see

[Listing 4.11](#)). The model checker states that the state is not reachable (the same tebl file is generated as in [Listing 3.8](#)). It seems to be that the state is reachable in the SUT.

```
1 Test transition close
2 opened -> close -> closed
3 generated close test in |project://rebel-core/examples/simple.transaction/
4   OpenedToClosedViaCloseTest.tebl|
5
6 Reachability transition: false
7 Execute transition result: true
8 Result successful transition test: false
```

Listing 4.11: Result run

When we take a look at the account in the SUT, it looks as follows in [Listing 4.12](#). The state of the account is in closed, which is correct according to the specification, but the balance of the account is 52 euro. In [Listing 3.5](#) we already discussed the event definition of the *close* transition, which is that the balance should be equal to zero. From this, we can conclude that we have discovered a bug in the SUT.

```
1 [{
2   "_id": 17592186045441,
3   "_version": 2,
4   "_status": "CLOSED",
5   "accountNumber": {
6     "iban": "NO3627716652225"
7   },
8   "balance": {
9     "value": 52.00,
10    "currency": "EUR"
11  }
12}]
```

Listing 4.12: account state in json

Now we know that we have discovered a bug, we want to know why this behaviour occurs and whether it is due to the generated code from the specification. The method which handles the *close* transition has the following check in [Listing 4.13](#). The if statement checks whether the balance of the account is not equal to 0 euro. The condition in the if statement is not satisfied with the balance of 52 euro. That is why the exception *BuildCASTransactionException* is not thrown.

```
1 if(! (isNotEqual(_entity.getBalance(), Money.of(org.joda.money.CurrencyUnit.of("EUR"), 0.00)))) {
2   throw new BuildCASTransactionException("Predicate did not hold: CloseTransaction: this.balance ==
3   EUR 0.00");
4 }
```

Listing 4.13: Code in Java

The question right now is, how is the above code generated. After taking a look at the synthesization of expression, the expressions from *Rebel* are not properly synthesized. The synthesization for an equal expression for the type *Money* or *Percentage* looks as follows in [Listing 4.14](#). The expression is synthesized to the method *isNotEqual* with two parameters.


```

1 private str g(e:(Expr)'<Expr lhs> == <Expr rhs>', tmap t) = "isNotEqual(<g(lhs, t)>, <g(rhs, t)>)"
2   when isType(t, lhs, (Type)'Percentage') || isType(t, lhs, (Type)'Money');

```

Listing 4.14: Generate equal expression in Rascal

So the expression is not properly synthesized, and it should be synthesized to *isEqual* instead of *isNotEqual*. With this modification, it is not possible anymore to close an account with some balance. This also applies to other statements which use the equal operator.

Bug: deposit with a maximum amount

The automated checking is implemented with the ability to first start the SUT and then run the tests against it. For a new test run, the specification has changed a little bit. It is now possible to only deposit with a maximum amount (see [Listing 4.15](#)). After the code is generated, the testing framework is not able to start the system. There is a compile error as you can see in [Listing 4.16](#), the binary operator "<" is not applicable on the type *org.joda.money.Money*. The compile error is thrown by the source code from [Listing 4.17](#), which is part of the method which handles the *deposit* transition.

```

1 event deposit(amount: Money) {
2   preconditions {
3     amount < EUR 250.00;
4   }
5   postconditions {
6     new this.balance == this.balance + amount;
7   }
8 }

```

Listing 4.15: *deposit* event definition from specification

```

1 Error:(63, 23) java: bad operand types for binary operator '<'
2   first type:  org.joda.money.Money
3   second type: org.joda.money.Money

```

Listing 4.16: *deposit* event definition from specification

```

1 if(! ((amount < Money.of(org.joda.money.CurrencyUnit.of("EUR"), 200.00)))) {
2   throw new BuildCASTransactionException("Predicate did not hold: DepositTransaction:
3     amount < EUR 250.00");
4 }

```

Listing 4.17: Code in Java

The functions for the synthesization, which generates a part of [Listing 4.17](#), are shown in [Listing 4.18](#). Also here are the *Rebel* expression not properly synthesized. The default expression with the binary operator "<" is properly synthesized to an expression with three expressions, the left-hand and right-hand side expression and the binary operator "<". As discussed before, the binary operator "<" doesn't work with *org.joda.money.Money*. Thus the default method to synthesize expressions with the binary operator "<" cannot be used for the type *org.joda.money.Money*.

On line number 1 of [Listing 4.18](#) is the synthesization method of the expression with the binary operator ">" shown. This method is already defined before in the corresponding file. To conclude, this method should synthesize expressions with the binary operator "<".

```
1 private str g(e:(Expr)'<Expr lhs> \> <Expr rhs>', tmap t) = "isGreaterThan(<g(lhs, t)>, <g(rhs, t)>)"
2   when isType(t, lhs, (Type)'Percentage') || isType(t, lhs, (Type)'Money');
3 private str g(e:(Expr)'<Expr lhs> \< <Expr rhs>', tmap t) = "<g(lhs, t)> \< <g(rhs, t)>";
```

Listing 4.18: Generate equal expression in Rascal

4.5 Evaluation

4.5.1 Bugs

In [Subsection 4.1.1](#) we discussed the expectations of the criteria bugs. We expected to find bugs in the SUT where it is possible to perform the opposite of a transition. Thus it is expected to find bugs where the preconditions are not properly generated.

With this experiment, we have found a bug in the SUT, which was discussed in [Section 4.4.1](#). The other bug is out of scope since the SUT is not able to compile. With the bug from [Section 4.4.1](#), it is possible that the final state closed is reached where the preconditions of the *close* transition do not hold. So as expected, we did find a bug in performing the opposite of a transition where the preconditions were not properly generated.

In this experiment traces are not used because they cannot be provided by the solver when a state is not reachable. The expectation is that with testing the opposite preconditions, not reachable states, traces are not provided. Remarkable is that with testing some transition, the traces are provided, because the state to reach with checking are reachable. For example, the *block* transition has no precondition which means that the state to reach is reachable with checking.

4.5.2 Efficiency

For the criteria efficiency, it is expected to check what is not possible according to the specification, i.e. testing the same transition in checking as well as in the SUT.

A part of the generated test for a transition is checking, which is used to test the state to reach with the replaced preconditions. So, in this experiment, we are testing what should be not possible according to the specification. The expectation is that the same transitions with checking should be performed on the SUT. However, the result of the checking from the SMT solver varies, *e.g.*, an opened account can be reached by the *openAccount* transition or by the transition *openAccount* and *withdraw*. Thus the test framework is not able to perform the same transitions on the SUT as the transitions from checking.

With testing all transitions from the account specification, it is possible that testing may take longer. As expected, this is the case since due to the initial state transitions are more executed and tested. To conclude, the testing process may take longer to test all the transitions.

4.5.3 Coverage

It is expected for this criteria to test all the transitions of the specification since the experiment generated tests for all transitions.

In the experiment, after the checking, a transition is performed in the SUT. In this experiment, it is unknown whether the performed transition with its parameters in the SUT is the same as the transition computed by the SMT solver. This causes some false positives in the test run. Also, it is difficult to play like the SMT solver; it is unknown which result the SMT solver will give. The SMT solver is also smarter/better in checking the satisfiability of a given constraints.

Failure occurring along the way in constructing the initial state of a transition may lead to the inability to test transitions. Unfortunately, there does not seem to be any faults in here.

4.6 Conclusion

In this experiment is the account specification used to test the SUT. This experiment generates automatically tests for transitions.

A part of the generated test for a transition is checking, which is used to test the state to reach with the replaced preconditions. So, in this experiment, we are testing what should be not possible according to the specification. The result of the checking from the SMT solver varies, *e.g.*, an opened account can be reached by the *openAccount* transition or by the transition *openAccount* and *withdraw*.

After the checking, a transition is performed in the SUT. In this experiment, it is unknown whether the performed transition with its parameters in the SUT is the same as the transition computed by the SMT solver. This causes some false positives in the test run. Also, it is difficult to play like the SMT solver; it is unknown which result the SMT solver will give. The SMT solver is also smarter/better in checking the satisfiability of a given constraints.

To conclude, the checking used in this experiment tests only the states, regardless of which transitions are being performed, and testing the SUT focuses more on testing transitions.

With this experiment, we have found a bug in the SUT, which was discussed in [Section 4.4.1](#). The other bug is out of scope since the SUT is not able to compile. The found bug belongs to the category injected code since the generated code for the precondition is wrong. In this case, the final state closed is reached where the preconditions of the *close* transition do not hold.

4.7 Threats to validity

Limited specifications

In the conducted experiment is the account specification account used to test the SUT. With this experiment and specification, we did find a bug in the code generators.

The used account specification in this experiment is quite simple. With the use of more interacting specifications, the chance is bigger to find faults in the code generators since the specifications are interacting with each other.

Invalid execution trace

The conducted experiment test only what should be not possible according to the specification. It is also important to test whether the SUT is conform to the specification, *i.e.*, testing the valid execution trace.

Chapter 5

Experiment 2: Valid execution traces

In the experiment from [Chapter 4](#), we designed a tool to test generated systems, *i.e.* testing invalid execution trace. However, testing valid execution trace can also provide valuable results to test whether the generated system conforms to the specification. In this chapter, we discuss how we are testing the valid execution traces and how to solve the limitations of the experiment from [Chapter 4](#).

5.1 More complex specifications

In the previous approaches is only the account specification used. In this experiment, we are going to use more complex specifications, complex in the sense that they depend and interact with each other.

We are going to use the same account specification from [Listing 3.4](#). As an addition to account specification, we use a transaction specification. Via this specification money can be transferred between two accounts. The *Rebel* implementation of the transaction specification ¹ is shown in [Listing 5.19](#). As shown in the transaction specification, it contains more fields than the account specification. The two remarkable fields are from and to, both are of the *IBAN* type. The type *IBAN* is a built-in *Rebel* type. [1, p. 3] Note that after the type definition an annotation is given to specify a reference to another specification, in this case, account specification. The fields to and from are being used to indicate between whom the transaction takes place.

According to the transaction specification, the transaction first needs to be started. When a transaction is in the state validated, and a booking cannot be made, the transition *fail* can be used to put the transaction in its final state failed. To successfully complete a transaction is the transition *book* used. In comparison to the account specification, does the transaction specification two final states, which are booked and failed. When the final state booked or failed is reached, then there is no further action allowed. Note that the transaction specification does not have an invariant.

Another difference in the transaction specification is that event definitions can contain sync expressions. From the previous event definitions, we have only seen pre- and postconditions. Sync expressions are used for synchronisation. These sync expressions are also translated to SMT formulas.

The sync expressions translated to the SMT solver are also logical formulas. These formulas do not have logic about synchronisation. Of course, the SUT has implemented synchronisation for these transitions. So it is possible to also test synchronisation in the SUT. There are also several studies which report that SMT-based approaches to model checking can be used to test distributed algorithms. [25, 26, 27]

The *book* transition uses the synchronisation feature to express sync operations (see [Listing 5.20](#)). A sync operation is here used to withdraw an amount from one account and to deposit to another account.

¹https://github.com/cwi-swat/rebel/blob/e58590c7f51f59e7ee6443bb89ef09dff6febab6/rebel-core/examples/simple_transaction/Transaction.ebl

```
1 specification Transaction {
2   fields {
3     id: Integer @key
4     amount: Money
5     from: IBAN @ref=Account
6     to: IBAN @ref=Account
7   }
8
9   events {
10    start []
11    book[]
12    fail []
13  }
14
15  lifeCycle {
16    initial uninit -> validated: start
17    validated      -> booked: book
18                  -> failed: fail
19    final booked
20    final failed
21  }
22 }
```

Listing 5.19: Transaction specification

```
1 event book() {
2   sync {
3     Account[this.from].withdraw(this.amount);
4     Account[this.to].deposit(this.amount);
5   }
6 }
```

Listing 5.20: *book* event definition from transaction specification

5.2 Method

As discussed in [Subsection 1.1.3](#), a model testing approach is already done to test existing banking systems. Although, in this approach, it was only possible to test the SUT interactively using the simulation. In this approach are the traces from the SMT solver used to check whether the SUT accepts the execution from the trace and whether it behaves as the specification. [1, p. 5]

By using the traces, it also solves the problems from the previous experiment. With the use of traces, we know exactly which possible transitions the SMT solver has performed. Then these transitions can be performed in the SUT. So, in this approach, we are going to use the traces to check the behaviour of the SUT.

5.2.1 Evaluation criteria

Bugs

In this approach we are using the traces from the SMT solver to test the SUT, thus testing what should be possible according to the specification. The expectation here is to find bugs in the SUT which does not accept the execution from the traces. For instance, the generated pre- or postconditions are not satisfied by the transition from the traces or the generated postcondition which leads to different results. An example of this is the bug that we have found in [Section 4.4.1](#).

In [Section 5.1](#) we discussed that we are going to use more complex specification which implements synchronisation. As discussed, it is possible to test synchronisation in the SUT. Thus the expectation is to find faults in the implementation of the synchronisation in the SUT.

In [Subsection 2.1.2](#) we discussed the Codegen-Akka generator. The SUT from the generator uses Akka as Actor Model. Akka is a toolkit which is used for building concurrent, distributed and resilient message-driven systems. [16, p. 4]

It is difficult to build and test distributed systems, according to the study [27, p. 1], this has two main reasons: partial failure and asynchrony. When components fail along the way that results into incomplete results or data is called partial failure. Within a system, asynchrony is the non-determinism of ordering and timing. That is to say; the expectation is to find faults in partial failure and asynchrony. Not only is the expectation to find these faults in the Codegen-Akka, but also in the other code generators since these code generators implements also synchronisation. For instance, the Codegen-Javadatomic uses CQRS where the commands are executed asynchronously.

At the time of experimenting, a new version of the Codegen-Akka is released. The new version implements Two-phase commit (2PC), which is a synchronisation protocol [28, p. 3204]. A protocol like 2PC is also referred to as an Atomic commit protocol (ACP). The essence of ACP is to achieve global atomicity. The protocol ensures a unanimous final outcome of a distributed transaction, regardless of the failures that may occur. [28, p. 3204] In our case, the expectation is to find bugs in synchronisation where the SUT does not produce a unanimous final outcome. However, the traces from the SMT solver must also be unanimous to the SUT.

Efficiency

In the experiment from [Chapter 5](#), we have seen that with the use of checking, the result from the SMT solver varies. For the testing framework, it's unknown whether the performed transition is the same as the computed transition by the SMT solver. The SMT solver is smarter/better in checking the satisfiability of a given constraints. This, limitation from the previous experiment can be solved by using the traces from the SMT solver.

By using the traces for testing, the same path computed by the SMT solver are performed in the SUT and we can check whether the given execution is possible in the SUT. Thus the expectation is to perform the same transition from the traces should be performed in the SUT. However, it may take longer to test all transitions from specifications as each transition needs to be tested and some transitions require an initial state for which transitions need to be performed to reach this state.

Coverage

In this experiment, the simulation is used to test transitions. Since the simulation can test single steps, it is expected to test all transitions of a specification.

As discussed, this experiment is going to use more complex specifications, *e.g.*, specifications which depend and interact with each other. With the use of the simulation, it is expected to test transitions which depend and interact with other specifications.

With the criteria bugs, we discussed that it is expected to find faults in partial failure and asynchrony. These characteristics may lead to not testing transitions. To illustrate what is meant, let us look at the case of partial failure within the *book* transition. When components fail along the way during the transitions to reach the initial state for the *book* transition, it may result in incomplete results or data. This leads to the inability to test the *book* transition.

5.3 Approach

The process of the testing approach with the simulation is shown in [Figure 5.1](#). This process starts first with the creation of a specification. This experiment uses the account and transaction specifications which are already defined.

Once the specification is defined, the specifications are being built, *i.e.*, CSTs are produced of these specifications. Using these CSTs, the code generator generates the code, which is then the SUT.

When the SUT is generated from the CSTs, the SUT can be tested by the test framework. The test case generator traverses the CSTs of the specifications to generate test cases for all transitions. Using the information from the CSTs, the simulation is requested for a test case for a chosen transition. The simulation provides then a trace (test case) for the requested transition.

To test the SUT using the generated test cases, the data from these test cases are provided to the test case adapter. Finally, the test case adapter communicates this data to the SUT.

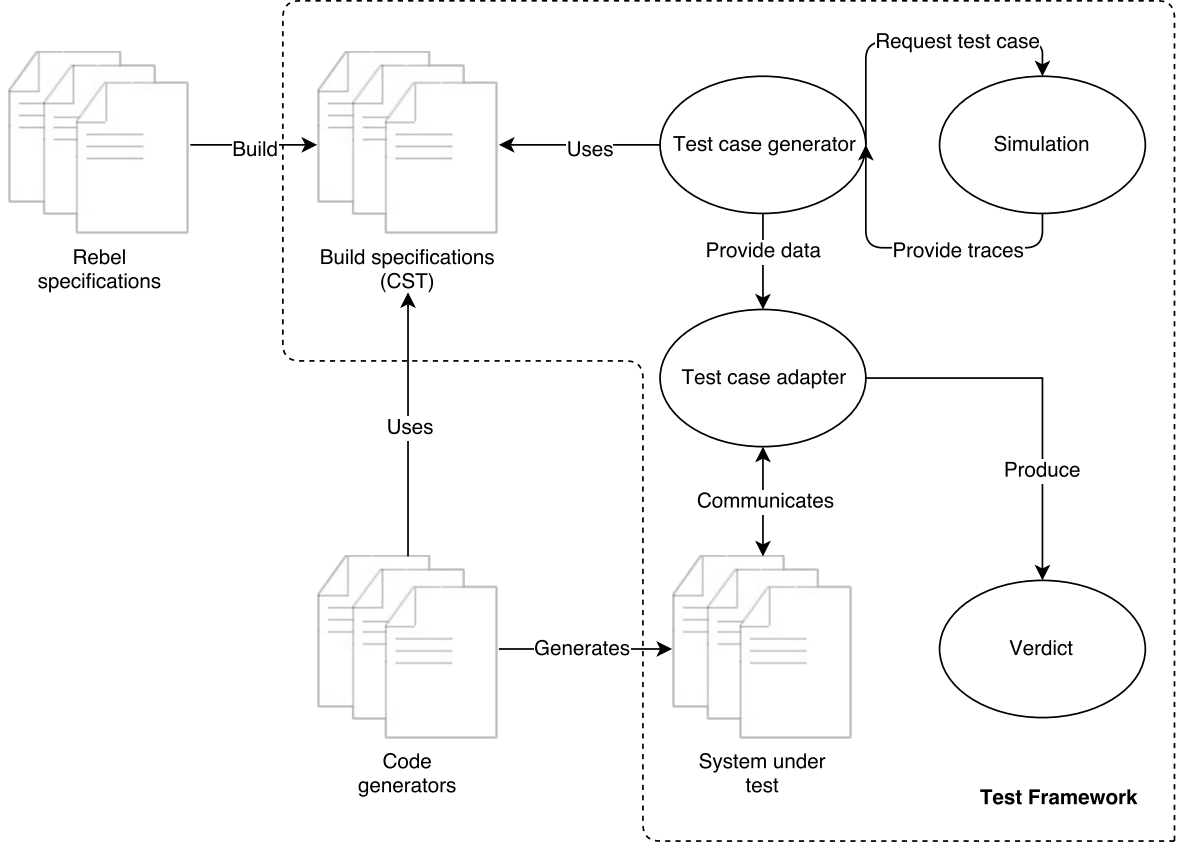


Figure 5.1: Testing approach with simulation

To playback the steps from a given trace on the SUT, we are going to split every transition into three steps as in the study [1, p. 6]:

1. **pre-transition check** Check whether the current state from the SUT is conform to the current state from the trace
2. **transition check** Execute the given transition from the trace on the SUT
3. **post-transition check** Check whether the new state from the SUT is conform to the new state from the trace

The transition function for the simulation looks as follows: $p(s_1, s_2)$, which has the pre- and post-condition of the to be executed transition [1, p. 6]. The current state s_1 holds the constraints of the current values of the simulated specification. Hereafter, to execute the transition, the user is asked to provide the data for the transition parameters.

Before using the simulation, two challenges need to be solved: defining the current state for a given transition and providing the transition parameters data values. These challenges are discussed in the paragraphs below.

5.3.1 Pre-transition check

Current state

The pre-transition checks entail the check for the current state of the SUT is conform to the current state from the trace. Although with the simulation it is only possible to reason about individual steps. For some transitions a current state is required, *e.g.*, an account needs to be blocked first to unblock it. To make use of the simulation, the current state needs to be defined to check whether the step can be made from the current state. This is also the case in the SUT since a current state needs to be initialised first before the transition is performed.

To initialise the current state in the SUT checking can be used. By defining the current state with checking, the current state is checked, and a valid trace is given by the model checker when the current state is satisfiable. So for every transition, a tebl needs to be generated for the state to reach (current state) to perform the transition. Although there are some caveats with the use tebl.

When a state to reach is defined for checking, the identifiers for the entities are unique within that trace. For example for [Listing 2.3](#), the identifier for the opened account from the traces is *NL10INGB00000001*. The identifiers for similar entities are auto-incremented, *e.g.*, when an additional account entity is specified in [Listing 2.3](#) the identifier is then *NL10INGB00000002*. This is not only the case with IBAN numbers but also with Integer, String, etc. This is not a problem when a single transition is tested. When multiple transitions are tested and even when there are multiple entities, this will result in collisions of existing entities with the same identifiers. Note also that the generated IBAN numbers by the SMT solver are also not valid.

The invalid IBAN numbers is not a problem for the checking since checking is only used to reason about possible traces. However, this does not hold in the SUT since SUT is a banking system and here it must conform to the IBAN standards. It is possible with checking to define the properties of an entity, *e.g.*, the IBAN for an account.

To solve the problem with the collision of existing entities, a random identifier should be given for each entity. Therefore, for each type like Integer, is a random generator implemented. This generator generates a random identifier which is used as the identifier for the entities in checking. Unlike the basic types, IBAN is a more complex type to generate since the type should conform to the IBAN standards. Therefore, `Iban4j` [29] is used to generate random IBAN account numbers which are compliant to the ISO_13616 and ISO_9362 standards.

Another pitfall with the use of checking is the use of more complex specifications, *e.g.*, the transaction specification. With such specifications, it is possible to have a reference to another specification. In the case of the transaction specification involves two references, namely account specification. To use checking in such specifications must be taken into account the references to other specifications. Which means that for every referencing specification in checking, imports need to be resolved and defined, and the entity needs to be defined with an identifier. Note that these identifiers should be again random. Otherwise, it will cause collisions of existing entities.

Altogether, a generated tebl for checking the current state looks as follows in [Listing 5.24](#). Note that the configurable time-out is set to four since the states from the account and transaction specification are reachable within four steps. The determination of the configurable time-out per specification is left as future work.

When the current state is defined, the generated tebl is given to the model checker. When the current state is satisfiable are the traces used to perform the transitions in the SUT. This is in more depth described in the section [Section 5.3.2](#). Performing each step from a trace entails the three steps to test and perform the transition.


```

1 module simple_transaction.Test
2
3 import simple_transaction.Transaction
4 import simple_transaction.Account
5
6 state doCheck {
7   validated Transaction with id == 65227;
8
9   Account with accountNumber == NO3631174980518;
10  Account with accountNumber == LB404150J311SB1FJV5KL1MKYAY4;
11 }
12
13 check doCheck reachable in max 4 steps;

```

Listing 5.21: Generated tebl for the transition book

Current state values

After the current state is defined, the same approach from the previous chapter can be used to check whether it is satisfiable. The SMT solver returns whether the current state is satisfiable with the traces, including the instances with its values.

To provide transition parameters data to the simulation is the last step taken with its instances. The last step is taken because the last step is the transition which led to the current state, which is going to be used by the simulation. The values from these instances of the simulated specifications are given to the current state s_1 .

The simulation checks only whether the step can be made from the given current state, it does not check whether the current state is reachable. With the use of checking for the current state is the current state checked whether this is satisfiable.

5.3.2 Transition check

Transition parameters data values

Since simulation is used for reasoning about individual steps, that explains why the transition parameters data values for the chosen transition should be provided by the user. As discussed earlier is in the model testing approach of the study [1, p. 6] chosen for interactively using the simulation.

Manually providing the transition parameters data values is not relevant for us, since we intend to test automatically the SUT. Certainly, these values can be generated randomly, but it should satisfy the pre- and postconditions for the chosen transition. However, it would be better to let the SMT solver fill these values, taking into account the pre- and postconditions of the chosen transition. At the time of the publication of the study [1], this was not possible in the simulation, so the simulation should be slightly modified.

In cooperation with the author of the study [1] is this feature added to the simulation.² The simulation is now able that both state variables and transition variables can be left open, in the sense of using the expression *ANY*. When the expression *ANY* is used, the SMT solver will fill a value for the corresponding transition parameter, satisfying the pre- and postconditions for the chosen transition.

what about
expression as
iban?

Traces

A valid trace is a chain of valid transitions from one state to the next state [1, p. 5]. The trace may contain multiple steps, *e.g.*, to unblock an account, an account needs first to be opened and blocked. This requires after executing each step; the step needs to be tested, in the sense of automatically repeating the testing process. Each step from the traces has instances with its state and values.

The instances are specific for that step, the results after performing the step are given in the state

²<https://github.com/cwi-swat/rebel/commit/0d29eb30a82cc5dd6d8be750daa4a24e4e2786be>

and values for instance. A trace for simulation contains the state before performing the transition, the step (transition), and then the state after performing the step.

After everything is defined for the simulation, the simulation can test whether it can make the single step. The simulation then works as follows [1, p. 6]:

1. Check whether it is possible to satisfy the constraints of the chosen transition given the current state and the transition parameters data values of the event: $p(s_1, s_2)$
2. Check whether the invariants hold in the state after the transition: $P(s_2)$.

When this step can be made, with the use of traces, the step can be played on the SUT. Therefore, *JSON* should be generated for the chosen transition. This is the responsibility of the test case adapter, which is explained more in depth in Section 5.3.3. To generate the *JSON*, the transition parameters data values and the transition name are read from the trace step. At last, the endpoint is determined for the given step, and the generated *JSON* is sent to the endpoint.

5.3.3 Post-transition check

Since a trace contains the instances before and after the step, this can be used in the post-transition check to test the SUT. As a result, we can check after or before performing the step, whether the SUT behaves similarly as the simulated/checked specification.

To check whether the new state from the SUT is conform to the trace, the state and values from the instances are read after performing the transition. The endpoint is then determined to retrieve the state of these instances in the SUT. Then, the results of these instances from the SUT and trace are compared to find any misbehaviour in the SUT.

check all instances

Normalisation

Before a *Rebel* specification is checked or simulated is the specification normalised. This normalisation process is done to make the SMT formulas easier and also to give it partially semantics [1, p. 5]. Desugaring the life cycle is part of the normalisation process. The life cycle is desugared to strengthen the pre- and postconditions of the transitions with the life cycle information. Therefore, are two fields added, `_state` and `_step`, to the fields of the specification. To each state and event is a distinct identity assigned. The identity of the current state is assigned to the `_state` field, and the identity of the transition which led to the current state is assigned to the field `_step`. This results into that the original life cycle can be expressed, by adding constraints on the `_state` and `_step` fields to the transitions pre- and postconditions [1, p. 5].

The newly added fields are also present within the trace from the SMT solver. In our case, we only use the `_state` field, because we already know which transition led to the current state. To compare the current state from the `_state` field, it needs to be sugared back to compare it to the SUT.

Test case adapter

As discussed in the code generation section, the request made to the API of the SUT are "standardized", but the response from the SUT is not. For the post-transition check, it is necessary to check the new state in the SUT. The model and SUT are at different level of abstraction, and these different levels need to be bridged. [30] Therefore, an adapter needs to be defined to communicate the results of the traces with the SUT.

An adaptor is often used in model-based testing approaches. [8, 31] The adapter is a component that concretises the test inputs and abstraction of test outputs, *i.e.*, the adapter adapts the abstract test data to the concrete SUT interface. [8, p. 4] On request of the test case generator, the adaptor provides inputs to, and receives outputs from the SUT. The adapter encodes and decodes the abstract actions from the test case adapter to concrete actions for the SUT. [31, p. 5] This leads to that the adapter is dependent on the specification and the SUT.

According to the study [8, p. 4], an adaptor is a concept, and that does not mean that an adapter needs to be a separate software component. The adapter may be integrated within the test scripts. The test script is code which executes a test case, abstracts the response from the SUT and creates

the verdict. In our case, the test case adapter is integrated into the test script, which is the test case generator.

The decision is made to only implement a test case adapter for the code generator Codegen-Akka since this is a more mature code generator and frequently used for experiments within ING Bank.

5.4 Results

5.4.1 Codegen-Akka

The results of the test run are shown in [Table 5.1](#) and [Table 5.2](#). From this result, we can conclude that the test for the *close* transition has failed.

Transition to test	Current state	Transition
openAccount	✓	✓
withdraw	✓	✓
deposit	✓	✓
interest	✓	✓
block	✓	✓
unblock	✓	✓
close	✓	✗

Table 5.1: Results: testing account specification transitions

Transition to test	Current state	Transition
start	✓	✓
book	✓	✓
fail	✓	✓

Table 5.2: Results: testing transaction specification transitions

5.4.2 Codegen-Javatomic

The results of the test run are shown in [Table 5.3](#) and [Table 5.4](#). From this result, we can conclude again that the test for the *close* transition has failed. Remarkable is that also the *interest* transition has failed.

Transition to test	Current state	Transition
openAccount	✓	✓
withdraw	✓	✓
deposit	✓	✓
interest	✓	✗
block	✓	✓
unblock	✓	✓
close	✓	✗

Table 5.3: Results: testing account specification transitions

Transition to test	Current state	Transition
start	✓	✓
book	✓	✓
fail	✓	✓

Table 5.4: Results: testing transaction specification transitions

5.4.3 Codegen-Scala-ES

The results of the test run are shown in [Table 5.3](#) and [Table 5.4](#). Again the test for the transition *close* has failed, but also for the transition *interest*.

Transition to test	Current state	Transition
openAccount	✓	✓
withdraw	✓	✓
deposit	✓	✓
interest	✓	✗
block	✓	✓
unblock	✓	✓
close	✓	✗

Table 5.5: Results: testing account specification transitions

Transition to test	Current state	Transition
start	✓	✓
book	✓	✓
fail	✓	✓

Table 5.6: Results: testing transaction specification transitions

5.4.4 Distributed Codegen-Akka

In [Subsection 5.4.1](#) we have seen the test results of the Codegen-Akka generator. In this test run is the SUT run as single nodes, both the Scala system and Cassandra.

With this test run, we will run the SUT distributed generated by the Codegen-Akka generator. The results of the test run are shown in [Table 5.7](#) and [Table 5.8](#). It is noteworthy that all tests fail.

Transition to test	Current state	Transition
openAccount	✗	✗
withdraw	✗	✗
deposit	✗	✗
interest	✗	✗
block	✗	✗
unblock	✗	✗
close	✗	✗

Table 5.7: Results: testing account specification transitions

Transition to test	Current state	Transition
start	<i>X</i>	<i>X</i>
book	<i>X</i>	<i>X</i>
fail	<i>X</i>	<i>X</i>

Table 5.8: Results: testing transaction specification transitions

5.5 Analyse

5.5.1 Codegen-Akka

The experiment uses in this test run the Codegen-Akka generator. Investigating the test run, it seems to be that all transitions are tested successfully, except the transition *close*. Also, there seems to be a limitation in testing the state of a specification.

Close transition

The result of *close* transition test is shown in [Listing 5.22](#) and [Listing 5.23](#). As shown on line number 3 of [Listing 5.23](#), constructing the current state for the *close* transition is successful. Then the simulation is asked to simulate the *close* transition, but according to the traces of the simulation, the simulation was not able to make the step. The simulation returns only the state before the transition.

```
1 Test transition close
2 opened -> close -> closed
3
4 0:
5   now = 14 Aug 2017, 13:49
6
7   instance: simple.transaction.Account, key = FO9402337176862639
8   ?
9   ?
10  var accountNumber (type: IBAN) (uninitialized)
11  var balance (type: Money) (uninitialized)
12
13 1:
14  now = 14 Aug 2017, 13:49
15  step: simple.transaction.Account.openAccount
16    var initialDeposit (type: Money) = EUR50.00
17    Transition to state = opened
18    Identified by accountNumber = FO9402337176862639
19
20 instance: simple.transaction.Account, key = FO9402337176862639
21 State = opened
22 ?
23 var accountNumber (type: IBAN) = FO9402337176862639
24 var balance (type: Money) = EUR50.00
```

Listing 5.22: No test generated for *close* transition (part 1)

```

1 Endpoint: /Account/FO9402337176862639/OpenAccount
2 JSON payload: { "OpenAccount": { "initialDeposit": "EUR 50.00" } }
3 Response: ("body": "CommandSuccess(OpenAccount(50.00 EUR))", "isSuccessful": "true", "message": "OK",
4 "errorBody": "", "code": "200")
5
6 1:
7   now = 12 Jul 2016, 12:00:00
8
9   instance: simple_transaction.Account, key = FO9402337176862639
10   State = opened
11   ?
12   var accountNumber (type: IBAN) = FO9402337176862639
13   var balance (type: Money) = EUR50.00

```

Listing 5.23: No test generated for *close* transition (part 2)

As discussed earlier, the precondition of the *close* transition is that there should be no remaining balance as shown in Listing 3.5. On line number 15 of the test run is shown that the simulation makes the step to open an account with a balance of 50 euros. Afterwards are no transitions performed. Thus this current state does not satisfy the precondition of the *close* transition. That is the reason why the simulation was not able to perform the transition since the given values from the current state to s_1 were not satisfying for s_2 .

The generated tebl for the current state is shown in Listing 5.24. Only the state to reach is specified with the identifier of the account. As we have seen the event definition of *openAccount* in Listing 2.2, the account must be opened with a balance of 50 euros.

To conclude, the simulation was not able to test the transition *close* since the precondition of this transition is not satisfied in the postcondition of s_1 . This also holds for testing SUTs by other code generators. In other words, this limitation is the result of the test approach

```

1 module simple_transaction.Test
2
3 import simple_transaction.Account
4
5 state doCheck {
6   opened Account with accountNumber == AD3517248539N3OTXZIDF13H;
7 }
8
9 check doCheck reachable in max 4 steps;

```

Listing 5.24: Generated tebl for the transition book

State testing

In Listing 5.25 is the test run shown of the transition *book*. Only the first transition is shown in this figure. As you can see on line 42 is the *openAccount* transition performed. On the line below is shown that the request is successful. On line 44 is an error message shown which tells that it is not able to find the state of the opened account. Note the question mark in this message. This error message is part of the post-transition check where the new state of the SUT is tested. On line number 35 is the instance account shown after the transition and on the line below you can see the same question mark. Both question mark relates to each other, which is the state of a given specification.

In testing single specifications, *e.g.*, in Listing 5.22 and Listing 5.23, we have seen that the state is present from the result of the SMT solver. In this case, the model checker/simulation does not return the state when multiple instances are involved. This also happens in the *start* and *fail* transitions.

Listing 5.25: State not found for entities

into an account entity with the state opened and with a negative balance. To conclude, the transition *interest* is possible according to the simulation.

Although, the request made for the transition *interest* is not successful, and when we take a look at the account in the SUT, the account looks as follows in [Listing 5.28](#). The state of the account is still opened, and the balance seems to be the same when the account was opened. From looking at the state of the account, the *interest* transition is not performed in the SUT and the state of the account is the same as before performing the transition.

```
1 Test transition interest
2 opened -> interest -> opened
3
4 0:
5   now = 13 Jul 2017, 12:26
6
7   instance: simple.transaction.Account, key = MD14FLBLJOYGVJMDUZVKLU4C
8   ?
9   ?
10  var accountNumber (type: IBAN) (uninitialized)
11  var balance (type: Money) (uninitialized)
12
13 1:
14  now = 13 Jul 2017, 12:26
15  step: simple.transaction.Account.openAccount
16    var initialDeposit (type: Money) = EUR50.00
17    Transition to state = opened
18    Identified by accountNumber = MD14FLBLJOYGVJMDUZVKLU4C
19
20  instance: simple.transaction.Account, key = MD14FLBLJOYGVJMDUZVKLU4C
21    State = opened
22    ?
23    var accountNumber (type: IBAN) = MD14FLBLJOYGVJMDUZVKLU4C
24    var balance (type: Money) = EUR50.00
25
26 Endpoint: /Account/MD14FLBLJOYGVJMDUZVKLU4C/OpenAccount
27 JSON payload: { "OpenAccount": { "initialDeposit": "EUR 50.00" } }
28 Response: ("body": "{ \"iban\": \"MD14FLBLJOYGVJMDUZVKLU4C\" }", "isSuccessful": "true",
29   "message": "OK", "errorBody": "", "code": "200")
```

Listing 5.26: Failing test on *interest* transition with the use of javadatomic generator (part 1)


```

1 1:
2   now = 12 Jul 2016, 12:00:00
3
4   instance: simple.transaction.Account, key = MD14FLBLJOYGVJMDUZVKLU4C
5   State = opened
6   ?
7   var accountNumber (type: IBAN) = MD14FLBLJOYGVJMDUZVKLU4C
8   var balance (type: Money) = EUR50.00
9
10 2:
11  now = 12 Jul 2016, 12:00:00
12  step: simple.transaction.Account.interest
13    var currentInterest (type: Percentage) = (- 7709)
14    Transition to state = opened
15    Identified by accountNumber = MD14FLBLJOYGVJMDUZVKLU4C
16
17  instance: simple.transaction.Account, key = MD14FLBLJOYGVJMDUZVKLU4C
18  State = opened
19  ?
20  var accountNumber (type: IBAN) = MD14FLBLJOYGVJMDUZVKLU4C
21  var balance (type: Money) = - EUR3804.50
22
23  Endpoint: /Account/MD14FLBLJOYGVJMDUZVKLU4C/Interest
24  JSON payload: { "Interest": { "currentInterest": "-77.09" } }
25  Response: ("body": "", "isSuccessful": false, "message": "Bad Request", "errorBody": "", "code": "400")

```

Listing 5.27: Failing test on *interest* transition with the use of javadatomic generator (part 2))

```

1 {
2   "_id": 17592186045441,
3   "_version": 1,
4   "_status": "OPENED",
5   "accountNumber": {
6     "iban": "MD14FLBLJOYGVJMDUZVKLU4C"
7   },
8   "balance": {
9     "value": 50.00,
10    "currency": "EUR"
11  }
12 }

```

Listing 5.28: Account state in the SUT after performing the *interest* transition

The event definition for the *interest* transition is given in [Listing 5.29](#). This event definition states that the precondition is that the *currentInterest* must be less than or equal 10%, and the postcondition is that the balance must be changed after applying the interest. The generated transition parameter *currentInterest*, - 7709, satisfies also this precondition.

```

1 function singleInterest (balance: Money, interest: Percentage): Money = balance * interest;
2
3 event interest [maxInterest: Percentage = 10%](currentInterest: Percentage) {
4   preconditions {
5     currentInterest <= maxInterest;
6   }
7   postconditions {
8     new this.balance == this.balance + singleInterest(this.balance, currentInterest);
9   }
10 }

```

Listing 5.29: *Interest* event definition from account specification

Now we know that we have discovered a bug since the simulated transition is conform to the specification, we want to know where this misbehaviour occurs and which code is not correctly generated. The generated code for the *interest* event definition from Listing 5.29 contains the following check in Listing 5.30. The generated code for the preconditions seems to be good since it uses a *isLessOrEqualThan* function with the given interest percentage. Looking at the log file created by the SUT, the exception *BuildCASTransactionException* is thrown when performing the *interest* transition, which is the exception from Listing 5.30. So the generated code seems to be good, but the function used for validating the interest returns an inappropriate value, which throws the exception.

```

1 if (! (isLessOrEqualThan(currentInterest, 10 /* % */))) {
2   throw new BuildCASTransactionException("Predicate did not hold: InterestTransaction: currentInterest
3     <= 10%");
4 }

```

Listing 5.30: Code in Java

The function *isLessOrEqualThan* is shown in Listing 5.31. This function takes two parameters, both of the type *BigDecimal*, and compares the *lhs* to the *rhs*; this result should be greater or equal than zero. This function is not correctly defined since this is the definition of the function *isGreaterOrEqualThan*. This code is not generated but is part of the fixed code.

Clearly, we have discovered a bug in the SUT for the transition *interest*. As discussed in Section 3.6, it is possible to have bugs in the fixed code. With this experiment and testing the Javatomic generator, we can conclude that we have found a bug in the fixed code.

```

1 public static boolean isLessOrEqualThan(BigDecimal lhs, BigDecimal rhs) {
2   return lhs.compareTo(rhs) >= 0;
3 }

```

Listing 5.31: Code in Java

5.5.3 Codegen-Scala-ES

This test run tests the SUT generated by the Scala-ES generator. Looking at the test run, as the test runs for the Javatomic generator, it seems to be that the transition *interest* fails. The results of the test run are shown in Listing 5.32 and Listing 5.33. Line number 23 of Listing 5.33 shows the failing request for the *interest* transition; an error message is returned with the HTTP status code 400. Also, in this test run is construction the current state for the *interest* transition successful (see line number 26 of Listing 5.32). In this test run, the simulation has generated the same trace for the *interest* transition as for the Javatomic generator. The state of the account in the SUT looks as follows in Listing 5.34. Also here the state of the account opened, and the balance seems to be the

same when the account was opened. So, the performed *interest* transition has failed, and the state of the account is the same before performing the transition.

```
1 Test transition interest
2 opened -> interest -> opened
3
4 0:
5   now = 12 Aug 2017, 18:29
6   instance: simple_transaction.Account, key = MT58PDLQ09015VOS06LIF4Q525NRO1I
7   ?
8   ?
9   var accountNumber (type: IBAN) (uninitialized)
10  var balance (type: Money) (uninitialized)
11 1:
12  now = 12 Aug 2017, 18:29
13  step: simple_transaction.Account.openAccount
14    var initialDeposit (type: Money) = EUR50.00
15    Transition to state = opened
16    Identified by accountNumber = MT58PDLQ09015VOS06LIF4Q525NRO1I
17
18  instance: simple_transaction.Account, key = MT58PDLQ09015VOS06LIF4Q525NRO1I
19  State = opened
20  ?
21  var accountNumber (type: IBAN) = MT58PDLQ09015VOS06LIF4Q525NRO1I
22  var balance (type: Money) = EUR50.00
23
24 Endpoint: /Account/MT58PDLQ09015VOS06LIF4Q525NRO1I/OpenAccount
25 JSON payload: { "OpenAccount": { "initialDeposit": "EUR 50.00" } }
26 Response: ("body": "{ \"iban\": \"MT58PDLQ09015VOS06LIF4Q525NRO1I\" }", "isSuccessful": "true",
27 "message": "OK", "errorBody": "", "code": "200")
```

Listing 5.32: Failing test on *interest* transition with the use of Scala-ES generator (part 1)

```

1 1:
2   now = 12 Jul 2016, 12:00:00
3   instance: simple.transaction.Account, key = MT58PDLQ09015VOS06LIF4Q525NRO1I
4   State = opened
5   ?
6   var accountNumber (type: IBAN) = MT58PDLQ09015VOS06LIF4Q525NRO1I
7   var balance (type: Money) = EUR50.00
8 2:
9   now = 12 Jul 2016, 12:00:00
10  step: simple.transaction.Account.interest
11   var currentInterest (type: Percentage) = (- 7709)
12   Transition to state = opened
13   Identified by accountNumber = MT58PDLQ09015VOS06LIF4Q525NRO1I
14
15 instance: simple.transaction.Account, key = MT58PDLQ09015VOS06LIF4Q525NRO1I
16 State = opened
17 ?
18 var accountNumber (type: IBAN) = MT58PDLQ09015VOS06LIF4Q525NRO1I
19 var balance (type: Money) = - EUR3804.50
20
21 Endpoint: /Account/MT58PDLQ09015VOS06LIF4Q525NRO1I/Interest
22 JSON payload: { "Interest": { "currentInterest": "-77.09" } }
23 Response: ("body": "", "isSuccessful": false, "message": "Bad Request",
24 "errorBody": "com.fasterxml.jackson.databind.JsonMappingException: Can not construct instance of
25 squants.Dimensionless: no String-argument constructor&#x2F;factory method to deserialize from String
26 value (&#x27;-77.09&#x27;)\\n at [Source: io.undertow.servlet.spec.ServletInputStreamImpl@578015db;
27 line: 1, column: 35] (through reference chain: nl.ing.corebank.dto.
28 account.Interest[&quot;currentInterest&quot;])", "code": "400")

```

Listing 5.33: Failing test on *interest* transition with the use of Scala-ES generator (part 2)

```

1 {
2   "id": "nl.ing.corebank.aggregates.AccountAggregate$|077708cd-769a-48ec-8006-d607241c4f45",
3   "_version": 1,
4   "_state": "OpenedState",
5   "accountNumber": {
6     "iban": "MT58PDLQ09015VOS06LIF4Q525NRO1I"
7   },
8   "balance": "50.00 EUR"
9 }

```

Listing 5.34: Account state in the SUT after performing the *interest* transition

Likewise, the Javadatomic generator, we have discovered a bug in the SUT, since the simulated transition is not conform to the specification. To know where this bug occurs and which code is not properly generated, the response from the *interest* transition request gives an error message. According to the error message, the SUT is not able to construct the instance of *Dimensionless* for the transition parameter *currentInterest*. This error seems to occur in the class *Interest*, which is shown in Listing 5.35. The parameter *currentInterest* indeed uses the type *Dimensionless* here.

```

1 @JsonRootName(value = "Interest")
2 @JsonCreator
3 case class Interest (@JsonProperty("currentInterest") currentInterest: Dimensionless)

```

Listing 5.35: Code in Scala

The class from [Listing 5.35](#) is used in [Listing 5.36](#). The *interest* method handles the *interest* transition and as you can see is the class *Interest* used as the second parameter. This class is used to bind the interest parameter as a type of the *Interest* class.

In short, we have discovered a bug in the SUT for the *interest* transition. The *interest* transition bug in the Javadatomic generator belongs to the fixed code category, but in this case, for the Scala-ES generator, the bug belongs to the injected code category, since the *Interest* class is generated and injected in the generated code.

```
1 @POST
2 @Consumes(Array[String](MediaType.APPLICATION_JSON))
3 @Produces(Array[String](MediaType.APPLICATION_JSON))
4 @Path("/{accountNumber}/Interest")
5 def interest (@Suspended asyncResponse: AsyncResponse,
6   @PathParam("accountNumber") accountNumber: IBAN, interest: Interest): Unit = {
```

Listing 5.36: Code in Scala

5.5.4 Distributed Codegen-Akka

In this test run is again the Codegen-Akka generator used, but the SUT runs distributed. By running the SUT as multiple nodes, two Scala system nodes and one Cassandra node, we can focus more on testing the unanimous final outcome of the SUTs. Therefore, the pre-transition check and transition check are performed in the first node (Node 1). The post-transition check is performed in the second node (Node 2). In previous experiments, the three steps are performed to execute and test transitions in a single node. By performing the post-transition check on Node 2, it can be checked whether the SUT produces a unanimous final outcome.

To investigate the failing tests, we discuss the test result for the transition *interest*. The results of the test run are shown in [Listing 5.37](#). The test for the current state as well as the transition fails for the transition *interest*. As shown on line number 27 the request for constructing the current state is successful. As part of the post-transition check, checking whether the new state from the SUT is conform to the new state from the trace fails (see line number 29-30). This also applies to the *interest* transition; the transition succeeds but the post-transition check fails.

```

1 Test transition interest
2 opened -> interest -> opened
3
4 0:
5   now = 18 Sep 2017, 08:57
6
7   instance: simple_transaction.Account, key = FO1227539908389742
8   ?
9   ?
10  var accountNumber (type: IBAN) (uninitialized)
11  var balance (type: Money) (uninitialized)
12 1:
13  now = 18 Sep 2017, 08:57
14  step: simple_transaction.Account.openAccount
15  var initialDeposit (type: Money) = EUR50.00
16  Transition to state = opened
17  Identified by accountNumber = FO1227539908389742
18
19  instance: simple_transaction.Account, key = FO1227539908389742
20  State = opened
21  ?
22  var accountNumber (type: IBAN) = FO1227539908389742
23  var balance (type: Money) = EUR50.00
24
25  Endpoint: /Account/FO1227539908389742/OpenAccount
26  JSON payload: { "OpenAccount": { "initialDeposit":"EUR 50.00" } }
27  Response: ("body":"CommandSuccess(OpenAccount(50.00 EUR))","isSuccessful":"true","message":"OK",
28  "errorBody":"","code":"200")
29  Could not find state opened, expected "state":{"Opened":{}}
30  Could not find value balance, expected "balance":"EUR 50.00"
31
32 1:
33  now = 12 Jul 2016, 12:00:00
34
35  instance: simple_transaction.Account, key = FO1227539908389742
36  State = opened
37  ?
38  var accountNumber (type: IBAN) = FO1227539908389742
39  var balance (type: Money) = EUR50.00
40 2:
41  now = 12 Jul 2016, 12:00:00
42  step: simple_transaction.Account.interest
43  var currentInterest (type: Percentage) = (- 7709)
44  Transition to state = opened
45  Identified by accountNumber = FO1227539908389742
46
47  instance: simple_transaction.Account, key = FO1227539908389742
48  State = opened
49  ?
50  var accountNumber (type: IBAN) = FO1227539908389742
51  var balance (type: Money) = - EUR3804.50
52
53  Endpoint: /Account/FO1227539908389742/Interest
54  JSON payload: { "Interest": { "currentInterest":"-77.09" } }
55  Response: ("body":"CommandSuccess(Interest(-77.09))","isSuccessful":"true","message":"OK",
56  "errorBody":"","code":"200")
57  Could not find state opened, expected "state":{"Opened":{}}
58  Could not find value balance, expected "balance":"EUR -3804.50"

```

Listing 5.37

As said, the post-transition check is performed on Node 2, and pre-transition check and transition check is performed on Node 1. In post-transition check, the instances from the SUT are retrieved, and its state and values are compared. According to the results of [Listing 5.37](#), the post-transition check fails and could not compare the state and values. In previous experiments, the post-transition check is performed against the same node as the pre-transition check and transition check. Since the post-transition check in these experiments is performed on the same node, as part of the post-transition check, the instance is retrieved from Node 1. The results of the retrieval from both Node 1 and Node 2 are shown in [Listing 5.38](#). As shown in these results, the retrieval fails on Node 2 but on Node 1 this is successful. It is remarkable since the transition is also performed on Node 1.

```

1 Node 2
2 Endpoint: /Account/FO1227539908389742
3 Response:
4 ("body":"","isSuccessful": false,"message":"Service Unavailable","errorBody":"The server was not able to
5   produce a timely response to your request.\r\nPlease try again in a short while!","code":"503")
6
7 Node 1
8 Endpoint: /Account/FO1227539908389742
9 Response:
10 {
11   "state":{
12     "SpecificationState":{
13       "state":{
14         "Opened":{
15
16         }
17       }
18     },
19   },
20   "data":{
21     "Initialised ":{
22       "data":{
23         "accountNumber":null,
24         "balance":"EUR -3804.50"
25       }
26     }
27   }
28 }
```

Listing 5.38: Account state in the SUT after performing the *interest* transition

Now we have only discussed the test results of the *interest* transition. According to the results of [Subsection 5.4.4](#), besides the *interest* transition, all other transitions fail. The reason for the failing tests for the transitions is the same as for transition *interest*. Except for the transition *close*, as mentioned earlier, the simulation is not able to the transition. To conclude, in this experiment the SUTs does not produce a unanimous final outcome.

5.6 Evaluation

5.6.1 Bugs

The expectation for the criteria bugs from [Subsection 5.2.1](#) is to find bugs in the SUT where execution traces from the SMT solver is not accepted by the SUT. For example, a generated pre- or postconditions are not satisfied by the execution from the traces.

In this experiment, we have found bugs in the code generators. For both bugs, [Subsection 5.5.2](#) and [Subsection 5.5.3](#), it was not possible to perform the execution from the traces returned by the SMT

solver. The preconditions are not satisfied for both bugs. To conclude, we did find bugs as expected where the SUT was not conform to the specification.

This experiment is using more complex specifications which implement synchronisation. As discussed, it should be possible to test synchronisation in the SUT. According to the results from [Section 5.4](#), the *book* transition which implements synchronisation has been successfully tested on all generators, except the distributed Codegen-Akka. As expected, the synchronisation is tested in the SUT, but no faults were found. There does not seem to be any faults in partial failure or asynchrony.

Not being able to find bugs does not mean that there are no faults in the synchronisation. This can be due to the way of testing or the setup how the SUT runs. With the Codegen-Akka generator, the SUT is run as single nodes, both the Scala system and Cassandra. The ACP is still applied with single nodes. By running multiple nodes, it is possible to focus more on the testing of the unanimous final outcome of the SUTs. Therefore, the test run is run to test with multiple nodes. Faults were found in different results between the nodes. The expectation is to find bugs in synchronisation where the SUT does not produce a unanimous final outcome. In the case of the expectation of the SUT producing a not unanimous final outcome, faults were found. Partial failure may also occur, for example when external systems are communicating with the distributed Codegen-Akka, but this is not the case.

The *book* transition in this experiment is the only transition which implements synchronisation. With the use of more complex specifications, in the sense of synchronised transitions, it is possible to test better synchronisation. Parallel execution of transitions can also give valuable results, as partial failure or asynchrony is more tested.

5.6.2 Efficiency

By using the traces, we can check whether the given execution is possible in the SUT. Also, it solves the limitation of the experiment from [Chapter 5](#). Therefore, the expectation is to perform the same computed path by the SMT solver should be performed in the SUT.

Checking and simulation are used to generate tests for transitions in this experiment. The traces given by the simulation and checking are used to check whether the SUT behaves as the specification and whether it accepts the execution from the trace. In this experiment, the transitions are in the SUT performed in three steps to identify misbehaviour in the SUT. To conclude, as expected the same transitions from the traces are performed in the SUT.

As expected, it takes longer to test all transitions from specifications since each transition is tested. Some transitions require an initial state for which transitions need to be performed to reach this state. As a consequence, transitions are executed and tested more often. For instance, the initial state *blocked* needs to be reached to test the transition *unblock*, but to reach the state *blocked* the transition *block* needs to be executed and tested. In addition, the test framework also tests the transition *unblock* separately. Thus transitions which are executed and tested more often is not efficient, and it may take longer to test all transitions.

What remarkable is that in this experiment the same trace from the SMT solver is used to test a transition in two different test runs, namely [Subsection 5.5.2](#) and [Subsection 5.5.3](#). By performing the test runs more often, it can be said that the SMT solver regularly generates the same traces for transitions. In our case, this is not bad, as long as every transition is tested despite the randomness of the traces. However, testing unique traces may result in finding more bugs in the SUT. A solution is to configure the SMT solver to use random seeds to control the propositional variable selection heuristic in the SMT core.

5.6.3 Coverage

The coverage criteria expect to test all the transitions from a specification since the simulation can test single steps. This also holds for the complex specifications.

With this experiment, it is possible to test all the transitions, except the *close* transition. Even it is possible to test transitions from the complex specifications. The reason for not able to test the *close* transition is that the precondition of this transition is not satisfied due to the current state defined for this transition. Altogether, with this experiment, it is possible to test all the transitions when the preconditions of a given transition are satisfied by the current state.

Another expectation is that faults like partial failure and asynchrony cause the inability to test transitions, which leads to a lower coverage. As discussed in the criteria bugs, no faults were found in partial failure or asynchrony. This leads not to the inability to test transitions. Again, not being able to find bugs does not mean that there are no faults. The reason for this can be the way of testing or the setup how the SUT runs.

In this experiment is a fixed configurable time-out used. Just as it is not possible to test the *close* transition, it is also possible not to test transitions due to the chosen configurable time-out. The reason for this is, for example, a high configurable time-out that makes it impossible to test the chosen transition, because of the precondition of the chosen transition is not satisfied by the current state, like the *close* transition; or a low configurable time-out that makes it impossible to reach a state to test the chosen transition.

5.7 Conclusion

In this experiment are more complex specifications used for testing. Checking and simulation are used to generate tests for transitions in this experiment. The traces given by the simulation and checking are used to check whether the SUT behaves as the specification and whether it accepts the execution from the trace. Unlike the previous experiment from [Chapter 4](#), it tests what should be possible according to the specification by using the traces from the SMT solver. Even are the transition parameters data values generated by the SMT solver. In this experiment, the transitions are performed in the SUT in three steps to identify misbehaviour in the SUT. Although there is a limitation in the testing of states. Within a trace, not every instance may contain the state of it.

Also with this experiment, we have found bugs in the code generators. The bug in the *interest* transition from [Subsection 5.5.2](#) is found in the fixed code. Another bug for the same *interest* transition from [Subsection 5.5.3](#) belongs to the category injected code since the bug is found in the generated code from the specification. As discussed in [Section 3.6](#), it is possible that there might be bugs in templating, in the fixed code or the injected code. With these test runs of this experiment, we have found bugs for both categories.

This experiment is using more complex specifications which implement synchronisation. The test framework has successfully tested all generators which implements the synchronisation. No faults were found, except in the distributed Codegen-Akka. In the distributed Codegen-Akka faults were found in not unanimous final outcome between the nodes. So with this test approach faults can be found in distributed systems.

As discussed before, with this experiment it was not possible to test the *close* transition. The precondition of this transition is not satisfied due to the current state defined for this transition. To sum up, with this experiment it is possible to test all the transitions when the preconditions of a given transition are satisfied by the current state. The theory for this experiment is as follows: $\forall e s_1 \rightarrow s_2, (ls_2pre(e) \vee s_2pre(e)ins_1post(e))$.

5.8 Threats to validity

Limited specifications

In the conducted experiments are the specifications account and transaction used to test the SUT from these specifications. With these experiments and specifications, we did find faults in the code generators. Although these specifications are quite simple, *e.g.*, the specifications within a bank would be more complex. These experiments take into account the generosity of specifications, but with such a large amount of complex specifications, it is questionable whether these experiments still produce valuable results.

Rebel interpretation in SMT solver

The SMT solver can be seen as an interpreter for *Rebel* specifications. The conducted experiments use the SMT solver to test the SUTs from the specifications. We already discussed before the limitation of

interpretation of *Rebel* specifications, and in some cases, workarounds have been used. There may be more unknown limitations of the interpretation of *Rebel* specifications, which can cause the conducted experiments give incorrect results.

Valid execution trace

The experiment from [Chapter 5](#) tests only valid execution traces from the SMT solver, *i.e.*, testing only what should be possible according to the specification. Testing valid execution traces is not enough, testing invalid execution traces can be valuable. The experiment from [Chapter 4](#) already does this, but as discussed it has a few limitations.

Chapter 6

Discussion

In this chapter, we discuss further the results from the experiments and answers will be given to the sub research questions.

6.1 RQ 1: How is the input/output of the generated system tested?

6.1.1 Experiment 1: Invalid execution traces

In [Chapter 4](#), we have seen an experiment where it is possible to automatically generate a test for every transition from a given specification. This experiment tests the opposite of a specification, i.e. testing what should be not possible according to the specification. Testing is done by using checking for a given transition and testing this in the SUT.

Testing the opposite is also often used in mutation testing. Mutation testing generates faulty programs (mutants) by syntactic changes, in our case, we create only one faulty version of the program. The faulty program is generated based on the mutation operator Negate Conditionals Mutator. The mutant in this testing approach is killed when the result from the SMT solver and the SUT are the same.

In [Section 4.5](#), we discussed the evaluation of this experiment. In short, this experiment produces some false positives/negatives. The reasons for this are varying results from the SMT solver and the comparability of performed transitions between the SMT solver and the SUT. Also, the SMT solver is smarter and better in checking the satisfiability. With this experiment, we did find two bugs in the SUT; only one bug is within our scope.

In this experiment are traces not used since with checking traces are not available when a state is not reachable. The assumption was that these states were not reachable due to the opposite preconditions. As discussed in the experiment, in some transitions the state to reach are reachable, *e.g.*, when no preconditions are supplied or when a precondition is not applied on a property of a specification (which is not part of a transition). In the experiment is mutation testing applied on the executed transition in the SUT. Instead of mutation testing the executed transition in the SUT, mutation testing can also be applied to the *Rebel* specifications. The specifications are then mutated, then tests can be generated that distinguish between the original model and the mutated specification. [8, p. 8] Traces are also available with this approach since only the specification is mutated and it can be interpreted by the SMT solver. These traces can be used to test the SUT, which allows this approach to be combined with the second experiment. As a result, the limitation of this experiment is solved with the use of SMT solver; it is not necessary anymore to check the satisfiability in the SUT. The study [32] reports that model-based testing technique using mutation has valuable fault detection effectiveness.

6.1.2 Experiment 2: Valid execution traces

The experiment of [Chapter 5](#) has solved a few limitation of the previous experiment. This experiment extends the model testing approach with *Rebel*.

In this model testing approach are the traces used to check whether the SUT accept the execution from the trace and whether it behaves as the specification. Therefore are the traces used in this approach.

In comparison to the previous experiment, this approach tests what should be possible according to the specification. Even the transition parameters data values are generated by the SMT solver which satisfies the constraints of the transition. This experiment uses two existing testing techniques within *Rebel* to generate tests for transitions, namely checking and simulation.

To perform and test transitions from traces are three steps done, pre-transition check, transition check and post-transition check. Also, with this testing approach, we did found bugs in the code generators. The intention is to test all transitions, but with this testing approach, it is only possible to test transitions when the preconditions of a given transition are satisfied by the current state. The theory for this testing approach is as follows: $\forall es_1 \rightarrow s_2, (!s_2pre(e) \vee s_2pre(e)ins_1post(e))$.

6.2 RQ 2: Are there any false positives/negatives when the generated system has been implemented correctly?

6.2.1 Varying results from the SMT solver

In [Section 4.5](#), we already discussed the limitation of this testing approach. The test run of this experiment produces some false positives/negatives. As mentioned earlier, this is due to the varying results from the SMT solver and the comparability of performed transitions between the SMT solver and the SUT.

6.2.2 Invalid current state

In [Section 5.5.1](#), we have seen that the experiment is not able to test the *close* transition. In short, it is not able to test this transition because the current state and its values were not satisfying for the transition to the next state. In this case, the simulation is not able to perform the transition, although this transition can be made in the SUT with satisfying parameters.

The current state for the transitions is generated by the test framework. It is possible to generate current states based on the conditions of the chosen transition, but this can become complex when multiple complex specifications are used. Again, this is playing the SMT solver; the SMT solver is better/smarter in doing this kind of computations. So it would be better to extend the model checker and define conditions of transitions. This is left as future work.

To conclude, this experiment is only able to test transitions when the preconditions of a given transition are satisfied by the current state.

6.2.3 Identifiers for entities

We discussed in [Section 5.3.1](#) that the identifiers for generating the current state are generated by the test framework. This is done for the following reasons, the uniqueness of the identifiers are only within a trace and the limitation of the SMT formulas of *Rebel* types.

In the case of the type *IBAN*, the given identifier by the SMT solver is auto-incremented, only unique in one trace and not compliant with the ISO_13616 and ISO_9362 standards. Thus the *Rebel* types are interpreted by the SMT solver are not conform to the *Rebel* types. This can cause problems when these values from the type are read from a given trace and tested against the SUT. Therefore, is a random generator implemented, for only *IBAN* and *Integer*, which generates appropriate values which can be used to test against the SUT.

On the other hand, due to the misinterpreted *Rebel* types, it is possible that SMT solver is not able to solve a given specification.

what are the definitions of Rebel types?

6.3 RQ 3: What kind of bugs can be found and what are the factors?

From the conducted experiments, five bugs have been found in the generated system that is generated by the code generators. We can categorise these bugs in the following categories: templating, compilation and distribution.

6.3.1 Templating

The code generators use templating to generate code from *Rebel* specifications. The generated code should correctly map to the input code from templates. Otherwise, the generated code and *Rebel* specifications will have different meanings.

Most of the bugs we did find with the experiments are within the category templating. We split this category into two parts, fixed code and injected code. Fixed code is definitive code that is part of generated systems and will not be changed regardless of the *Rebel* specifications. Injected code is code which is generated from *Rebel* specifications that fills some isolated parts of the fixed code that are dependent on the input of the specifications.

With the experiment from [Chapter 5](#), we did find two templating bugs in the code generators. One bug is from the Javadatomic generator, the code which causes the bug is not generated from the specification. This code is part of the fixed code where the code from the specification is generated. So, the identified bug in the SUT belongs to the category templating within the fixed code. The bug within the Scala-ES generator belongs to injected code. The identified bug is caused by the generation of code from a given specification.

To conclude, within the category templating, it is possible to have bugs either in fixed code or injected code.

6.3.2 Compilation

In the category compilation belongs bugs which are not compilable systems generated by the code generators from a given specification. For this category, we did find a bug which is discussed in [Section 4.4.1](#). Just as mentioned there, this bug is out of scope, since the intention is to find misbehaviour in SUT with our test framework. Therefore, the generated system from the code generators needs to compile.

6.3.3 Distribution

Distribution is another category for bugs, although we did not find any bugs related to distribution with our experiments when the SUT is run as single nodes. In the experiment from [Chapter 5](#) faults were found in distribution when the Codegen-Akka system is run in a distributed mode. In the distributed Codegen-Akka faults were found in not unanimous final outcome between the nodes.

The experiment from [Chapter 5](#) tests also transitions which contain synchronisation. No bugs are found in synchronisation, it can be assumed that the synchronised transitions works, in the sense of the result of the tested transitions were the same as the trace from the SMT solver. As discussed before, the synchronised transitions are also translated to logical formulas. These formulas contain no logic about synchronisation, and therefore it should be possible to identify misbehaviour in synchronisation. Several studies also claim to find faults in distributed algorithms.

Chapter 7

Conclusion

In this work, we have shown two proof of concepts to test generated systems from *Rebel* specifications. With these proof of concepts, it can be tested whether the generated systems are generated properly based on *Rebel* specifications. The result of this is that we regained the benefits from *Rebel* domain, and again able to test and reason about *Rebel* specifications and generated system.

How to validate the generated code from a Rebel specification?

In an earlier study [3, p.3], the author proposed a possible solution, which is to use the SMT solver to test generated systems. In both proofs of concepts, the SMT solver holds the key in testing the generated systems.

The generated systems are tested in two ways, invalid execution and valid execution. The first experiment tests invalid execution in the generated systems, *i.e.*, testing what should be not possible according to the specification. Therefore, the test framework uses checking to check the satisfiability for a transition from the specification, and then test this in the SUT. In this experiment is a mutation operator, which is used in mutation testing, applied to test invalid execution. With this experiment, we did find two bugs in the SUT; only one bug is within the research scope.

The second experiment tests valid execution in the generated systems, *i.e.*, testing what should be possible according to the specification. This experiment uses two existing testing techniques within *Rebel* to generate tests for transitions, namely checking and simulation. The traces from this are used to test SUT in this experiment. In this experiment is a model-based testing approach taken to check whether the SUT accept the execution from traces whether the SUT behaves as the specification. Even the transition parameters data values are generated by the SMT solver which satisfies the constraints of the transition. With this experiment, we did find three bugs in the SUT; one bug is identified by slightly different test approach to find distribution faults.

To sum up, with the experiments a total of five bugs have been found in the generated system that is generated by the code generators. These bugs can be categorised in the following categories: templating, compilation and distribution.

7.1 Future work

7.1.1 mutation model-based testing

Acknowledgements

Some acknowledgements.

Bibliography

- [1] J. Stoel, T. V. D. Storm, J. Vinju, and J. Bosman, “Solving the bank with rebel: on the design of the rebel specification language and its application inside a bank,” *Proceedings of the 1st Industry Track on Software Language Engineering - ITSLE 2016*, 2016.
- [2] T. B. T. M. T. Glenford J. Myers, Corey Sandler, *The art of software testing*. John Wiley & Sons, 2nd ed ed., 2004.
- [3] J. Stoel, “A case for rebel, a dsl for product specifications,” in *Proceedings of Domain-specific Language Design and Implementation 2015 (DSLDI 2015)*, pp. 9–11, arXiv, 2015.
- [4] A. S. Tanenbaum and M. Van Steen, *Distributed systems: principles and paradigms*. Createspace Independent Publishing Platform, 3 ed., 2017.
- [5] K. Sen, “Concolic testing,” in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pp. 571–572, ACM, 2007.
- [6] P. Godefroid, M. Y. Levin, D. A. Molnar, *et al.*, “Automated whitebox fuzz testing,” in *NDSS*, vol. 8, pp. 151–166, 2008.
- [7] Y. Jia and M. Harman, “An analysis and survey of the development of mutation testing,” *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2011.
- [8] M. Utting, A. Pretschner, and B. Legeard, “A taxonomy of model-based testing approaches,” *Software Testing, Verification and Reliability*, vol. 22, no. 5, pp. 297–312, 2012.
- [9] J. Tretmans, “Model based testing with labelled transition systems,” *Formal methods and testing*, pp. 1–38, 2008.
- [10] S. R. Dalal, A. Jain, N. Karunanithi, J. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, “Model-based testing in practice,” in *Proceedings of the 21st international conference on Software engineering*, pp. 285–294, ACM, 1999.
- [11] M. Leucker and C. Schallhart, “A brief account of runtime verification,” *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, 2009. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS07).
- [12] K. Havelund and A. Goldberg, “Verify your runs,” *Verified Software: Theories, Tools, Experiments*, pp. 374–383, 2008.
- [13] Y. Falcone, J.-C. Fernandez, and L. Mounier, “Runtime verification of safety-progress properties,” *RV*, vol. 5779, pp. 40–59, 2009.
- [14] P. Klint, T. van der Storm, and J. J. Vinju, “Easy meta-programming with rascal. leveraging the extract-analyze-synthesize paradigm for meta-programming,” in *Proceedings of the 3rd International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE’09)*, LNCS, Springer, 2010.
- [15] J. Herrington, *Code Generation in Action*. Manning, revised ed., 2003.

- [16] R. Roestenburg, R. Bakker, and R. Williams, *Akka in action*. Manning Publications Co., 2016.
- [17] A. Lakshman and P. Malik, “Cassandra: a decentralized structured storage system,” *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.
- [18] J. Anderson, M. Gaare, N. Bailey, T. Pratley, *et al.*, *Professional Clojure*. John Wiley & Sons, 2016.
- [19] M. Fowler, “Cqrs,” *Command Query Responsibility Segregation*), <http://martinfowler>, 2011.
- [20] M. Fowler, “Event sourcing,” *Online*, Dec, p. 18, 2005.
- [21] L. D. Moura and N. Björner, “Z3: An efficient smt solver,” *Tools and Algorithms for the Construction and Analysis of Systems Lecture Notes in Computer Science*, p. 337340, 2008.
- [22] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. Kats, E. Visser, and G. Wachsmuth, *DSL engineering: Designing, implementing and using domain-specific languages*. dslbook. org, 2013.
- [23] A. Mkrtchyan, “Pit mutation operators.” <http://pitest.org/quickstart/mutators/>.
- [24] K. N. King and A. J. Offutt, “A fortran language system for mutation-based software testing,” *Software: Practice and Experience*, vol. 21, no. 7, pp. 685–718, 1991.
- [25] I. Konnov, H. Veith, and J. Widder, “What you always wanted to know about model checking of fault-tolerant distributed algorithms,” in *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pp. 6–21, Springer, 2015.
- [26] F. Alberti, S. Ghilardi, A. Orsini, and E. Pagani, “Smt-based approaches to model checking of distributed broadcast algorithms: some case studies,” 2015.
- [27] C. McCaffrey, “The verification of a distributed system,” *Communications of the ACM*, vol. 59, no. 2, pp. 52–55, 2016.
- [28] Y. J. Al-Houmaily and G. Samaras, “Two-phase commit,” in *Encyclopedia of Database Systems*, pp. 3204–3209, Springer, 2009.
- [29] A. Mkrtchyan, “iban4j.” <https://github.com/arturmkrtchyan/iban4j/tree/ce1ca12b947755d1d96aab294d1ea15c78090ca2>, 2015.
- [30] M. Utting and B. Legeard, *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2010.
- [31] G. Tretmans and H. Brinksma, *TorX: Automated Model-Based Testing*, pp. 31–43. 12 2003.
- [32] A. Paradkar, “Case studies on fault detection effectiveness of model based test generation techniques,” in *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 1–7, ACM, 2005.