Runtime testing generated systems from Rebel specifications

Thanusijan Tharumarajah
tthanusijan@gmail.com

October 11, 2017, 61 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

Growing systems is a concern for large organisations. 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 to generate running systems from this domain knowledge. *Rebel* is a formal specification language in which financial products can be specified.

Software testing is an important part of software projects. To facilitate the process *Rebel* offers automated simulation and checking of specifications with the use of a SMT solver. Simulation and checking make use of bounded model checking. 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. It is not always straightforward to generate a correct system from *Rebel*. 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. *Rebel* is declarative while an implementation is not. The running systems which are generated from specifications need to be properly based on these specifications; it should conform to these specifications.

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. In both proofs of concepts, the satisfiability modulo theories solver holds the key in testing the generated systems. 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. 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. The second experiment tests valid execution in the generated systems, *i.e.*, testing what should be possible according to the specification.

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

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.

The full source code of the test framework and the experiments are made available on *GitHub*[1]. This repository also contains the full LaTeXsource for this thesis. Note that the test framework communicates with generated systems generated by the code generators from *ING*, which are closed-source.

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]https://github.com/thanus/master-thesis

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.

Expectations

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. The implementation correctness dimensions can also be found in the experiments. These are therefore discussed in detail in these chapters.

These are dimensions that can cause faults in the implementation. A fault can be introduced by templating or compilation or distribution errors. The expectation is to find the first faults in the first two dimensions, templating or compilation since faults in distribution are more difficult to find. The three implementation correctness dimensions are discussed below:

- **Templating** 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.
- **Compilation** The generated code from the code generators needs to compile. Otherwise, it is not possible to run or test the generated system.
- **Distribution** 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. As part of postconditions, 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).

Assumptions

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. It is not effective to test incorrect specifications. Testing inconsistent behaviour is senseless and therefore wasted time. It is also much more difficult because there is nothing to tell about the expectations.
- **The generated system can be compiled.** Note that we are testing the generated code, not the generator. Showing that a generator always generates compilable code is a different interesting questions which is out of the scope of this thesis.

- **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** Which false positives occur when the generated system is correctly implemented?
>
> **SQ3** What kind of faults 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 will be developed; 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 fault and test it with the SMT solver. A fault is seen as the deviation between the current behaviour and the expected behaviour. [5, 6] Typically this is identified by the deviation between the current and expected state of the system. In our case, the expected state and behaviour is defined in the *Rebel* specifications.

For the lightweight version, it is an easily reproducible fault. 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 [7], Fuzz testing [8] and Mutation testing [9].

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 faults and factors encountered in the generated system that was found using the methodologies.

1.3 Related Work

Testing generated systems can be performed on different aspects. This section briefly introduces relevant work of this thesis.

Model-based testing

Model-based testing entails the process and techniques for automatic generation of test cases using abstract models. [10, 11, 12] 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. [10, 11]

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. [10] 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. [10, 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. [10, 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. [6, 13, 14] Runtime verification is seen as a lightweight verification in addition to verification techniques like model checking and testing. [6, 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. [6, 14]

A so-called monitor in runtime verification performs the checking whether an execution in the system meets the safety property. [6, 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. [6, 14]

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. [6, 14] 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] Some property of interest which is used in bounded model checking to check whether it holds is called a safety property. 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. [6, 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.

Property-based testing

Property-based testing is a software testing approach where the generic structure of valid inputs of the program needs to be defined combined with properties which are expected to hold for every valid input. [15, p. 3] The properties relate to the behaviour of the program and the input-output relationship. Using these data, a property-based testing tool can automatically generate randomly valid input. This input is then applied to the program while monitoring the execution to test whether it behaves as expected. A well-known property-based testing tool is QuickCheck [16] for Haskell.

Property is a partial high-level specification of the SUT. In comparison to full specification, properties are compact and easy to write and understand. [15, p. 3] For example, a property could be for a given method which takes a list as an argument, the returned list from this method must have the same size as the passed list. This property must hold despite the passed list. Like this, properties can be specified for *Rebel* or a *Rebel* specification.

At the same time of this research, another master's student has also researched the testing of generated systems from *Rebel* specifications. This approach uses property-based testing. [17] There are three main differences between this and our approach.

Firstly, the property-based testing approach uses one *Rebel* specification to test the generated system. The defined properties in this *Rebel* specification should also hold in the generated environment. These properties should hold in the generated environment despite the defined *Rebel* specifications because these properties are bound to the *Rebel* domain semantics. In our approach, we can use any *Rebel* specification to test the generated system whether it behaves according to the specification. Our approach is less tied to the *Rebel* semantics but places more emphasis on the defined specification.

Secondly, the property-based testing approach uses offline testing, and our approach uses online testing. The property-based testing approach generated unit tests based on the defined properties. With offline testing [10] test cases are generated strictly before they are run. This is also the case with property-based testing approach since we know in advance which test cases are generated. In our testing approach, we got two online systems, namely the SMT solver and the generated system. The SMT solver has in our approach an important part since it is used to generate test cases. We do not know in advance which test cases are being generated by the SMT solver.

Thirdly, test execution is in our approach done at runtime. As mentioned earlier, the property-based testing approach generated unit tests. These unit tests are run in the test mode of the generated system.

Testing distributed systems

Distributed systems are difficult to build, because of partial failure and asynchrony. [18, p. 1] In order to create a correct system, these two concepts of distributed systems need to be addressed. Solving these problems often result in complex solutions. The study [18] describes verifications of distributed systems in two categories, namely formal verification and verification in the wild.

Formal verification is a systematic process which uses mathematical reasoning to proves properties about a system. [18, 19] This results in a system that can be said to be provably correct. Formal specification languages allow to model and verify the correctness of concurrent systems. [18, p. 2] An example of a formal specification language is TLA+ which is used with by Amazon Web Services (AWS) to verify critical systems. [20, p. 1] AWS has applied various techniques (fault-injection testing, stress testing, etc.). However, they still found subtle bugs hidden in complex fault-tolerant systems. The use of TLA+ has yielded valuable results for AWS in two ways. Finding bugs that could not be found in other ways, and making performance optimisations without the loss of correctness. [20, p. 3]

Model checking, which is also a formal method, determines whether a system is provably correct. Model checkers systematically use state-space exploration to provide the paths for a system. [18, p. 3] A system is provably correct when all path have been executed. AWS has used TLA+ specifications in combination with a model checker. This resulted in the identification of a bug that could cause data loss in DynamoDB, which is a data store. [20, p. 7] The shortest trace for this bug contained 35 steps.

Compared to formal verification, given that formal verification is expensive, test methods can be used that give confidence that the systems are built correctly. [18, p. 4] Simple test methods such as unit and integration tests or property-based testing can already be a way to test distributed systems. Fault-injection testing is a test method for causing or introducing an error in the system. By forcing the occurrence of failures, it allows the observation and measurement of the systems by the engineers.

The main similarity with our approach and the approach described above is the use of formal verification. *Rebel* is also formal specification language. [1, p. 1] Model checking is also available with

Rebel specifications, although this is bounded. Other test methods as described above can be used to test distributed systems, but this is out of the scope of this thesis.

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 faults. Finally, a conclusion of this thesis is given in Chapter 7.

Chapter 2

Background

2.1 Rebel

Rebel is a formal specification language written in the language workbench Rascal [21]. The specification language is developed by ING[1] 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. For example, *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*.

```
specification Account {
  fields {
    accountNumber: IBAN @key
    balance: Money
  }

  events {
    openAccount[]
  }

  lifeCycle {
    initial init -> opened: openAccount
    final opened
  }
}
```

Listing 2.1: A simple account specification

[1] https://www.ing.nl/
[2] https://www.cwi.nl/

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 transitions (events). These definitions are specified somewhere else to promote reuse of transitions 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. The keyword new is used in the postcondition to refer to the value of the variable in the post-state after the execution of the transition. [1, p. 4]

```
event openAccount[minimalDeposit: Money = EUR 0.00](initialDeposit: Money) {
  preconditions {
    initialDeposit >= minimalDeposit;
  }
  postconditions {
    new this.balance == initialDeposit;
  }
}
```

Listing 2.2: *openAccount* transition definition from specification

2.1.2 Code generation

Writing programs that write programs is called code generation. [22, p. 3] 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 Rascal (which has a page-template feature) [21] to build code. 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 these errors can be fixed through revising the templates and starting the code generation process. [22, p. 15-17] These fixes are applied consistently throughout the code base.

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 [23, p. 4] as Actor Model and Cassandra [24] is used for storage.
- Codegen-Javadatomic: This generator generates a Java system based on the *Rebel* specifications. The generated system uses Datomic [25, 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) [26] and Event Sourcing [27].

The API's of the generated system from the code generators are not completely 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 transition parameters which causes the current state. The labelled transitions map to the transitions and their preconditions and postconditions. *Rebel* has also support to specify invariants for a given specification. These are predicates which should always be true during the lifecycle of an instance of the specification.

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 [28] 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.

The bounded analysis tries to find the smallest (the least possible steps within bounds) 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 timeout (bound) is met. A configurable timeout is used to control the maximum spent waiting time of the user. [1, p. 5]

An example of checking *Rebel* specifications is given in Listing 2.3. These checks can be defined in so-called tebl files. As configurable time-out is six used. In this case, the SMT solver tries to find the smallest possible counterexample with an opened account with the balance above 0 euro. The SMT solver checks incremental whether the state can be reached in steps until a counterexample is found or the configuration timeout (bound) is reached.

```
module simple_transaction.OpenAccountCheck

import simple_transaction.Account

state openAccountCheck {
  opened Account with balance > EUR 0.00;
}

check openAccountCheck reachable in max 6 steps;
```

Listing 2.3: Checking opened account

2.2.2 Simulation

The purpose of simulation and checking differs. As explained in the previous paragraph, checking is used to reason about possible counterexamples. 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 fault 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* transition can be used to put the account to the state *blocked*. The final state of account is *closed*. To execute the transition *close*, 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 validate every account with a positive balance.

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

```
specification  Account {
  fields  {
    accountNumber: IBAN @key
    balance:  Money
  }

  events  {
    openAccount[minimalDeposit = EUR 50.00]
    withdraw[]
    deposit []
     interest  []
    block []
    unblock[]
     close []
  }

  invariants  {
    positiveBalance
  }

  lifeCycle  {
     initial   init   −> opened: openAccount

    opened −> opened: withdraw, deposit, interest
            −> blocked: block
            −> closed: close

    blocked −> opened: unblock

     final  closed
  }
}
```

Listing 3.4: Account specification

The account specification has a *close* transition[2] to close an account which is illustrated in Listing 3.5. The precondition of the *close* transition 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.

```
event  close ()  {
  preconditions  {
    this . balance == EUR 0.00;
  }
}
```

Listing 3.5: *close* transition definition from account specification

3.2 Method

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

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

version which can trigger a fault and test it with the SMT solver. For the lightweight version, it is an easily reproducible fault. This fault is created manually in the generated system. This lightweight version is then able to trigger and test one specific fault.

Listing 3.6 illustrates the code which is generated to check the precondition for the *close* transition. 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.

```
case Close() => {

  checkPreCondition(({
    require(data.nonEmpty, s"data should be set, was: $data")
    require(data.get.balance.nonEmpty, s"data.get.balance should be set, was: $data.get.balance")
    data.get.balance.get
  } == EUR(0.00)), "this.balance == EUR 0.00")

}
```

Listing 3.6: Generated Precondition for *close* transition

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

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

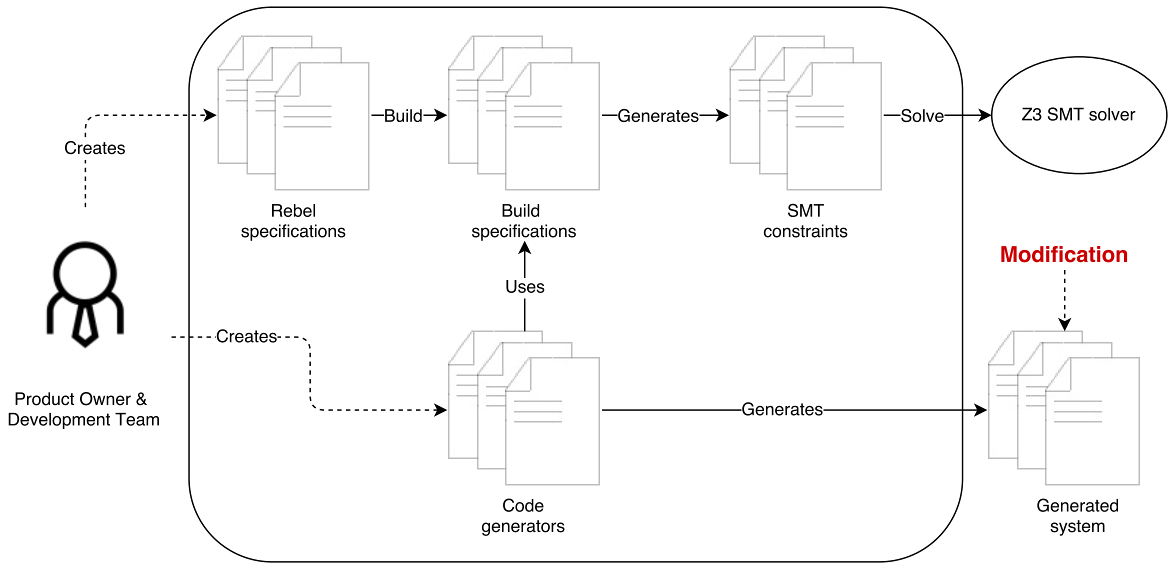

Figure 3.1: Modification in specification development

```
case Close() => {
  RebelConditionCheck.success
}
```

Listing 3.7: Modified Precondition for *close* transition

3.2.1 Evaluation criteria

Faults

Since the precondition of the *close* transition is modified in the SUT, we know that the SUT contains the fault to close an account with some balance. For the lightweight version, it is expected to find this fault 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 fault in the SUT, *i.e.* finding the fault for the *close* transition. Thus from the account specification, we are testing only the transition *close*. However, it is also possible to find faults in the transitions *openAccount* since an account needs to be opened before it is getting closed.

3.3 Approach

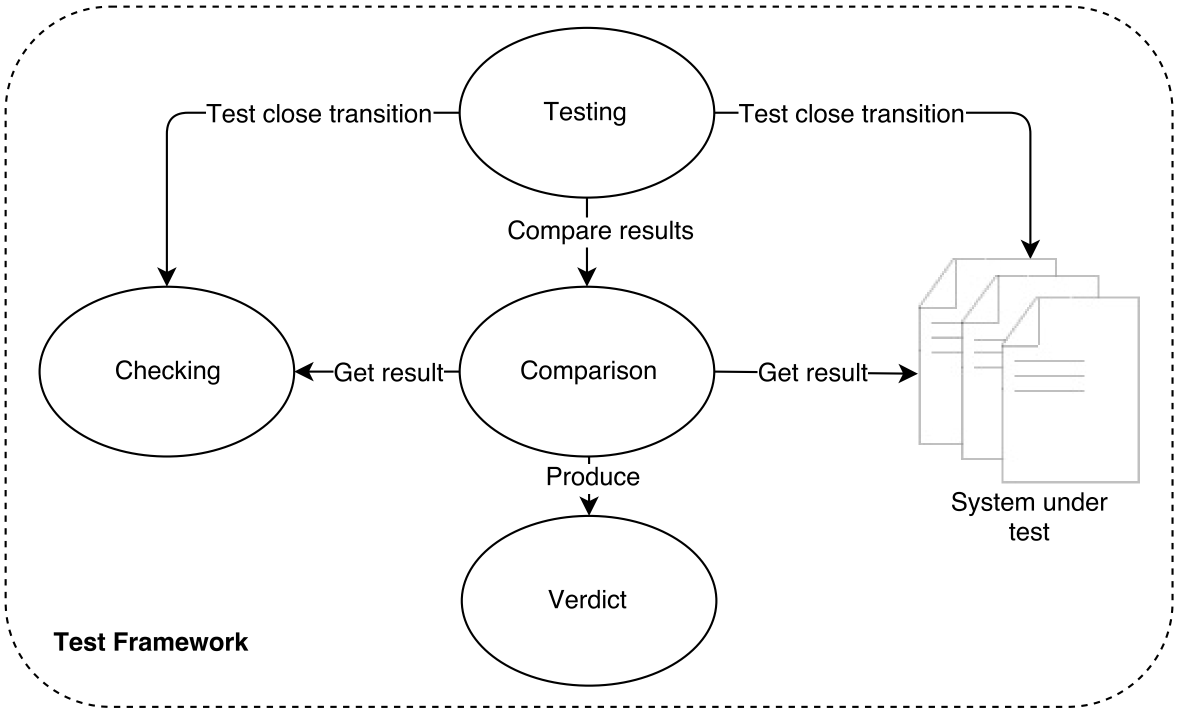

Figure 3.2: Testing approach for *close* transition

The testing approach is shown in Figure 3.2. We discussed earlier that we are going to use the SMT solver to find faults 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, a tebl file is 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 timeout because the state can be reached in less than six steps. The SMT solver tries to solve this problem in max six steps.

```
module simple_transaction.ClosedAccountWithBalance

import simple_transaction.Account

state closedAccountWithBalance {
  closed Account with balance != EUR 0.00;
}

check closedAccountWithBalance reachable in max 6 steps;
```

Listing 3.8: Checking closed account

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 fault 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.

```
{
    "state":{
        "SpecificationState":{
            "state":{
                "Closed":{

                }
            }
        }
    },
    "data":{
        " Initialised ":{
            "data":{
                "accountNumber":null,
                "balance":"EUR 50.00"
            }
        }
    }
}
```

Listing 3.9: Account state after *close* transition

3.6 Evaluation

Faults

The expectation for the faults criteria is to find the fault in the *close* transition. As expected, we did find the fault for the *close* transition due to the proper test which finds the fault. The fault is the result of the manually modified precondition.

Efficiency

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

The model checker provides traces only when a given state is satisfiable. When a state is not reachable, the model checker does not provide traces. This is also the case with checking a closed account with some balance since this state is not reachable. Models (traces) are not available from the Z3 solver when a given SMT problem is unsatisfiable. In this case, traces cannot be used since they are not provided.

Although to reach the given state, the lightweight version uses *openAccount* and *close* transition.

Coverage

The lightweight version is used only to trigger a single fault 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. This is out of scope for the lightweight version.

3.7 Conclusion

As we have seen with this lightweight version, it can find one specific fault 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. According to Voelter, the author of the study [29], 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 faults in the fixed code. The second part is injected code which is generated from models that fill some isolated parts of the fixed code that are dependent on the input of the models. The manually introduced fault also belongs to this category since the preconditions are generated based on the *Rebel* specifications.

Chapter 4

Experiment 1: Invalid execution

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 fault 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 fault. The fault 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.

With the lightweight version, we discussed that the model checker provides traces only when a given state is satisfiable. When a state is not reachable, the model checker does not provide traces. Models (traces) are not available from the Z3 solver when a given SMT problem is unsatisfiable. 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

Faults

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 faults in the SUT where it is possible to perform the opposite of a transition. For example, faults can be found like preconditions which are not properly generated. An example of this is the manually created fault (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 faults, 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 discussed 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. [9, 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. Mutation takes place on checking of the specification and the execution of the transition in the SUT. 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. [9, p. 3-4] The mutation operator for our approach is Negate Conditionals Mutator [30], this operator belongs to the type relational operator replacement [31, 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.

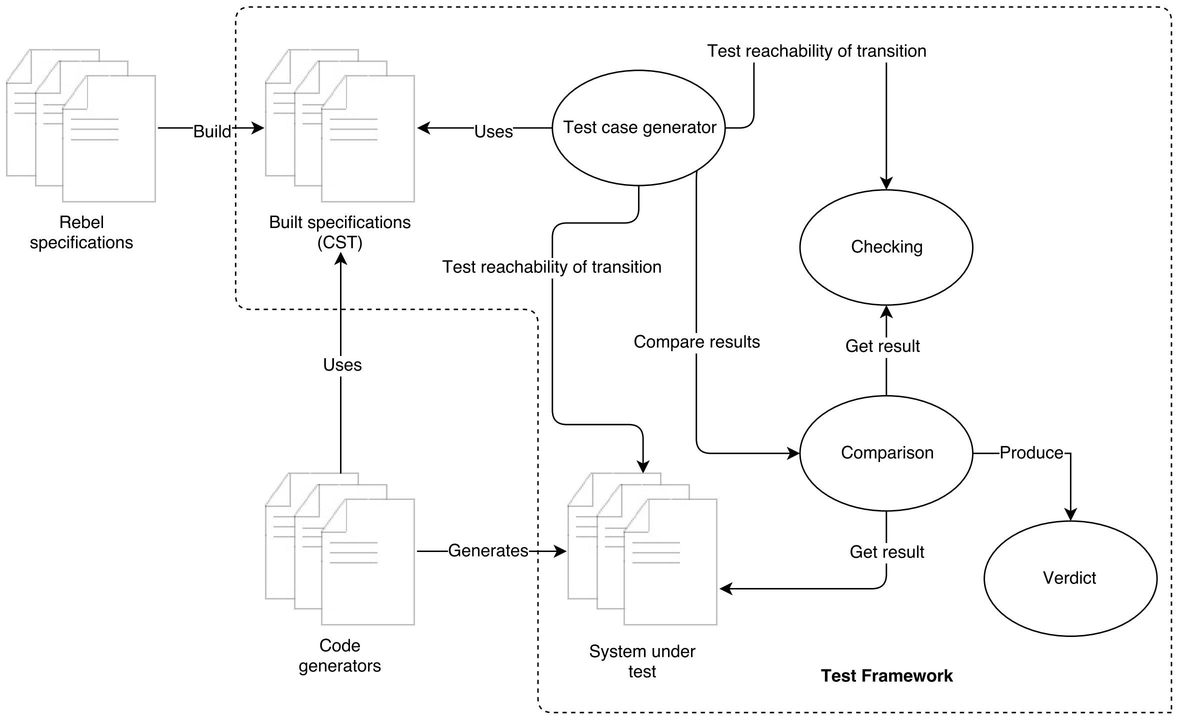

Figure 4.1: Testing approach invalid execution

4.2.1 Mutating checking

Only expressions which contain a reference to the specification fields need 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* 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. Note also that the *this* reference is removed, in tebl specifying *this* is not necessary since the property related to the instance.

Replacing conditionals to negated conditionals is done for all conditionals with relational operators from Table 4.1. 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: Relational conditionals replacement [30]

4.2.2 Mutating transitions

The test case generator must test the reachability in the SUT just like the generated tebl for checking. As discussed before, the results (traces) from checking cannot be used to check the reachability in the

SUT since traces are not available when a state is not reachable.

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. Thus in constructing the initial state, it is not necessary to apply the replacement of conditionals. Of course, with checking the SMT solver constructs its initial state to reach a state.

For example, we now only deal with how the transition *deposit* is executed by the test case generator in the SUT, but the approach also applies to the other transitions. 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. Therefore, the transition *openAccount* is performed in the SUT. 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 on the SUT, 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, the test case generator picks values which satisfy the negated conditionals. To communicate this to the SUT, this transition with its transition parameter values must be converted to *JSON*. For example, the following transition parameter is generated in *JSON* by the test case generator to be used in the *deposit* transition `"amount": "EUR -2.00"`.

```
event deposit(amount: Money) {
  preconditions {
    amount > EUR 0.00;
  }
  postconditions {
    new this.balance == this.balance + amount;
  }
}
```

Listing 4.10: *deposit* transition definition from specification

4.3 Results

4.3.1 Codegen-Javadatomic

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-Javadatomic

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). On line 6 is shown that the model checker states that the state is not reachable (the same tebl file is generated as in Listing 3.8). On the next line, it seems to be that the state is reachable in the SUT. So the test for *close* transition is not successful.

```
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: Results: test run for the *close* transition

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 transition 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 fault in the SUT.

```
[{
  "_id": 17592186045441,
  "_version": 2,
  "_status": "CLOSED",
  "accountNumber": {
    "iban": "NO3627716652225"
  },
  "balance": {
    "value": 52.00,
    "currency": "EUR"
  }
}]
```

Listing 4.12: Account state after *close* transition

Now we know that we have discovered a fault, 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.

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

Listing 4.13: Generated precondition for the *close* transition

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.

```
private str g(e:(Expr)`<Expr lhs> == <Expr rhs>`, tmap t) = "isNotEqual(<g(lhs, t)>, <g(rhs, t)>)"
  when isType(t, lhs, (Type)`Percentage`) || isType(t, lhs, (Type)`Money`);
```

Listing 4.14: Equals expression generator

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.

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.

```
event deposit(amount: Money) {
  preconditions {
    amount < EUR 250.00;
  }
  postconditions {
    new this.balance == this.balance + amount;
  }
}
```

Listing 4.15: *deposit* transition definition from specification

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

Listing 4.16: Compile error in generated system

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

Listing 4.17: Generated precondition for the *deposit* transition

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: GreaterThan and LessThan expression generator

4.5 Evaluation

4.5.1 Faults

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

With this experiment, we have found a fault in the SUT, which was discussed in Section 4.4.1. The other fault is out of scope since the SUT is not able to compile. With the fault 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 fault 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 that the traces are not provided. The reasons for this is that with the opposite preconditions that the state is not reachable, and when the state to reach is not reachable traces cannot be provided by the model checker. 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 criterion 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*. This can be limited by taking a lower configuration timeout in checking. Mainly it remains that with checking it is not possible to focus on a specific transition. 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 criterion 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 constraint.

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

This experiment uses the account specification 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 *deposit*, *withdraw* and *interest*.

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, mainly because it remains that with checking it is not possible to focus on a specific transition. The SMT solver is also smarter/better in checking the satisfiability of a given constraint.

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 fault in the SUT, which was discussed in Section 4.4.1. The other fault is out of scope since the SUT is not able to compile. The found fault 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 fault 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. Testing valid execution can use traces as these states are reachable.

Chapter 5

Experiment 2: Valid execution

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 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 execute 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 transition definitions can contain sync expressions. From the previous transition 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 the implementation of 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. [32, 33, 18]

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. This allows the SUT to run distributed to test with our approach.

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

```
specification  Transaction {
  fields  {
    id: Integer @key
    amount: Money
    from: IBAN @ref=Account
    to: IBAN @ref=Account
  }

  events {
    start []
    book[]
     fail []
  }

   lifeCycle  {
     initial  uninit  -> validated: start
    validated      -> booked: book
             -> failed:  fail
     final  booked
     final  failed
  }
}
```

Listing 5.19: Transaction specification

```
event book() {
  sync {
    Account[this.from].withdraw(this.amount);
    Account[this.to].deposit(this.amount);
  }
}
```

Listing 5.20: *book* transition 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, the traces from the SMT solver are used to check whether the SUT accepts the execution from the traces, and whether the execution is conforms to 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

Faults

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 faults 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 fault 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. [23, p. 4]

It is difficult to build and test distributed systems, according to the study [18, 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 [34, 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. [34, p. 3204] In our case, the expectation is to find faults 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. Some transitions require an initial state for which transitions need to be executed to reach this state. As a result, previously executed transitions can be executed again.

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 faults, 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 generates test cases for all transitions. Using the information from the CSTs, the simulation and checking are requested for a test case for a chosen transition. The simulation and checking provide 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.

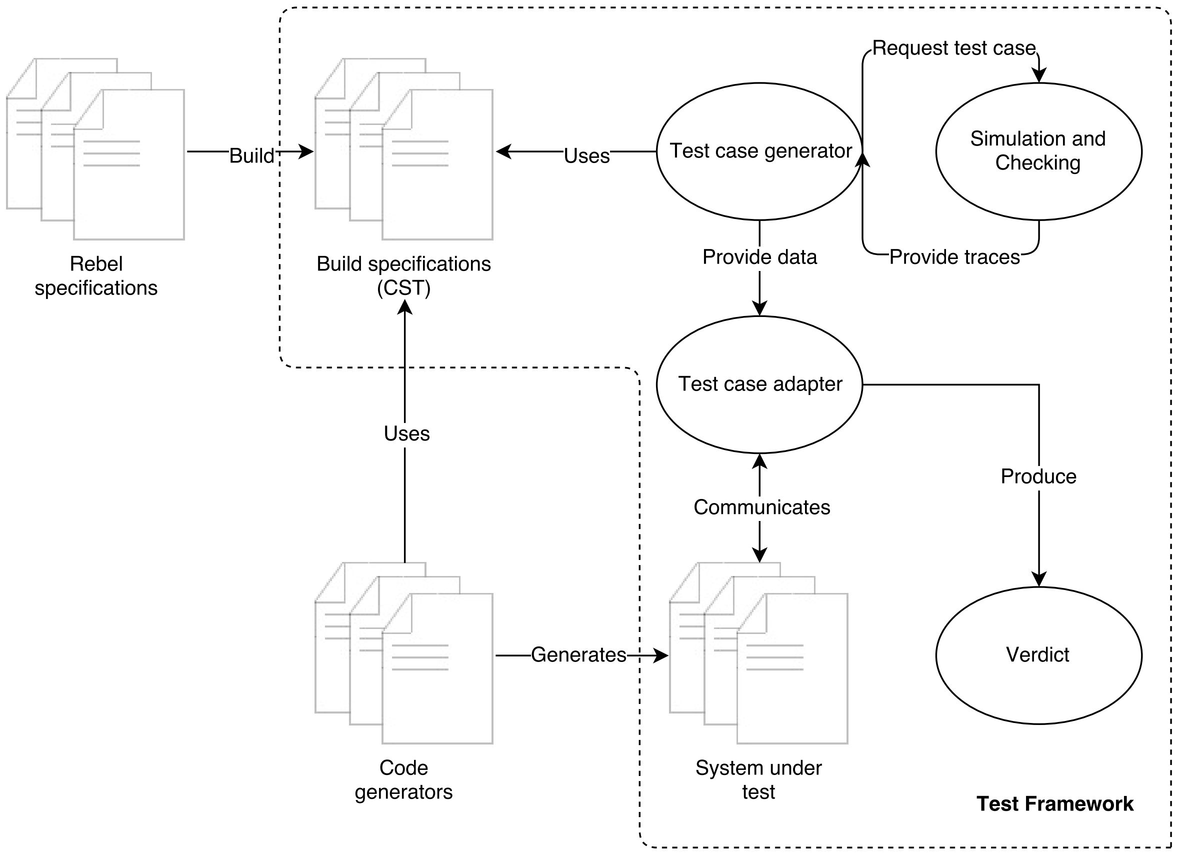

Figure 5.1: Testing approach valid execution

As discussed before, the study [1, p. 5] carried out a model testing approach to test existing banking systems. Although, in this approach, it was only possible to test the SUT interactively using the simulation. The traces from the SMT solver are used in this approach to check whether the SUT accepts the execution from the traces, and whether the execution is conforms to the specification. [1, p. 5] Thus the traces are played in the SUT.

In our approach 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 to reach the resulting state, s_2.

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 *NL10INGB0000001*. 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 *NL10INGB0000002*. 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 [35] 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. The model checker provides traces when the current state is satisfiable. Then the traces are executed on the SUT, which 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 execute the transition.

```
module simple_transaction.Test

import simple_transaction.Transaction
import simple_transaction.Account

state doCheck {
  validated Transaction with id == 65227;

  Account with accountNumber == NO3631174980518;
  Account with accountNumber == LB404150J311SB1FJV5KL1MKYAY4;
}

check doCheck reachable in max 4 steps;
```

Listing 5.21: A tebl for the current state for the transition *book* generated by the test case generator

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. Although, this will not be suitable for expressions like *IBAN*. As discussed earlier, the value generated will not conform to the *IBAN* standards.

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

[2] https://github.com/cwi-swat/rebel/commit/0d29eb30a82cc5dd6d8be750daa4a24e4e2786be

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 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 transition: $p(s_1, s_2)$
2. Check whether the invariants hold in the state after the transition: $P(s_2)$. The function P is the invariants (safety properties).

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 the trace are compared to find any misbehaviour in the SUT.

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 transition 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, the distinct identity of *_state* field needs to be converted 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. [36] 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. [10, 37] 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. [10, 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. [37, p. 5] This leads to that the adapter is dependent on the specification and the SUT.

According to the study [10, 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-Javadatomic

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 in a distributed mode as multiple 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	✗	✗
book	✗	✗
fail	✗	✗

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: Test run for the *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: Test run for the *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 transition 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

```
module simple_transaction.Test

import simple_transaction.Account

state doCheck {
  opened Account with accountNumber == AD3517248539N3OTXZIDF13H;
}

check doCheck reachable in max 4 steps;
```

Listing 5.24: Generated tebl for the transition *book*

Inconclusive state

Listing 5.25 shows the test run 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. Thus the state is inconclusive. This also happens in the *start* and *fail* transitions.

```
Test transition book
validated −> book −> booked

0:
  now = 15 Aug 2017, 11:17
  instance: simple_transaction.Transaction, key = 97691
    ?
    ?
    var id (type: Integer) ( uninitialized )
    var from (type: IBAN) (uninitialized)
    var amount (type: Money) (uninitialized)
    var to (type: IBAN) (uninitialized)

  instance: simple_transaction.Account, key = CY4945493642LWV6W6RZ3EDZSGTB
    ?
    ?
    var accountNumber (type: IBAN) (uninitialized)
    var balance (type: Money) (uninitialized)

  instance: simple_transaction.Account, key = NL60IZNV8233056080
    ?
    ?
    var accountNumber (type: IBAN) (uninitialized)
    var balance (type: Money) (uninitialized)

1:
  now = 15 Aug 2017, 11:17
  step: simple_transaction.Account.openAccount
    var initialDeposit (type: Money) = EUR50.00
    Transition to state = opened
     Identified by accountNumber = NL60IZNV8233056080

  // ... other instances from the state above

  instance: simple_transaction.Account, key = NL60IZNV8233056080
    ?
    ?
    var accountNumber (type: IBAN) = NL60IZNV8233056080
    var balance (type: Money) = EUR50.00

Endpoint: /Account/NL60IZNV8233056080/OpenAccount
JSON payload: { "OpenAccount": { "initialDeposit":"EUR 50.00" } }
Response: ("body":"CommandSuccess(OpenAccount(50.00 EUR))","isSuccessful":"true","message":"OK",
  "errorBody":"","code":"200")
Could not find state ?, expected "state":{"?":{}}
```

Listing 5.25: Test run with inconclusive states

5.5.2 Codegen-Javadatomic

In this test run is the Javadatomic generator used. After investigating the test run, it seems to be that testing the transition *interest* fails. The output of the test run for this transition is shown in Listing 5.26 and Listing 5.27. As you can see in Listing 5.27 on line number 26, the request made for *interest* transition is not successful, and the HTTP status code returned 400 is returned. Constructing the current state for the *interest* transition seems to be successful (see line number 28 of Listing 5.26). According to the simulation, on line number 10, the step interest is made with a negative percentage (-7709). On line number 17, you can see the instance after performing the interest step, which resulted 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.

```
Test transition interest
opened -> interest -> opened

0:
  now = 13 Jul 2017, 12:26

  instance: simple_transaction.Account, key = MD14FLBLJOYGVJMDUZVKLU4C
    ?
    ?
    var accountNumber (type: IBAN) (uninitialized)
    var balance (type: Money) (uninitialized)

1:
  now = 13 Jul 2017, 12:26
  step: simple_transaction.Account.openAccount
    var initialDeposit (type: Money) = EUR50.00
    Transition to state = opened
     Identified by accountNumber = MD14FLBLJOYGVJMDUZVKLU4C

  instance: simple_transaction.Account, key = MD14FLBLJOYGVJMDUZVKLU4C
    State = opened
    ?
    var accountNumber (type: IBAN) = MD14FLBLJOYGVJMDUZVKLU4C
    var balance (type: Money) = EUR50.00

Endpoint: /Account/MD14FLBLJOYGVJMDUZVKLU4C/OpenAccount
JSON payload: { "OpenAccount": { "initialDeposit":"EUR 50.00" } }
Response: ("body":"{\"iban\":\"MD14FLBLJOYGVJMDUZVKLU4C\"}","isSuccessful":"true",
  "message":"OK","errorBody":"","code":"200")
```

Listing 5.26: Test run for the *interest* transition with the use of javadatomic generator (part 1)

```
1:
  now = 12 Jul 2016, 12:00:00

  instance: simple_transaction.Account, key = MD14FLBLJOYGVJMDUZVKLU4C
    State = opened
    ?
    var accountNumber (type: IBAN) = MD14FLBLJOYGVJMDUZVKLU4C
    var balance (type: Money) = EUR50.00

2:
  now = 12 Jul 2016, 12:00:00
  step: simple_transaction.Account.interest
    var currentInterest (type: Percentage) = (− 7709)
    Transition to state = opened
     Identified by accountNumber = MD14FLBLJOYGVJMDUZVKLU4C

  instance: simple_transaction.Account, key = MD14FLBLJOYGVJMDUZVKLU4C
    State = opened
    ?
    var accountNumber (type: IBAN) = MD14FLBLJOYGVJMDUZVKLU4C
    var balance (type: Money) = − EUR3804.50

Endpoint: /Account/MD14FLBLJOYGVJMDUZVKLU4C/Interest
JSON payload: { "Interest": { "currentInterest":"−77.09" } }
Response: ("body":"","isSuccessful":" false ","message":"Bad Request","errorBody":"","code":"400")
```

Listing 5.27: Test run for the *interest* transition with the use of javadatomic generator (part 2))

```
{
  "_id": 17592186045441,
  "_version": 1,
  "_status": "OPENED",
  "accountNumber": {
    "iban": "MD14FLBLJOYGVJMDUZVKLU4C"
  },
  "balance": {
    "value": 50.00,
    "currency": "EUR"
  }
}
```

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

The transition definition for the *interest* transition is given in Listing 5.29. This transition 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.

```
function  singleInterest (balance: Money, interest: Percentage): Money = balance * interest;

event  interest [maxInterest: Percentage = 10%](currentInterest: Percentage) {
  preconditions  {
    currentInterest  <= maxInterest;
  }
  postconditions {
    new this.balance == this.balance + singleInterest(this.balance,  currentInterest );
  }
}
```

Listing 5.29: *Interest* transition definition from account specification

Now we know that we have discovered a fault 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* transition 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.

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

Listing 5.30: Generated precondition of the *interest* transition

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 fault in the SUT for the transition interest. As discussed in Section 3.6, it is possible to have faults in the fixed code. With this experiment and testing the Javadatomic generator, we can conclude that we have found a fault in the fixed code.

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

Listing 5.31: Method used by the generated precondition from Listing 5.30

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 Javadatomic 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 Javadatomic 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.

```
Test transition interest
opened −> interest −> opened

0:
  now = 12 Aug 2017, 18:29
  instance: simple_transaction.Account, key = MT58PDLQ09015VOS06LIF4Q525NRO1I
    ?
    ?
    var accountNumber (type: IBAN) (uninitialized)
    var balance (type: Money) (uninitialized)
1:
  now = 12 Aug 2017, 18:29
  step: simple_transaction.Account.openAccount
    var initialDeposit (type: Money) = EUR50.00
    Transition to state = opened
     Identified by accountNumber = MT58PDLQ09015VOS06LIF4Q525NRO1I

  instance: simple_transaction.Account, key = MT58PDLQ09015VOS06LIF4Q525NRO1I
    State = opened
    ?
    var accountNumber (type: IBAN) = MT58PDLQ09015VOS06LIF4Q525NRO1I
    var balance (type: Money) = EUR50.00

Endpoint: /Account/MT58PDLQ09015VOS06LIF4Q525NRO1I/OpenAccount
JSON payload: { "OpenAccount": { "initialDeposit":"EUR 50.00" } }
Response: ("body":"{\"iban\":\"MT58PDLQ09015VOS06LIF4Q525NRO1I\"}","isSuccessful":"true",
"message":"OK","errorBody":"","code":"200")
```

Listing 5.32: Test run for the *interest* transition with the use of Scala-ES generator (part 1)

```
1:
  now = 12 Jul 2016, 12:00:00
  instance: simple_transaction.Account, key = MT58PDLQ09015VOS06LIF4Q525NRO1I
    State = opened
    ?
    var accountNumber (type: IBAN) = MT58PDLQ09015VOS06LIF4Q525NRO1I
    var balance (type: Money) = EUR50.00
2:
  now = 12 Jul 2016, 12:00:00
  step: simple_transaction.Account.interest
    var currentInterest (type: Percentage) = (− 7709)
    Transition to state = opened
     Identified by accountNumber = MT58PDLQ09015VOS06LIF4Q525NRO1I

  instance: simple_transaction.Account, key = MT58PDLQ09015VOS06LIF4Q525NRO1I
    State = opened
    ?
    var accountNumber (type: IBAN) = MT58PDLQ09015VOS06LIF4Q525NRO1I
    var balance (type: Money) = − EUR3804.50

Endpoint: /Account/MT58PDLQ09015VOS06LIF4Q525NRO1I/Interest
JSON payload: { "Interest": { "currentInterest":"−77.09" } }
Response: ("body":"","isSuccessful":" false ","message":"Bad Request",
"errorBody":"com.fasterxml.jackson.databind.JsonMappingException: Can not construct instance of
squants.Dimensionless: no String−argument constructor&#x2F;factory method to deserialize from String
value (&#x27;−77.09&#x27;)\n at [Source: io.undertow.servlet.spec.ServletInputStreamImpl@578015db;
line: 1, column: 35] (through reference chain: nl.ing.corebank.dto.
account.Interest[&quot;currentInterest&quot;])","code":"400")
```

Listing 5.33: Test run for the *interest* transition with the use of Scala-ES generator (part 2)

```
{
   "_id":"nl.ing.corebank.aggregates.AccountAggregate$|077708cd−769a−48ec−8006−d607241c4f45",
   "_version":1,
   "_state":"OpenedState",
   "accountNumber":{
      "iban":"MT58PDLQ09015VOS06LIF4Q525NRO1I"
   },
   "balance":"50.00 EUR"
}
```

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

Likewise, the Javadatomic generator, we have discovered a fault in the SUT, since the simulated transition is not conform to the specification. To know where this fault 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.

```
@JsonRootName(value = "Interest")
@JsonCreator
case class Interest (@JsonProperty("currentInterest") currentInterest: Dimensionless)
```

Listing 5.35: Generated Interst class

```
str generateDTOBody(tuple[str package, str name, list[Parameter] params] tpl) {
  cn = capitalize(tpl.name);

  return
  "package nl.ing.corebank.dto.<uncapitalize(tpl.package)>
  '
  'import com.fasterxml.jackson.annotation._
  '
  '<lines({"import <i>" | mi <- tpl.params, just(i) := scalaImport(mi.tipe)})>
  '
  '@JsonRootName(value = \"<cn>\")
  '@JsonCreator
  'case class <cn>(<commas([ "@JsonProperty(\"<n>\") <n>: <t>" | mi <- tpl.params,
     t := asScalaType(mi.tipe), n := mi.name ])>)
  '";
}

str asScalaType((Type)`Percentage`) = "Dimensionless";
```

Listing 5.36: Generator of the Interest class from Listing 5.35

Listing 5.35 is generated by the following methods from Listing 5.36. As shown, the method *asScalaType* returns the type *Dimensionless*. The generated *Interest* class from Listing 5.35 is used in Listing 5.37. 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 fault in the SUT for the *interest* transition. The *interest* transition fault in the Javadatomic generator belongs to the fixed code category, but in this case, for the Scala-ES generator, the fault belongs to the injected code category since the *Interest* class is generated and injected into the generated code.

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

Listing 5.37: Request handler for the transition *interest*

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 5.38. 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.

```
Test transition interest
opened -> interest -> opened

0:
  now = 18 Sep 2017, 08:57

  instance: simple_transaction.Account, key = FO1227539908389742
    ?
    ?
    var accountNumber (type: IBAN) (uninitialized)
    var balance (type: Money) (uninitialized)
1:
  now = 18 Sep 2017, 08:57
  step: simple_transaction.Account.openAccount
    var initialDeposit (type: Money) = EUR50.00
    Transition to state = opened
     Identified by accountNumber = FO1227539908389742

  instance: simple_transaction.Account, key = FO1227539908389742
    State = opened
    ?
    var accountNumber (type: IBAN) = FO1227539908389742
    var balance (type: Money) = EUR50.00

Endpoint: /Account/FO1227539908389742/OpenAccount
JSON payload: { "OpenAccount": { "initialDeposit":"EUR 50.00" } }
Response: ("body":"CommandSuccess(OpenAccount(50.00 EUR))","isSuccessful":"true","message":"OK",
  "errorBody":"","code":"200")
Could not find state opened, expected "state":{"Opened":{}}
Could not find value balance, expected "balance":"EUR 50.00"

1:
  now = 12 Jul 2016, 12:00:00

  instance: simple_transaction.Account, key = FO1227539908389742
    State = opened
    ?
    var accountNumber (type: IBAN) = FO1227539908389742
    var balance (type: Money) = EUR50.00
2:
  now = 12 Jul 2016, 12:00:00
  step: simple_transaction.Account.interest
    var currentInterest (type: Percentage) = (- 7709)
    Transition to state = opened
     Identified by accountNumber = FO1227539908389742

  instance: simple_transaction.Account, key = FO1227539908389742
    State = opened
    ?
    var accountNumber (type: IBAN) = FO1227539908389742
    var balance (type: Money) = - EUR3804.50

Endpoint: /Account/FO1227539908389742/Interest
JSON payload: { "Interest": { "currentInterest":"-77.09" } }
Response: ("body":"CommandSuccess(Interest(-77.09))","isSuccessful":"true","message":"OK",
  "errorBody":"","code":"200")
Could not find state opened, expected "state":{"Opened":{}}
Could not find value balance, expected "balance":"EUR -3804.50"
```

Listing 5.38: Test run for the transition *interest* with the use of distributed Codegen-Akka

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 5.38, 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 5.39. 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.

```
Node 2
Endpoint: /Account/FO1227539908389742
Response:
("body":"","isSuccessful":"false","message":"Service Unavailable","errorBody":"The server was not able to
  produce a timely response to your request.\r\nPlease try again in a short while!","code":"503")

Node 1
Endpoint: /Account/FO1227539908389742
Response:
{
   "state":{
      "SpecificationState":{
         "state":{
            "Opened":{

            }
         }
      }
   },
   "data":{
      "Initialised":{
         "data":{
            "accountNumber":null,
            "balance":"EUR -3804.50"
         }
      }
   }
}
```

Listing 5.39: Account state in the SUT after performing the *interest* transition

```
[info] [ERROR] [09/14/2017 14:42:31.146] Failed to serialize remote message
  [class com.ing.rebel.Rebel$CurrentState$CurrentStateInternal] using serializer
  [class com.romix.akka.serialization.kryo.KryoSerializer].
  Transient association error (association remains live)
[info] akka.remote.MessageSerializer$SerializationException: Failed to serialize remote message
  [class com.ing.rebel.Rebel$CurrentState$CurrentStateInternal] using serializer
  [class com.romix.akka.serialization.kryo.KryoSerializer].
```

Listing 5.40: Serialization error in Node 1

When the post-transition check is performed on Node 2, an exception is thrown which is shown in Listing 5.40. This exception is a serialisation error which is thrown because it is not able to serialise the *Rebel* instance account. It is remarkable that this exception appears in Node 1 when the post-transition check is performed on Node 2. Thus it is not possible to retrieve an instance from Node 2 because of a serialisation error.

Now we have only discussed the test results of the *interest* transition. According to the results of Subsection 5.4.4, the other transitions also 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 simulate the transition. To conclude, in this experiment the SUT's does not produce a unanimous final outcome.

5.6 Evaluation

5.6.1 Faults

The expectation for the criteria faults from Subsection 5.2.1 is to find faults in the SUT which does not accept the execution from the traces. For example, a generated pre- or postconditions are not satisfied by the execution from the traces.

In this experiment, we have found faults in the code generators. For both faults, 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 faults. To conclude, we did find faults 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 faults 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 faults 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. The cause of this is serialisation errors in the nodes. 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 faults 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 faults, no faults were found in partial failure or asynchrony. This leads not to the inability to test transitions. Again, not being able to find faults 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. The state is inconclusive.

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

More complex specifications are used in this experiment which implements 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.

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

The experiment from Chapter 5 tests only valid execution from the traces of the SMT solver, *i.e.*, testing only what should be possible according to the specification. Testing valid execution is not enough, testing invalid execution 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 SQ 1: How is the input/output of the generated system tested?

6.1.1 Experiment 1: Invalid execution

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. Traces cannot be used in this experiment since opposite preconditions are not satisfiable. The model checker provides traces only when a given state is satisfiable.

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 checking the satisfiability. With this experiment, we did find two faults in the SUT; only one fault is within our scope. The found faults belong to the fault categories templating and compilation.

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. [10, 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 [38] reports that model-based testing technique using mutation has valuable fault detection effectiveness.

6.1.2 Experiment 2: Valid execution

The experiment of Chapter 5 has solved a few limitation of the previous experiment. This experiment extends the model-based testing approach which was already done with *Rebel*.

This experiment tests valid execution in the generated systems. In comparison to the previous experiment, this approach tests 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. It uses the traces from the SMT solver to check whether the SUT accept the execution from the traces and whether it behaves as the specification. Even the transition parameters data values are generated by the SMT solver which satisfies the constraints of the transition.

In this experiment, the transitions are performed in the SUT in three steps to identify misbehaviour in the SUT. The execution of the transition contains the followings three steps: pre-transition check, transition check and post-transition check. These three steps are used to check that the execution from the SUT matches to the traces of specifications.

More complex specifications are used in this experiment which implements 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. The nodes of the SUT are not consistent since each node may produce a different result. So with this test approach faults can be found in distributed systems.

Although there is a limitation in the testing of states. Within a trace, not every instance may contain the state of it (inconclusive state). 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.

All in all, three faults are found with this experiment in the generated systems and code generators. One fault is identified by a slightly different test approach to find distribution faults. The found faults belong to the fault categories templating and distribution.

6.2 SQ 2: Which false positives occur when the generated system is correctly implemented?

6.2.1 Varying results from the SMT solver

In the first experiment Section 4.5, we already discussed the limitation of this testing approach. This experiment has its limitations due to the inability to use traces since with checking traces are not available when a state is not reachable. As a result, the test run of this experiment produces some false positives. 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 and its values.

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, a random generator is 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.

6.3 SQ 3: What kind of faults can be found and what are the factors?

From the conducted experiments, five faults have been found in the generated system that is generated by the code generators. We can categorise these faults 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 faults 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 faults in the code generators. One fault is from the Javadatomic generator, the code which causes the fault 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 fault in the SUT belongs to the category templating within the fixed code. The fault within the Scala-ES generator belongs to injected code. The identified fault is caused by the generation of code from a given specification.

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

6.3.2 Compilation

In the category compilation belongs faults which are not compilable systems generated by the code generators from a given specification. For this category, we did find a fault which is discussed in Section 4.4.1. Just as mentioned there, this fault 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 faults, although we did not find any faults related to distribution with our experiments when the SUT is run as single nodes.

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. Concepts such as transactions [4, p. 6] and locking [4, p. 10] influence transition atomicity in the implementation of the specifications (generated system). Partial failure and asynchrony makes it also difficult to build and test distributed systems. [18, p. 1]

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. There is no consistency between the nodes since each node may produce a different result. The cause of this is serialisation errors in the nodes.

The experiment from Chapter 5 tests also transitions which contain synchronisation. No faults 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. Also, there does not seem to be any faults in partial failure or asynchrony. As discussed before, the synchronised transitions are also translated to logical formulas. These SMT formulas contain no logic about the implementation of synchronisation. Therefore it should be possible to identify misbehaviour in synchronisation. Several studies also reports to find faults in distributed algorithms with SMT-based approaches to model checking. [32, 33, 18]

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. This experiment has its limitations due to the inability to use traces since with checking traces are not available when a state is not reachable. With this experiment, we did find two faults in the SUT; only one fault 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 accepts 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. The limitation of this experiment is that it is not possible to test all transitions. This experiment is only able to test transitions when the preconditions of a given transition are satisfied by the current state. With this experiment, we did find three faults in the SUT; one fault is identified by a slightly different test approach to find distribution faults.

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

7.1 Future work

Mutation model-based testing

The first experiment has its limitations due to the inability to use traces since with checking traces are not available when a state is not reachable. Instead of applying mutation testing on the execution transition in the SUT, it can be applied to the *Rebel* specification. The study [38] reports that model-based testing technique using mutation has valuable fault detection effectiveness. 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 generated systems, 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.

Bounded checking

Model checking *Rebel* specifications is bounded. The bounded analysis is used to find the smallest possible counterexample. [1, p. 5] Finding a counterexample is automatic and incremental, *i.e.* check incremental whether an invalid state can be reached in steps until a counterexample is found or the configuration timeout is reached. The configuration timeout can also be found in the experiments. Determining this value has been left as a future work. In determining this value, two aspects should be taken into account, namely risk and time. For high-risk systems, a higher configuration timeout can be used to test more extensively. On the other hand, due to a high configuration timeout, testing can be time-consuming.

Invalid current state

The second experiment tests valid execution in the generated systems, with the use of existing testing techniques within *Rebel* to generate tests for transitions, namely checking and simulation. The intention of this experiment is to test all transitions, but it is only possible to test transitions when the preconditions of a given transition are satisfied by the current state. As a result, the simulation is not able to simulate some transitions. To test all transitions, this should be solved. Taking into account, the preconditions which should hold in the current state can be solved by the test framework or the SMT solver. Although it is better to solve this by the SMT solver as this is one of the purposes of the SMT solver.

More complex specifications

In the conducted experiments are only the specifications account or combined with the transaction specification used to test the SUT from these specifications. With these experiments and specifications, we did find faults in the generated systems and the code generators. Although these specifications are quite simple, *e.g.*, the specifications within a bank would be more complex. With such a large amount of complex specifications, the chance is bigger to find faults in the code generators since the specifications are interacting more with each other. This also affects the testing of distributed systems. In our approach, a simple approach has been chosen for testing distributed systems. To find faults like deadlock and race conditions sufficient communication is needed.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, dr. Jurgen J. Vinju, the academic supervisor. He guided me throughout this the thesis, put me in the right direction and always came up with useful suggestions. Without him, the thesis would have taken a different turn. Also, I would like to thank Jouke Stoel, who does his PhD in the field of *Rebel*, for sharing his knowledge. He has also added functionality in the *Rebel* toolchain so that I can carry out my research.

Special thanks go to Jorryt-Jan Dijkstra, my company supervisor, for giving the opportunity to work on my thesis at ING, and for providing feedback on my work. I want to thank my colleagues of ING for the knowledge they have shared. I would also like to thank my co-intern Alex Kok from the UvA, who is also doing his thesis in the field of *Rebel*. Thanks also go to my fellow students during my master Software Engineering. I would like to thank also UvA teacher dr. Ana Oprescu, who is always willing to help students.

Finally, I would like to thank my family and friends who supported me throughout the master.

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 1 st 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] N. Delgado, A. Q. Gates, and S. Roach, "A taxonomy and catalog of runtime software-fault monitoring tools," *IEEE Transactions on software Engineering*, vol. 30, no. 12, pp. 859–872, 2004.

[6] 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).

[7] K. Sen, "Concolic testing," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pp. 571–572, ACM, 2007.

[8] P. Godefroid, M. Y. Levin, D. A. Molnar, *et al.*, "Automated whitebox fuzz testing.," in *NDSS*, vol. 8, pp. 151–166, 2008.

[9] 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.

[10] 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.

[11] J. Tretmans, "Model based testing with labelled transition systems," *Formal methods and testing*, pp. 1–38, 2008.

[12] 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.

[13] K. Havelund and A. Goldberg, "Verify your runs," *Verified Software: Theories, Tools, Experiments*, pp. 374–383, 2008.

[14] Y. Falcone, J.-C. Fernandez, and L. Mounier, "Runtime verification of safety-progress properties.," *RV*, vol. 5779, pp. 40–59, 2009.

[15] M. Papadakis and K. Sagonas, "A proper integration of types and function specifications with property-based testing," in *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, pp. 39–50, ACM, 2011.

[16] K. Claessen and J. Hughes, “Quickcheck: a lightweight tool for random testing of haskell programs,” *Acm sigplan notices*, vol. 46, no. 4, pp. 53–64, 2011.

[17] A. L. Kok, “Property-based testing rebel semantics in the generated code,” Master’s thesis, Universiteit van Amsterdam, the Netherlands, 2017.

[18] C. McCaffrey, “The verification of a distributed system,” *Communications of the ACM*, vol. 59, no. 2, pp. 52–55, 2016.

[19] A. Sanghavi, “What is formal verification?,” *EE Times_Asia*, 2010.

[20] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, “Use of formal methods at amazon web services,” tech. rep., Technical report, 2014.

[21] 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.

[22] J. Herrington, *Code Generation in Action*. Manning, revised ed., 2003.

[23] R. Roestenburg, R. Bakker, and R. Williams, *Akka in action*. Manning Publications Co., 2016.

[24] A. Lakshman and P. Malik, “Cassandra: a decentralized structured storage system,” *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[25] J. Anderson, M. Gaare, N. Bailey, T. Pratley, *et al.*, *Professional Clojure*. John Wiley & Sons, 2016.

[26] M. Fowler, “Cqrs,” *Command Query Responsibility Segregation), http://martinfowler*, 2011.

[27] M. Fowler, “Event sourcing,” *Online, Dec*, p. 18, 2005.

[28] L. D. Moura and N. Bjrner, “Z3: An efficient smt solver,” *Tools and Algorithms for the Construction and Analysis of Systems Lecture Notes in Computer Science*, p. 337340, 2008.

[29] 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.

[30] A. Mkrtchyan, “Pit mutation operators.” `http://pitest.org/quickstart/mutators/`.

[31] 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.

[32] 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.

[33] F. Alberti, S. Ghilardi, A. Orsini, and E. Pagani, “Smt-based approaches to model checking of distributed broadcast algorithms: some case studies,” 2015.

[34] Y. J. Al-Houmaily and G. Samaras, “Two-phase commit,” in *Encyclopedia of Database Systems*, pp. 3204–3209, Springer, 2009.

[35] A. Mkrtchyan, “iban4j.” `https://github.com/arturmkrtchyan/iban4j/tree/ce1ca12b947755d1d96aab294d1ea15c78090ca2`, 2015.

[36] M. Utting and B. Legeard, *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2010.

[37] G. Tretmans and H. Brinksma, *TorX: Automated Model-Based Testing*, pp. 31–43. 12 2003.

[38] 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.