Property-based testing Rebel semantics in the generated code

Alex L.E. Kok
alexkok08@gmail.com

October 11, 2017, 78 pages

Research supervisor: Jurgen J. Vinju
Host supervisor: Jorryt-Jan Dijkstra
Host organisation: ING, http://www.ing.nl
Abstract

Rebel is a domain specific language focused on the financial industry. Banking products can be specified in Rebel. The tool chain for Rebel can be used to check, simulate and visualize Rebel specifications to reason about the specified banking product. The tool chain for Rebel also provides some generators. These generators can generate a system based on Rebel specifications. These generated systems provide an API to work with the specified banking products.

Although the generated system is based on the Rebel specification(s), the generated code is not being checked against the Rebel specification(s). This means that the generated system is perhaps working as expected, or perhaps not. In this thesis we aim to improve this, by automatically testing the generated code using property-based testing and the Rebel language itself.

First we define the expected properties of Rebel, using the existing axioms of algebra as inspiration. Then we write these properties down as “events” in a new Rebel specification. We introduce our test framework, to automatically test this Rebel specification on the generated code. This is done as follows: the test framework uses the existing generator in the Rebel tool chain to generate the system for this Rebel specification. It then generates a test suite containing test cases for each “event” in this Rebel specification. As last, it runs the test suite against the generated system.

With this approach we identified some problems in the generated code that were unknown before. We have found precision errors, overflow/underflow errors and a compilation error. However, the number and the kind of bugs that can be found with this approach depend on the properties that were defined in the first place. We conclude that, by using this approach, the semantics of the generated code can be checked automatically on whether it satisfies the defined properties. The set of defined properties in this thesis is not complete for Rebel. Therefore, it is not the case that all the generated code is tested, but only the generated code concerning the defined properties is being tested.
Chapter 1

Introduction

Large systems often suffer from domain knowledge that is implicit, incomplete, out of date or contains ambiguous definitions. This is what Rebel aims to solve for the financial domain [1]. The toolchain of Rebel can be used to check, simulate and visualize the specifications, allowing to reason about the final product [2]. Checking is done based on bounded model checking by using the Z3 solver.

Generators are being used to generate a software system from the Rebel specifications. The generated system provides an API in order to work with the specified product and handles the database connectivity. However, the implementation of the generated system is not checked against the specifications, meaning that the generated system is perhaps not doing what it is supposed to do according to its specifications. The aim of this project is to improve this, by automatically testing the generated system against a Rebel specification.

1.1 Problem statement

From the Rebel specifications, a system can be generated by the generator. However, neither the generator nor the generated program is being tested against the specification. Thus it could be that the generated system doesn’t work according to what was specified in the Rebel specification. Although the generator should translate everything correctly, we cannot assume that it actually does translate it correctly for each case and that the implementation works as expected.

Currently, there are no tests for the generator or the generated system, during the development of the generator the results are being checked manually. Testing is a major cost factor in software development, with test automation being proposed as one of the solutions to reduce these costs [3]. We aim for an approach such that much of the testing is automated to reduce the time (and costs) needed for testing certain components of the generated system.

The main research question is as follows:

How can the generator in the Rebel toolchain be tested automatically, by using the generated system, to check whether the implementation works as expected?

We investigate the following solution: generating tests based on a Rebel specification and then run these tests against the generated system.

Property-based testing is an approach to validate whether an implementation satisfies its specification [4, 5]. It describes what the program should or should not do. As described in [4]: “Property-based testing validates that the final product is free of specific flaws.”. With property-based testing, a property is being defined which should hold on the system. Next, the property is being tested for a certain number of tries, using different input values to check whether the property holds. In case the property doesn’t hold, it will result in a failure, reporting that there is a case in which the property
doesn’t hold. This indicates that a bug in the system has been triggered.

Property-based testing has already shown a success in earlier studies [4, 6, 7], by detecting errors in a system that were not known before. In this thesis, we will use property-based testing to check the generator, by using the generated system to check whether the properties hold.

We hypothesize that there are yet unknown bugs in the generator, resulting in that the generated system does not work as expected. By using property-based testing we expect to detect bugs in the generator.

To answer the main research question, we will first answer the following research questions:

RQ 1: Which properties are expected to hold on the semantics of the generated code?

RQ 2: How can we test each property as automatically as possible to find bugs in the generator?

RQ 3: What kind of bugs can be found using this approach and how many?

The third research question is influenced by the first research question. When the expected properties are known, we can indicate what kind of bugs can be found by using this approach. Namely, bugs that invalidate these properties. These properties should be meaningful for Rebel, we focus on properties of Rebel when using the available operators and types in the Rebel language.

1.2 Solution direction

The expected properties which we would like to test on the generated code first have to be defined. This is the result of the first research question. The generator in the Rebel tool chain requires a Rebel specification as input to generate a Scala system (called “generated system” throughout this thesis). A Rebel specification contains the states, the data fields and the transitions between the states along with the pre- and post conditions of the transitions. The transitions in a Rebel specification are called “events”. These levels of abstraction can be confusing. In Section 2.4 we describe the confusing terminologies and abstractions in detail to avoid this confusion throughout this thesis. Rebel is described in more detail in Section 2.1.

When the properties are defined, a Rebel specification can be created in which the properties are written down as “events” in the Rebel specification. Each property can be implemented as one “event” in the Rebel specification. Next, a test can be generated from each “event” in this Rebel specification. These tests can then be run against the generated system to check each property on the generated code. When a test fails, a bug has been found. In the next section we will describe this in more detail.

In order to run the test suite, we assume that the generated system can be compiled and that it can be run. But nothing more, such that our approach can be reused between different generators for Rebel. The specification which was used to generate the system should be syntactically and semantically correct. Which means that the Rebel type checker should not report any errors about the specification.

The test framework generates a test suite that can be run against the generated system. In case the test suite finishes without errors, it means that it did not find any bugs and that the generator satisfies the properties that were tested. This doesn’t mean that there are no bugs in the generated system, instead, it means that our test suite was not able to find errors in the properties that it checks for. The generated system will probably still contain bugs which are not detected by using the test framework. In this case, improving the test framework might extend the number of bugs that it can find.
1.3 Research method

To check each property on the generated code, we divide this process in different phases. Most of the phases are executed by the test framework such that minimal manual steps are needed to test the defined properties. An overview of our approach is shown in Figure 1.1, each step is explained in detail in Chapter 4.

![Figure 1.1: Overview of the test framework and the phases](image)

We start off with defining the properties that are expected to hold on the semantics of the generated code. Coincidentally, the Rebel syntax is strong enough such that the defined properties can be written down in the Rebel syntax. This means that a specification can be created from these properties, in which each property is defined as an “event” in the Rebel specification (Figure 1.1, phase 1). Only this phase is done manually, the next phases are executed by the test framework.

This Rebel specification is checked by the type checker and then being built, resulting in a Concrete Syntax Tree (CST) (Figure 1.1, phase 2). The Rebel tool chain provides a generator to generate a Scala/Akka system by using the CST of the specification (Figure 1.1, phase 3). The same CST is used to generate the test suite (Figure 1.1, phase 4). The test suite consists of multiple tests based on the events in the CST. Note that, as described earlier in phase 1, the defined properties are written down as events in the Rebel specification.

The test suite is then being run against the generated system (Figure 1.1, phase 5). After running the test suite, we analyse the results of the experiment. When one or more tests are failing, a bug is found. An investigation of the failing case is needed to discover the actual bug is because the cause of the failing case is not clearly visible in the log.

We evaluate the test framework in each experiment using test coverage and the number of bugs found as metrics. Based on the results of an experiment, we improve the test framework and continue to the next experiment. In the next experiment we run the test framework again with the improvements, analyse the results and evaluate the test framework again.

The full source of our test framework is available on Github\(^1\). However, note that the test framework makes use of the Scala/Akka generator that is written by ING and is closed-source. Thus in order to run the test framework, access to the code of the generator is also required as the test framework depends on this generator.

\(^1\)https://github.com/alexkok/master-thesis
1.4 Contribution & outline

In Chapter 2 we describe the background (Rebel, the generator and property-based testing). In Chapter 3 we provide definitions of the properties that are expected to hold when using the Rebel language and its toolchain. Allowing to reason about whether the properties are satisfied when using the generator and answering the first research question: Which properties are expected to hold on the semantics of the generated code? There were no existing property definitions of Rebel, so we provide a starting point for this with the focus on important properties of Rebel.

The defined properties are being used to check the generator. In Chapter 4 we describe this process in detail, which contributes to answering the second research question: How can we test each property as automatically as possible to find bugs in the generator?

In the experiments (Chapter 5, Chapter 6 and Chapter 7), the test framework is run with the set of defined properties. For each experiment we evaluate the results, investigate the failing tests and evaluate the test framework. The results of the experiments contribute to answering the third research question: What kind of bugs can be found using this approach and how many?

As last, we provide a discussion about the findings and answer the research questions in Chapter 8. Followed up by the conclusion in Chapter 9.

1.5 Related work

1.5.1 Axiom-Based Testing

In axiom-based testing (also known as property-based testing) axioms are defined which describe what the expected behaviour is of the system under test. It was introduced already in the early eighties, with DAISTS [8]. The axioms, which they write themselves, are being used as a basis for unit testing. The set of axioms describe the expected behaviour and are used to determine whether the implementation agrees with the defined axioms. Testing is automated, the user only defines the axioms and writes the implementation for this.

DAISTS focused on algebraic axioms, using the equality operator in the definitions. Axioms can also be specified in object-oriented style. An example of this is ASTOOT [9] which applies the ideas of axiom-based testing. Axioms are defined in an object-oriented style, describing the methods used and the expected outcome. Whereas in algebraic style, the axioms are written more in a functional notation (as is done in DAISTS). In this thesis, we use the algebraic style and use the equality as well as the inequality operators. Furthermore, we also have to define the expected behaviour of Rebel expression.

In [10] axioms are used to automatically generate tests from these. They use the Concepts in C++ which contain the definitions of the axioms. Based on these Concepts they automatically generate tests for each defined axiom and run these tests to check whether the system under test satisfies these axioms. In this thesis, we define the properties (axioms) that are expected to hold in Rebel and then generate tests based on these definitions by using the generator. The generated tests will then be run against the generated system that can be generated from a Rebel specification. Our approach can be compared to the approach in [10], but we define the axioms in a Rebel specification and then run the generated tests against the system that was generated based on the Rebel specification that we created.

1.5.2 Random testing

Random testing is a technique in which random values are being used as input for the test cases. QuickCheck [6] and Randoop [11] are examples of random testing techniques. These differ in how they automatically test systems and what actually is being tested. QuickCheck is based on property-
based testing, which we also use throughout this thesis. Randoop on the other hand is based on feedback-directed random testing.

With feedback-directed random testing, random tests are generated which will immediately be run. The result of earlier test attempts can affect the next test that is being generated, which can be seen as feedback for the next generated test. This allows each test case to ‘learn’ from earlier attempts and to create unique tests.

Randoop is built for Java projects and checks some built-in specifications of Java that can’t be checked by the compiler. The test cases are simple unit tests, consisting of a unique sequence of methods due to the feedback of earlier attempts. The method sequences are unique because it also checks whether the same case has already been checked. Since there can be unlimited sequences of methods to test, the test suite will be terminated after a defined timeout. Next, the result is determined and failing cases are being reported, although when using this approach it cannot determine whether the whole system is correct according to the Java specifications. Instead, it just wasn’t able to find a case for which it fails. This is a useful approach to generate unique tests, but its goal is to check systems built-in Java and thus is not compatible with the semantics of Rebel. It works with calling the Java objects and using methods on those, while the generated system consists of mainly states and events. When using an approach like Randoop with random input values, it cannot be known whether the result of a specific transition was expected to succeed or fail.

Approaches like QuickCheck [6] and Randoop [11] enforce the system under test to be written in a specific language (Haskell for QuickCheck, Java for Randoop). For QuickCheck there are alternative solutions for other languages. In our case, we need to use Rebel when generating test cases, such that we test the generator when generating the tests. Another reason why we can’t use a method like Randoop is that Randoop strictly checks for Java properties, which are not in line with the Rebel language.

### 1.5.3 Testing generated systems

Using domain-specific languages leads to the creation of code generators to automatically generate executable code [12]. This is also the case for Rebel, the toolchain provides some generators which use a Rebel specification as input. These generators and the resulting generated systems can be automatically tested too. In [12] they focus on testing non-functional properties, namely testing the performance of the generated code. In [13] the authors present a regression testing automation framework for a domain-specific language. They detect regressions in the generated systems by generating and executing test cases.

In this thesis, we also focus on automatically testing the generator (by using the generated system) but we check more basic functionalities that are specific to Rebel and its supported types.

### 1.5.4 Product-line testing

In [14] the authors describe a way to efficiently test a software product line. A software product line consists of a set of software systems that share a common set of features. However, many different products can be created from such a software product line, adding a challenge to test each combination of those products. In [14] the authors introduce IncLing, which efficiently tests combinations of these products and take already executed combinations into account when selecting a new combination. To determine the possible combinations they use the IPCL algorithm introduced in [15], an algorithm that can be applied to determine the combinations of software product lines at the size and complexity found in projects in the industry.

IncLing executes its tests sequentially by selecting a combination of features that were not covered in the previous test. This is a different way of testing each possible combination and ensuring unique test cases. In this thesis, we basically define the possible combinations by defining the properties
(axioms) that we expect to hold in Rebel. Additionally, the input values are determined random, allowing different combinations as in values. However, we do not test combinations of the properties combined, but test each individually. In our approach there could be some overlap over the various defined properties.

1.5.5 Model-based testing

Model-based testing is an evolving technique to generate tests based on a model [16]. The model is used to describe the requirements to which the system under test. Essentially, the model is a “specification” of the inputs to the system under test. To avoid confusion with our thesis, we stick with “requirements” in this section.

From the data model, tests are being generated and run against the system under test. The expected outcomes of the test cases are known beforehand based on the requirements. Thus, after running the tests against the system under test, the actual results are compared to the expected results. When the results are different compared to each other, it indicates that there is an error.

In Rebel a banking product can be specified. Which is essentially the abstract model of the product. When using the model-based testing approach one could check a generated system with the Rebel specification that was used to generate that system. Another master’s thesis was done in parallel on the same topic [17]. They use model-based testing to test the generated system and uses the Rebel specification that was used to generate the system as data model. They use the existing solver that is available in the Rebel tool chain to generate tests. Next, they run each test against the generated system.

Although the topics are similar, the approach and the results are different. We will describe the differences in between these approaches in detail shortly. The results are different in that the bugs found are different. For example, they found distribution errors while we found precision errors. It is also important to note that we have only used one generator throughout our thesis, while [17] uses multiple generators.

We expect that some bugs found in the other generators can be detected with our approach too, such as the incorrect implementations for certain operators. When the properties have been defined for those, we should be able to detect those. However, we did not find such errors in the generator that we used when testing the properties that we defined. However, the vice-versa is also possible, their approach could probably find the bug we have found. Although they are depended on the traces that are being generated by the solver.

We discuss the differences in input, the test cases and the number of tests between these approaches:

Input specification(s)

The first difference are the input specifications that are being used to test the generated system. In our approach we create a new Rebel specification based on the defined properties. This can be done due to the fact that the Rebel syntax is strong enough to write the properties down in Rebel. Then we use this specification to generate tests for the generated system.

With the model-based testing approach in [17], existing Rebel specifications are used as model to test the generated system. The Rebel specifications for the model are the same specifications as those that were used to generate the system.

From this, we can conclude that: in our approach we test the generated system on certain properties. Exactly these properties should then be tested for all Rebel specifications. Thus we test things that should hold for all Rebel specifications. While with the model-based testing approach only specific Rebel specifications are being tested on the generated system. Allowing to reason about specific Rebel specifications, but requires test runs for each specification. While with our approach, we run
CHAPTER 1. INTRODUCTION

the test framework once and reason about all Rebel specifications.

Tests

Another difference is the way how the tests are being determined and generated. In our approach we generate a test case for each “event” in the Rebel specification. Remember that the properties are being written down as “events” in the Rebel specification that we create. Thus we know exactly what we test. A test case consists of checking the property 100 times with random input values. These test cases can be considered “unit tests” as each test checks one and only one property definition. We first generate all the test cases and then run the test suite against the generated system.

In [17] they use the SMT solver to generate a test case. Note that they use existing Rebel specifications as input, which already define the possible transitions for a specific banking product. Due to the SMT solver that they use, the actual test cases are unknown as these are determined by the SMT solver. The test cases can differ in each run and thus cannot exactly know the contents of each test. However, due to the solver they assume that the test case is valid and thus finds bugs when a test fails.

Thus we generate unit tests and run the whole test suite against the generated system. While in [17] they use the SMT solver to generate the tests at runtime, requiring to have both the solver and the generated system to be running available while the test framework is running.

Number of tests

The number of tests that are done also differ. Since we generate unit tests for each property, many tests are being generated. Additionally, each test consists of checking a property 100 times with random input values. Despite of the many generated tests, the execution time of the whole test suite is considerably fast since each test is considerably small. Although we did not measure the efficiency in our experiments specifically.

In [17] they generate a test case by using the SMT solver. The test is then run against the generated system, checks the result and continuous to the next test by repeating this process. Executing such a test case can be considered slower compared to our test cases. Although we test each property 100 times in a single test case, increasing this amount would affect the execution time too. However, we can reason about properties on all Rebel specifications while they can reason about a existing Rebel specifications.
Chapter 2

Background and context

2.1 Rebel

Rebel is a domain specific language that focuses on the banking industry [1]. Banking products can be specified in the language, with the use of Rebel types like Money, IBAN and Percentage. In Subsection 2.1.1 an example of a Rebel specification is described.

The tool chain of Rebel allows to check, visualize and simulate the specified banking product. For checking and simulation an efficient, state-of-the-art SMT solver is being used called Z3, which is developed by Microsoft [18]. Rebel is written in Rascal and is developed by the ING in corporation with the CWI. Currently, the tool chain of Rebel is also written in Rascal.

Checking a Rebel specification is based on Bounded Model Checking [1]. The Rebel specification is being translated to SMT constraints, next, the Z3 solver is being used to check whether the specification is consistent. An inconsistent specification means that a counter example has been found (meaning that a trace is found for which an invariant doesn’t hold). It is bounded since it only checks if a counter example can be found within a predefined number of steps. Besides checking the specification, the specification can also be simulated. For simulation, the SMT solver is also being used to determine whether a transition can happen. After successfully checking the specification, meaning that no counter examples could be found, the result is still that the specification ‘might’ be valid. As the checking method is bounded, it stops at a certain point (which, in the Rebel toolchain, is defined as the maximum depth of the traces that can be used for checking). This means that there can still be a long or untested trace for which the invariant doesn’t hold.

From the Rebel specification, a system can be generated by using a generator which is developed by ING. The generators are also written in Rascal. This requires a specification that is consistent and that does not trigger errors by the type checker.

2.1.1 A Rebel example

In Listing 2.1 an example of a simple bank account specification is shown in Rebel. In this specification, an account can be opened and money can be deposited to an opened account.
CHAPTER 2. BACKGROUND AND CONTEXT

The `module` keyword describes the module of this specification. The `import` statement is used to import logic from another file, which we will cover when describing the `events` block. The specification of the account itself is defined in the `specification` block, which consists of the following three blocks:

Fields

The `fields` block describes which data is available in this bank account. In this case, `accountNumber` and `balance` which are of type `IBAN` and `Money` respectively. The `accountNumber` is used as identifier for this account specification, because of the `@key` notation behind the type.

Life cycle

The `lifeCycle` block describes the life cycle, it can be seen as the life cycle of a finite state machine. It describes the states and the transitions between them. It also defines the `initial` and `final state`. Note that the transitions between these states are the events that have been defined in the specification.

Events

In the `events` block the events for this specification are described. The optional parameters of the event definition can be overloaded inside the brackets. Note that the implementation of the events are not described here. This is defined in another file called `Library`, shown in Listing 2.2.
An event definition can consist of preconditions, postconditions and a sync block. Each of those can only occur once in an event definition. The sync block defines that each operation inside that block should execute synchronously, without the interaction of other events in the meanwhile. However, in this simple specification the sync block is not used.

The preconditions and postconditions blocks contain expressions and can use functions. These functions can be defined in Rebel too. The new keyword in the postconditions block describes that the value of the field should be updated when the event transition takes place.

### 2.1.2 Rebel checkers versus our approach

A Rebel specification can be checked by the Rebel tool chain, while we are checking properties when using the generator. Additionally, the Rebel tool chain also has a type checker. In this section we aim to solve this confusion by describing both in more detail.

The Rebel tool chain contains a checker which checks a Rebel specification using bounded model checking. It is limited in that it only checks a Rebel specification, by trying to find cases in which the invariant doesn’t hold. It does not do anything with Rebel specifications without an invariant. Thus, it only checks a Rebel specification and is not able to check or test the implementation of the generator or the generated system. The input is a set of Rebel specifications, only these Rebel specifications are being checked, nothing more. It only supports checking a Rebel specification. Due to this it cannot be used on the generated system, as this is written in Scala.

The type checker in the Rebel tool chain is used to determine whether certain expressions are allowed in Rebel. Operations like +, -, *, and / can be used in Rebel, but these are not compatible with each and every type. For example, a Money value cannot be multiplied by another Money value. Such errors should be prevented by the type checker. The type checker will also be taken into account when defining the properties in Chapter 3.

In our property-based testing approach we check whether certain properties hold when using the generator. These properties describe the expected working of the Rebel language. Due to the syntax of Rebel, we can define a Rebel specification containing these properties. The type checker will prevent type-related errors in the property definitions. Our test framework uses this Rebel specification to check whether these properties hold when using the generator. Thus, all defined properties are being checked by only running the test framework once.
CHAPTER 2. BACKGROUND AND CONTEXT

This means that, in our approach, we test whether the generated code works as expected for every Rebel specification that can be created. While the existing checker in the Rebel tool chain, that uses bounded model checking, only checks whether a Rebel specification is consistent. This existing checker is not able to check whether the generated system is working as expected. In our approach we check the generated system, but we only check parts of the whole implementation. Namely, the parts which are covered by the defined properties.

2.2 The Scala/Akka generator

There are multiple generators developed within ING. The generators are different in that the resulting product is written in a different language or uses a different implementation, like database or messaging layer. Each generator is written in Rascal.

The Scala/Akka generator is one of these generators. It generates a system that is written in Scala, uses Akka [19] as actor system and uses Cassandra [20] as database. This generator is often used within experiments inside ING and is considered the most mature generator among the currently developed generators. Because of this, it’s interesting if we encounter yet unknown problems in the generator itself or the resulting system. We will only make use of this generator throughout this thesis.

A specification can be defined in terms of a Labeled Transition System [1], containing the states, the data fields and the transitions between the states along with the pre- and post conditions of the transitions. The generated system is based on these states and transitions defined in Rebel, resulting in a system that works like an Extended Finite State Machine. Thus the generated program also implements it like states and transitions between them. An instance of a banking product can have fields and is in a specific state. In order to go to another state, a transition can be done which might have pre- or post conditions. In case there are pre- or post conditions, these have to be satisfied in order to successfully complete a transition.

Rebel introduces custom types, such as IBAN. A library or own implementation is used for the custom types available in Rebel. An example is the Money type, which is available in the Squants [21] library. The generated system uses this library to deal with the Money type and its operations. Another example is the Percentage type, which is simply translated by calling a method percentage(). In order to conclude that the generated program is doing something incorrectly, we have to specify what the expected properties are in Rebel.

2.3 Property-based testing

With property-based testing properties are defined and being tested. It uses random values as input and checks whether the defined property holds. After a certain number of succeeding cases the test succeeds and the next property is being checked.

A well-known tool which is based on property-based testing is QuickCheck, which is written for Haskell [6]. It tests the properties automatically by using random input values. For each property QuickCheck tries to find counter examples, which are a set of values for which the desired property does not hold. If 100 test cases are succeeding in a row, it goes on to the next property. In case it found a counter example, it will try to minimize the values to try to report the edge case of the failure. However, one might have properties that only hold under certain conditions. For this QuickCheck allows using preconditions. Although, this doesn’t work well for every case as QuickCheck will just generate new pairs of values in case the precondition didn’t hold for the generated set of values. An example of this is where 2 of the input values have to be equal, the chance that this happens with random values is rather low. QuickCheck will try to generate new values each time, with a maximum of 1000 tries by default. In case this maximum is reached, it reports the case as “untested” and
CHAPTER 2. BACKGROUND AND CONTEXT

continues to the next property. Note that these values of 100 and 1000 are the default values, these can be adjusted when needed.

Due to the effectiveness of QuickCheck, many ports for other languages were written. Most ports implement the basics of QuickCheck, additionally, each port could have added extra features. Examples of some ports are FortressCheck (for Fortran) [22] and ScalaCheck (for Scala) [23]. FortressCheck supports polymorphic types and, unlike QuickCheck, heavily uses reflection for its value generation to solve certain problems with polymorphic constructs. Although Scala supports polymorphism, ScalaCheck does not test this [22].

There is no QuickCheck implementation for Rebel. This is not interesting for Rebel because it is a declarative language, it does not execute. However, when testing the generated code it is interesting, which is what we do in this thesis. Since the generated system is written in Scala, ScalaCheck might be applicable for us. However, this would result in using ScalaCheck as a black box, implementing the random functionality ourselves makes sure we know what is going on. Thus resulting in a white box implementation for our test framework. We can also modify it to our needs when we want to improve our test suite. One of the things that we want to improve is to generate values under certain conditions. For example, if a property only holds under a certain condition, the chance that random values satisfy the condition can be very low. Resulting in a test case that wouldn’t do anything most of the time. An example of such a property is Symmetry ($x = y \implies y = x$).

Additionally, we might also need to slightly interact with other components, such as the messaging layer, that the generated system uses. This could make the implementation more complicated when using ScalaCheck due to the format of a property test when using ScalaCheck.

2.4 Terminology

In this thesis there are some levels of abstraction, the terminology used throughout this theses can cause confusion. Words like “specification”, “properties” and “tests” generally can have a different meaning depending on the context. In this section, we describe the confusing terminologies and abstractions in detail.

Specification
In this thesis we use the word “specification” exclusively to indicate banking products described in the Rebel language. This includes the state machines (life cycle), pre and post-conditions and logical invariants.

Properties
We use the word “property” to describe semantic properties of the Rebel language. The set of properties we introduce in this thesis can be seen as a partial “specification” of the semantics of Rebel, but we do not use this word to avoid confusion. We stick with “properties”.

Implicative property
A “property” that uses the implication ($\implies$) operator in its definition.

Generator(s)
The generator(s) that can be used to generate a system based on a “specification”. When using the term “the generator”, we refer to the Scala/Akka generator which is the only generator that we use throughout this thesis.

Test framework
The test framework that was developed during this thesis. Which builds the specification, generates a system from the specification by using the generator, generates the test suite and runs the test suite against the generated system.

Tests
The generated tests by the “test framework”, intended to check whether a certain “property” holds.
CHAPTER 2. BACKGROUND AND CONTEXT

**Test suite**
The collection of generated tests by the “test framework”, along with its configurations which can be run to test the generated system. Note that the test suite initially doesn’t exist. Instead, it is being generated and added to “the generated system” when we run the “test framework”.

**Events**
The event definitions in a “specification”, these define the transitions between the states.

**Scala Build Tool (SBT)**
The tool that is used to compile and run the “generated system” and the “test suite”.

**Concrete Syntax Tree (CST)**
A “specification” can be checked and built. This results in a “CST” containing the data of the specification. This “CST” is being used by the toolchain of Rebel and by the “test framework”.
Chapter 3

Properties of Rebel

_Rebel_ introduces custom types, like _IBAN_, _Percentage_ and _Money_ and allows operations on those [1]. But what are the expected properties of the generator? In this chapter, we will try to answer the first research question:

**RQ 1:** Which properties are expected to hold on the semantics of the generated code?

To answer this question, we first describe a way how we can determine the properties. Followed by the property definitions that will be used throughout this thesis, with a motivation why these properties are expected to hold. The properties that are defined in this chapter will be used in our experiments (Chapter 5, Chapter 6 and Chapter 7).

### 3.1 Determining the properties

Currently, there are no definitions available of which properties are expected in _Rebel_. Due to the missing definitions of these types, it means that we first have to define what the expected properties on these types are and substantiate these. Only then we can determine whether the generator is working as expected with these properties.

There are many operations available among the available types in _Rebel_. We are not able to define all the properties that exist in the _Rebel_ language, because this is not effective for the scope of this thesis. Our goal is to show that this approach works. Next, one can write all the relevant properties down and use the test framework to test these properties automatically. During this thesis we will focus on the _Money_ type, considering this is the most important type for a bank and has the highest priority to be implemented correctly.

For types like _Integer_, the axioms of algebra can be used as inspiration to determine whether the implementation is correct. An _Integer_ in _Rebel_ is most likely translated to an _Integer_ in the generated system too, with perhaps the expectation that these have the same properties in _Scala_. However, it is not possible to rely on the _Integer_ definition of a specific language. Because another generator might generate a system in another language or might implement it differently in the same language. Would that mean that the properties of that other language should now hold on the _Integer_ type? Well, as this is not defined for _Rebel_, this is unknown. In this chapter, we will define properties that are expected to hold when using the _Money_ type in _Rebel_. The properties that we define are based on the known axioms in algebra [24, 25, 26]. We provide an explanation of why a certain property is expected in _Rebel_.

The _Money_ type can be seen as a currency with an amount value. The amount of a _Money_ value can have multiple decimals depending on the currency. Thus, the amount can be seen as a floating number. Does this mean that it inherits the computational properties of Floating-Point Arithmetic, as defined in the IEEE standards 754 or 854? Since the _Rebel_ is intended to be a formal specification
CHAPTER 3. PROPERTIES OF REBEL

language for banking products, we don’t expect that the described problems with this arithmetic are intended to exist on the Money type. Considering high volume flows within a bank in terms of Money, using the Floating-Point Arithmetic properties can result in the known precision, overflow and underflow errors as described in [27]. Such errors could be avoided by using the Money type. The author of [28] also describes that the intention of the Money type is to avoid this:

“You should absolutely avoid any kind of floating point type, as that will introduce the kind of rounding problems that Money is intended to avoid.” – Martin Fowler [28]

In [28] the operations that can be done with the Money type are described, which are: +, -, *, allocate, <, >, ≤, ≥ and =. Where the allocate method is used instead of the division (/) operation. This is due to the division problem, requiring a number to be rounded off at a certain time. For example, when splitting 1 EUR by 3, each part would be 33 cents with the consensus of two decimals. However, what is done with the last cent that is left? This is the problem that is being solved by the allocate method, representing the ratio in which the rest amount should be allocated (where the last cent would go to in this case) [28]. The allocate method is a thing that Rebel does not have, instead, it just allows the use of the division operator. Because of this, we expect the division problem to occur while running the test framework.

The amount of a Money value is often rounded when it is being represented to the user, as it could have many decimals. The representation of the Money value is up to the business on how this is done, as there are multiple factors influencing this. Instead, we only focus on the internal value that is used when operating with the Money type.

It is unsupported to use operations on Money values when using values that are of different currencies. This could be done by taking the exchange rates into account, as described in [28]. However, this is not implemented in Rebel yet and thus it is unsupported to use operations on it when using different currencies.

When using equality both the currency and the amount are taken into account, which should be equal for both variables. In case of different currencies, it is not considered equal, even if the amount would be equal.

We say that the amount of a Money value in Rebel should hold the exact value as if we would calculate the same expression by ourselves. The representation of the Money value to the customer is up to the business.

3.2 Property definitions

The properties that we define are based on the known axioms in algebra, although not every property is relevant or holds for each type in Rebel.

For example, it isn’t possible to multiply two Money types with each other in order to support the multiplicative property. It is unknown to which dimension “Money squared” belongs, this does not occur in the financial domain. Because of this, detecting this prevents such mistakes. When this is accidentally defined in the Rebel specification, the type checker would detect and report this as an error. Division has a similar case, as something cannot be divided a Money type.

Instead we can use these operations with Money in combination with other types, such as Integer and Percentage. The type checker for Rebel will be used to determine these combinations. Due to this, one property definition can lead to different possible combinations which can be used. This results in multiple property definitions when using different types, for each definition a unique name is being defined to identify each separate property definition.

We can separate the properties into two categories: “Field properties” and “Properties of equal-
ity and inequality”. In the following sections we describe these categories in detail along with the properties in the category. These property definitions will be used throughout the thesis. Additionally we will describe some property definitions that were added to this list of properties in the third experiment (Chapter 7).

3.2.1 Field properties

The field properties define the behaviour of addition, multiplication, subtraction and division on rational and real numbers\(^1\). As described in [26], such operations can be used to determine the sum and the product of such values:

“Along with the set \( \mathbb{R} \) of real numbers we assume the existence of two operations called *addition* and *multiplication*, such that for every pair of real numbers \( x \) and \( y \) we can form the *sum* of \( x \) and \( y \), which is another real number denoted by \( x + y \), and the *product* of \( x \) and \( y \), denoted by \( xy \) or by \( x \cdot y \).” – Tom M. Apostol [26]

These operations are also available when using the *Money* type in Rebel and are expected to apply these operations on the amount value\(^2\). However, not every combination is possible in Rebel when it comes to using these operations. This is because Rebel does not support certain combinations. We define the following properties in this category:

**Associativity**

<table>
<thead>
<tr>
<th>Formula</th>
<th>Property name</th>
<th>Variable (Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>((x + y) + z = x + (y + z))</td>
<td>associativeAddition</td>
<td>(x : \text{Money})</td>
</tr>
<tr>
<td>(\quad \quad \quad y : \text{Money})</td>
<td>(z : \text{Money})</td>
<td></td>
</tr>
<tr>
<td>((x \cdot y) \cdot z = x \cdot (y \cdot z))</td>
<td>associativeMultiplicationInteger1</td>
<td>(x : \text{Integer})</td>
</tr>
<tr>
<td>(\quad \quad \quad y : \text{Integer})</td>
<td>(z : \text{Money})</td>
<td></td>
</tr>
<tr>
<td>((x \cdot y) \cdot z = x \cdot (y \cdot z))</td>
<td>associativeMultiplicationInteger2</td>
<td>(x : \text{Money})</td>
</tr>
<tr>
<td>(\quad \quad \quad y : \text{Integer})</td>
<td>(z : \text{Integer})</td>
<td></td>
</tr>
<tr>
<td>((x \cdot y) \cdot z = x \cdot (y \cdot z))</td>
<td>associativeMultiplicationPercentage1</td>
<td>(x : \text{Money})</td>
</tr>
<tr>
<td>(\quad \quad \quad y : \text{Percentage})</td>
<td>(z : \text{Integer})</td>
<td></td>
</tr>
<tr>
<td>((x \cdot y) \cdot z = x \cdot (y \cdot z))</td>
<td>associativeMultiplicationPercentage2</td>
<td>(x : \text{Integer})</td>
</tr>
<tr>
<td>(\quad \quad \quad y : \text{Money})</td>
<td>(z : \text{Percentage})</td>
<td></td>
</tr>
</tbody>
</table>

**Table 3.1:** Associativity when using *Money*

The law of associativity is known on addition and multiplication [24]. It defines that the order in which certain operations are done does not affect the result of the whole expression. In Table 3.1 we define possible combinations for this property, which are accepted by the type checker of Rebel.

---

\(^1\)https://en.wikipedia.org/wiki/Field_(mathematics)  
\(^2\)We do not claim that the *Money* type in Rebel is a field. As it does not support each described operation when only using *Money*. Instead, we use the field definitions to determine the properties on the *Money* type.
CHAPTER 3. PROPERTIES OF REBEL

Non-associativity

<table>
<thead>
<tr>
<th>Formula</th>
<th>Property name</th>
<th>Variable (Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>((x - y) - z \neq x - (y - z))</td>
<td>nonassociativity</td>
<td>(x : \text{Money}) (y : \text{Money}) (z : \text{Money})</td>
</tr>
</tbody>
</table>

Table 3.2: Non-associativity when using Money

In contrast to associativity (Associativity), non-associativity describes that the order of the arguments does affect the result of the whole expression. As we can see in Table 3.2 subtraction is a relation where this property holds. An exception to this would be when each argument is zero.

Commutativity

<table>
<thead>
<tr>
<th>Formula</th>
<th>Property name</th>
<th>Variable (Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x + y = y + x)</td>
<td>commutativeAddition</td>
<td>(x : \text{Money}) (y : \text{Money})</td>
</tr>
<tr>
<td>(x \cdot y = y \cdot x)</td>
<td>commutativeMultiplicationInteger1</td>
<td>(x : \text{Integer}) (y : \text{Money})</td>
</tr>
<tr>
<td>(x \cdot y = y \cdot x)</td>
<td>commutativeMultiplicationInteger2</td>
<td>(x : \text{Money}) (y : \text{Integer})</td>
</tr>
<tr>
<td>(x \cdot y = y \cdot x)</td>
<td>commutativeMultiplicationPercentage1</td>
<td>(x : \text{Percentage}) (y : \text{Money})</td>
</tr>
<tr>
<td>(x \cdot y = y \cdot x)</td>
<td>commutativeMultiplicationPercentage2</td>
<td>(x : \text{Money}) (y : \text{Percentage})</td>
</tr>
</tbody>
</table>

Table 3.3: Commutativity when using Money

These properties are based on the commutative law [24]. The result of an addition or multiplication does not vary when swapping the input variables. Because of the Money type, we can only do addition on Money values with other Money values. As described in earlier, it is not possible to multiply two Money variables with each other, but it is possible to multiply it by an Integer or Percentage. Also in this case, the order shouldn’t matter if we would put the Money value as the first input parameter to multiplication or the other way around.

Anticommutativity

<table>
<thead>
<tr>
<th>Formula</th>
<th>Property name</th>
<th>Variable (Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x - y = -(y - x))</td>
<td>anticommutativity</td>
<td>(x : \text{Money}) (y : \text{Money})</td>
</tr>
</tbody>
</table>

Table 3.4: Anticommutativity when using Money

Commutativity described the commutative properties. Note that the properties in Commutativity only use addition and multiplication. Subtraction is an operation that is anticommutative as swapping the order of the two arguments is negates the result. The anticommutative property thus negates the
result of swapping the two arguments, intending to result in the actual value again, as shown in Table 3.4.

Identity

<table>
<thead>
<tr>
<th>Formula</th>
<th>Property name</th>
<th>Variable (Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x + 0 = x$</td>
<td>additiveIdentity1</td>
<td>$x : \text{Money}$</td>
</tr>
<tr>
<td>$0 + x = x$</td>
<td>additiveIdentity2</td>
<td>$x : \text{Money}$</td>
</tr>
<tr>
<td>$x \cdot 1 = x$</td>
<td>multiplicativeIdentity1</td>
<td>$x : \text{Money}$</td>
</tr>
<tr>
<td>$1 \cdot x = x$</td>
<td>multiplicativeIdentity2</td>
<td>$x : \text{Money}$</td>
</tr>
</tbody>
</table>

Table 3.5: Identity when using \textit{Money}

The identity relation describes a function that returns the same value as the value that was given as input. For additivity, this entails the addition of zero to the input value and for multiplicativity, this entails multiplying the value by 1. Also, the Commutativity property holds here, as the order does not matter in which this function is applied. Since it is not possible to just add 0 to a \textit{Money} value, the 0 showed in Table 3.5 must be defined in a \textit{Money} format. Thus it must have the same currency as the parameter, with the amount of 0. The \textit{Integer} type can be used for multiplication.

Inverse

<table>
<thead>
<tr>
<th>Formula</th>
<th>Property name</th>
<th>Variable (Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x + (-x) = 0$</td>
<td>additiveInverse1</td>
<td>$x : \text{Money}$</td>
</tr>
<tr>
<td>$(-x) + x = 0$</td>
<td>additiveInverse2</td>
<td>$x : \text{Money}$</td>
</tr>
</tbody>
</table>

Table 3.6: Inverse when using \textit{Money}

The inverse relation describes (for additivity) that using addition with the input parameter and the negative of the input parameter, results in the value zero. Note that the operation is used on the \textit{Money} type, so the expected value is 0 with the same currency as the currency of the input parameter. Although the inverse relation could also be used with multiplication and division (defined as $x \cdot (1/x) = 0$), it is not possible to use this definition in our case. As we cannot divide something by a \textit{Money} type, which is why we only define the inverse relation using addition.
CHAPTER 3. PROPERTIES OF REBEL

Distributivity

<table>
<thead>
<tr>
<th>Formula</th>
<th>Property name</th>
<th>Variable (Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \cdot (y + z) = x \cdot y + x \cdot z$</td>
<td>distributiveInteger1</td>
<td>$x$ : Money</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$y$ : Integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$z$ : Integer</td>
</tr>
<tr>
<td>$(y + z) \cdot x = y \cdot x + z \cdot x$</td>
<td>distributiveInteger2</td>
<td>$x$ : Integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$y$ : Money</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$z$ : Money</td>
</tr>
<tr>
<td>$x \cdot (y + z) = x \cdot y + x \cdot z$</td>
<td>distributivePercentage1</td>
<td>$x$ : Percentage</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$y$ : Money</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$z$ : Money</td>
</tr>
<tr>
<td>$(y + z) \cdot x = y \cdot x + z \cdot x$</td>
<td>distributivePercentage2</td>
<td>$x$ : Percentage</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$y$ : Money</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$z$ : Money</td>
</tr>
</tbody>
</table>

Table 3.7: Distributivity when using Money

The law of distributivity is another well-known law [24]. Unlike Associativity, the order does matter here when using different operations. These operations can be used on Money and since we can see Money as a number, this property is also expected on this type. Remember that it is not possible to multiply Money types with each other. Thus the variable types are an important part of these properties, in Table 3.7 property definitions are shown based on this property.

3.2.2 Properties of equality and inequality

The properties in this category are mainly using the equality ($=$, $\neq$) and inequality ($<$, $>$, $\leq$, $\geq$) operators in their definitions. For the property names we abbreviate these operations with the naming LT, GT, LET and GET respectively. The following property definitions belong to this category:

Property of Zero

<table>
<thead>
<tr>
<th>Formula</th>
<th>Property name</th>
<th>Variable (Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \cdot 0 = 0$</td>
<td>multiplicativeZeroProperty1</td>
<td>$x$ : Money</td>
</tr>
<tr>
<td>$0 \cdot x = 0$</td>
<td>multiplicativeZeroProperty2</td>
<td>$x$ : Money</td>
</tr>
</tbody>
</table>

Table 3.8: Property of Zero when using Money

The property of zero on multiplication states that if something is multiplied by zero, the result will always be zero. Since Rebel allows the use of multiplication on the Money type, it’s possible to multiply it by 0. The order in which this happens shouldn’t matter. Since the value of a Money variable is based on a decimal number, this property states that the value will be exactly 0 (or 0.00 in the representation of a Money value). It should not contain any decimals that would make the amount bigger than zero.
Reflexivity

<table>
<thead>
<tr>
<th>Formula</th>
<th>Property name</th>
<th>Variable (Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = x$</td>
<td>reflexiveEquality</td>
<td>$x : \text{Money}$</td>
</tr>
<tr>
<td>$x \leq x$</td>
<td>reflexiveInequalityLET</td>
<td>$x : \text{Money}$</td>
</tr>
<tr>
<td>$x \geq x$</td>
<td>reflexiveInequalityGET</td>
<td>$x : \text{Money}$</td>
</tr>
</tbody>
</table>

**Table 3.9:** Reflexivity when using *Money*

The reflexive property is defined as a relation of a type with itself [25]. An instance of type *Money* should be equal to itself. The inequality relations *smaller or equal to* and *greater or equal to* should hold too, as we can compare *Money* variables.

Symmetry

<table>
<thead>
<tr>
<th>Formula</th>
<th>Property name</th>
<th>Variable (Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = y \implies y = x$</td>
<td>symmetric</td>
<td>$x : \text{Money}$ $y : \text{Money}$</td>
</tr>
</tbody>
</table>

**Table 3.10:** Symmetry when using *Money*

Reflexivity described relations on the same variable of the *Money* type. When two different variables are used, the order should not matter and thus it should work in both ways. Which is known as the symmetric property [25].

Antisymmetry

<table>
<thead>
<tr>
<th>Formula</th>
<th>Property name</th>
<th>Variable (Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \leq y \land y \leq x \implies x = y$</td>
<td>antisymmetryLET</td>
<td>$x : \text{Money}$ $y : \text{Money}$</td>
</tr>
<tr>
<td>$x \geq y \land y \geq x \implies x = y$</td>
<td>antisymmetryGET</td>
<td>$x : \text{Money}$ $y : \text{Money}$</td>
</tr>
</tbody>
</table>

**Table 3.11:** Antisymmetry when using *Money*

The antisymmetric relation describes that whenever there is a relation from *x* to *y* and a relation from *y* to *x*, then *x* and *y* should be equal. The *lower or equal than* ($\leq$) and *greater or equal than* ($\geq$) relations fit in this category, as shown in Table 3.11. This antisymmetric relation is also expected to hold, as the *Money* type supports these operations.
CHAPTER 3. PROPERTIES OF REBEL

Transitivity

<table>
<thead>
<tr>
<th>Formula</th>
<th>Property name</th>
<th>Variable (Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = y \land y = z \implies x = z$</td>
<td>transitiveEquality</td>
<td>$x$ : Money</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$y$ : Money</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$z$ : Money</td>
</tr>
<tr>
<td>$x &lt; y \land y &lt; z \implies x &lt; z$</td>
<td>transitiveInequalityLT</td>
<td>$x$ : Money</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$y$ : Money</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$z$ : Money</td>
</tr>
<tr>
<td>$x &gt; y \land y &gt; z \implies x &gt; z$</td>
<td>transitiveInequalityGT</td>
<td>$x$ : Money</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$y$ : Money</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$z$ : Money</td>
</tr>
<tr>
<td>$x \leq y \land y \leq z \implies x \leq z$</td>
<td>transitiveInequalityLET</td>
<td>$x$ : Money</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$y$ : Money</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$z$ : Money</td>
</tr>
<tr>
<td>$x \geq y \land y \geq z \implies x \geq z$</td>
<td>transitiveInequalityGET</td>
<td>$x$ : Money</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$y$ : Money</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$z$ : Money</td>
</tr>
</tbody>
</table>

Table 3.12: Transitivity when using *Money*

Operations can be done on the *Money* types. The transitive properties [25] on the (in)equality operators should still hold on the *Money* type as we can compare the *Money* values.

Additivity

<table>
<thead>
<tr>
<th>Formula</th>
<th>Property name</th>
<th>Variable (Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = y \implies x + z = y + z$</td>
<td>additiveEquality</td>
<td>$x$ : Money</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$y$ : Money</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$z$ : Money</td>
</tr>
<tr>
<td>$x = y \land z = a \implies x + z = y + a$</td>
<td>additiveEquality4params</td>
<td>$x$ : Money</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$y$ : Money</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$z$ : Money</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$a$ : Money</td>
</tr>
</tbody>
</table>

Table 3.13: Additivity when using *Money*

Addition was earlier mentioned for the *Commutativity* and *Associativity* properties. The properties mentioned here extend these by defining properties that are true when the input values are equal. When using the addition operator such that the resulting values on both sides remain the same, as shown in Table 3.10, it should not break the equality property on the resulting values.
CHAPTER 3. PROPERTIES OF REBEL

Division

<table>
<thead>
<tr>
<th>Formula</th>
<th>Property name</th>
<th>Variable (Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \cdot y = z \implies x = z/y$</td>
<td>division1</td>
<td>$x : \text{Money}$ \hspace{1cm} $y : \text{Integer}$ \hspace{1cm} $z : \text{Money}$</td>
</tr>
<tr>
<td>$x = z \cdot y \implies x/y = z$</td>
<td>division2</td>
<td>$x : \text{Money}$ \hspace{1cm} $y : \text{Integer}$ \hspace{1cm} $z : \text{Money}$</td>
</tr>
</tbody>
</table>

Table 3.14: Division when using Money

When using division with the Money type, it is not possible to use a Money value as the denominator. A Money type can be divided by an Integer, thus we can define the division properties by using both the Money and Integer type. Note that the denominator cannot be zero, as division by zero is not possible.

Trichotomy

<table>
<thead>
<tr>
<th>Formula</th>
<th>Property name</th>
<th>Variable (Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x &lt; y \lor x = y \lor x &gt; y$</td>
<td>trichotomy</td>
<td>$x : \text{Money}$ \hspace{1cm} $y : \text{Money}$</td>
</tr>
</tbody>
</table>

Table 3.15: Trichotomy when using Money

The law of trichotomy defines that for every pair of arbitrary real numbers, exactly one of the relations $<, =, >$ holds. We can define a property for this when using Money, as shown in Table 3.15.
### 3.2.3 Additional properties of equality and inequality

For the third experiment (Chapter 7), the property definitions for division have been updated and additional properties have been added to the list of defined properties. These extra properties can be put in the category of equality and inequality properties and are using the implication (\( \Rightarrow \)) sign in its definitions.

**Division**

<table>
<thead>
<tr>
<th>Formula</th>
<th>Property name</th>
<th>Variable (Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{round}(x \cdot y) = \text{round}(z) \Rightarrow \text{round}(x) = \text{round}(z/y) )</td>
<td>divisionEquality1</td>
<td>( x : \text{Money} ) ( y : \text{Integer} ) ( z : \text{Money} )</td>
</tr>
<tr>
<td>( \text{round}(x) = \text{round}(z \cdot y) \Rightarrow \text{round}(x/y) = \text{round}(z) )</td>
<td>divisionEquality2</td>
<td>( x : \text{Money} ) ( y : \text{Integer} ) ( z : \text{Money} )</td>
</tr>
<tr>
<td>( x = y \land z \neq 0 \Rightarrow x/z = y/z )</td>
<td>divisionEquality3</td>
<td>( x : \text{Money} ) ( y : \text{Money} ) ( z : \text{Integer} )</td>
</tr>
<tr>
<td>( x &lt; y \land z &gt; 0 \Rightarrow x/z &lt; y/z )</td>
<td>divisionInequalityLT1</td>
<td>( x : \text{Money} ) ( y : \text{Money} ) ( z : \text{Integer} )</td>
</tr>
<tr>
<td>( x &lt; y \land z &lt; 0 \Rightarrow x/z &gt; y/z )</td>
<td>divisionInequalityLT2</td>
<td>( x : \text{Money} ) ( y : \text{Money} ) ( z : \text{Integer} )</td>
</tr>
<tr>
<td>( x &gt; y \land z &gt; 0 \Rightarrow x/z &gt; y/z )</td>
<td>divisionInequalityGT1</td>
<td>( x : \text{Money} ) ( y : \text{Money} ) ( z : \text{Integer} )</td>
</tr>
<tr>
<td>( x &gt; y \land z &lt; 0 \Rightarrow x/z &lt; y/z )</td>
<td>divisionInequalityGT2</td>
<td>( x : \text{Money} ) ( y : \text{Money} ) ( z : \text{Integer} )</td>
</tr>
</tbody>
</table>

Table 3.16: Updated definition of Division when using *Money*

The initial property definitions for *Division* did not take the division problem into account and thus had to be updated. The updated definitions are using the \( \text{round}() \) method to prevent the occurrence of the division problem. The \( \text{round}() \) method rounds the value to 4 decimals and uses the “HALF_UP” rounding technique, this is described in more detail in Chapter 7. In addition to this modification, more property definitions have been added for division, as shown in Table 3.16.
Additivity

<table>
<thead>
<tr>
<th>Formula</th>
<th>Property name</th>
<th>Variable (Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = y \implies x + z = y + z$</td>
<td>additiveEquality</td>
<td>$x : \text{Money}$, $y : \text{Money}$, $z : \text{Money}$</td>
</tr>
<tr>
<td>$x = y \land z = a \implies x + z = y + a$</td>
<td>additiveEquality4params</td>
<td>$x : \text{Money}$, $y : \text{Money}$, $z : \text{Money}$, $a : \text{Money}$</td>
</tr>
<tr>
<td>$x &lt; y \implies x + z &lt; y + z$</td>
<td>additiveInequalityLT</td>
<td>$x : \text{Money}$, $y : \text{Money}$, $z : \text{Money}$</td>
</tr>
<tr>
<td>$x &gt; y \implies x + z &gt; y + z$</td>
<td>additiveInequalityGT</td>
<td>$x : \text{Money}$, $y : \text{Money}$, $z : \text{Money}$</td>
</tr>
</tbody>
</table>

Table 3.17: Additivity when using Money

For additivity, additional property definitions are added that use inequality, as shown in Table 3.17. The existing ones (described in the first 2 rows) only used equality in its definitions.

Subtraction

<table>
<thead>
<tr>
<th>Formula</th>
<th>Property name</th>
<th>Variable (Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = y \implies x - z = y - z$</td>
<td>subtractiveEquality</td>
<td>$x : \text{Money}$, $y : \text{Money}$, $z : \text{Money}$</td>
</tr>
<tr>
<td>$x &lt; y \implies x - z &lt; y - z$</td>
<td>subtractiveInequalityLT</td>
<td>$x : \text{Money}$, $y : \text{Money}$, $z : \text{Money}$</td>
</tr>
<tr>
<td>$x &gt; y \implies x - z &gt; y - z$</td>
<td>subtractiveInequalityGT</td>
<td>$x : \text{Money}$, $y : \text{Money}$, $z : \text{Money}$</td>
</tr>
</tbody>
</table>

Table 3.18: Subtraction when using Money

The property definitions concerning subtraction are kind of similar to those of Additivity. It uses subtraction instead of addition in its definition. Nevertheless, to check whether the properties holds for subtraction we defined these properties as shown in Table 3.18.
CHAPTER 3. PROPERTIES OF REBELL

Multiplication

<table>
<thead>
<tr>
<th>Formula</th>
<th>Property name</th>
<th>Variable (Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = y \implies x \cdot z = y \cdot z$</td>
<td>multiplicativeEquality</td>
<td>$x: Money$, $y: Money$, $z: Integer$</td>
</tr>
<tr>
<td>$x &lt; y \land z &gt; 0 \implies x \cdot z &lt; y \cdot z$</td>
<td>multiplicativeInequalityLT1</td>
<td>$x: Money$, $y: Money$, $z: Integer$</td>
</tr>
<tr>
<td>$x &lt; y \land z &lt; 0 \implies x \cdot z &gt; y \cdot z$</td>
<td>multiplicativeInequalityLT2</td>
<td>$x: Money$, $y: Money$, $z: Integer$</td>
</tr>
<tr>
<td>$x &gt; y \land z &gt; 0 \implies x \cdot z &gt; y \cdot z$</td>
<td>multiplicativeInequalityGT1</td>
<td>$x: Money$, $y: Money$, $z: Integer$</td>
</tr>
<tr>
<td>$x &gt; y \land z &lt; 0 \implies x \cdot z &lt; y \cdot z$</td>
<td>multiplicativeInequalityGT2</td>
<td>$x: Money$, $y: Money$, $z: Integer$</td>
</tr>
</tbody>
</table>

Table 3.19: Multiplication when using Money

The property definitions for multiplication are looking similar to those of Division. The property definitions described in Table 3.19 are different in that these use the multiplication operator along with the inequality symbols.

3.3 Conclusion

In this chapter, we focused on the properties when using the Money type in Rebel. The research question lead as follows: Which properties are expected to hold on the semantics of the generated code?

Since there was no definition available of the types in Rebel, it was required to define the properties in detail. Describing what the expected behaviour is when operating with the types and which operations are allowed on each type. In this chapter, we have defined a set of properties that are expected to hold with Rebel. These property definitions will be used in the experiments to test the generator.

The properties are based on the axioms in algebra, but the requirements of Rebel have been taken into account when it comes to using certain operations with certain types. The defined properties are separated into 2 categories: “Field properties” and “Properties of equality and inequality”.

The properties that are defined in this chapter will be used to test the generator. However, it is not true that there are no bugs in the generator in case every property holds. It does mean that there were no errors found in the properties that are being checked. Many properties can be defined within the Rebel language, the properties defined in this chapter are certainly not all the properties that exist in Rebel.
3.4 Threats to validity

Lack of properties

We defined a set of properties that are expected to hold in Rebel. This can be seen as an incomplete set, as many more properties can hold and can be expected when using Rebel. The set of properties we defined are aimed to cover many operations when using the Money type, as this is considered the most important type for a bank.

We have used the axioms of algebra as inspiration for the property definitions in this chapter. We did not specify Rebel, as we have not defined all the operators and all the important properties of it. This leads to a threat of validity for the next chapters in that this affects the answer to our third research question: What kind of bugs can be found using this approach and how many? A complete list of property definitions in Rebel could result in more bugs, and thus affect the answer to this research question. For this, a profound study should be done on how Rebel works exactly, describing all the expected properties in Rebel.

Invalid definitions

We defined some properties that are expected to hold on Rebel types, specifically when using the Money type. We already discussed the lack of properties, but it could be the case that some of our definitions are invalid. We also assume that these properties do hold on Rebel types.

Some might disagree with certain properties that we have defined, which leaves this as a threat to the first research question: Which properties are expected to hold on the semantics of the generated code? But when this is the case, the properties can be updated when needed. When we encounter a bug in our experiments by using the properties in the following chapters, the designers or implementers of Rebel might disagree with the fact that it’s a bug (a false-positive). However, this issue is then getting known by using this approach, which does not mark this as a major threat.

Minimum number properties

We defined many properties in this chapter. This might lead to questions like: is this the minimal (or optimal) number of property definitions in order to detect the bugs found in the experiments? Because having a minimal (or optimal) set of properties would mean that there are less tests needed to test the same thing, resulting in that less time is required to run the test framework. The answer to this question is no, this is out of scope for this thesis. This leads to a threat to answering our third research question: What kind of bugs can be found using this approach and how many?

Some of the defined properties might be overlapping each other. It might be possible to lower the number of properties that have to be defined. However, the purpose of each property on what it tests is different. So one should be careful when determining some property unnecessary. For this thesis, we used known axioms of algebra as inspiration and defined some properties for Rebel.
Chapter 4

Test mechanics

In order to check whether the defined properties in Chapter 3 hold when using the generator, we need to determine how the test framework should work. In this chapter we will try to answer the following research question:

RQ 2: How can we test each property as automatically as possible to find bugs in the generator?

We use property-based testing as testing technique. The aim of this project is to test the implementation of the generator and trying to find, yet unknown, bugs in it. Unfortunately, we cannot test the properties right away on the generator, but we aim to test the properties as automatic as possible. To check the generator, we use the system that it generates. In order to do that, a valid Rebel specification is required. In this chapter, we describe how the test framework is setup such that it can automatically check whether the defined properties hold when using the generator.

4.1 The test framework

A Rebel specification can be created with the property definitions. Which can then be used to generate the test cases. The collection of resulting test cases is the content of the test suite, which we can be run against the generated system. We can divide this process into different phases. The goal of the test framework is to combine most of the required phases such that each defined property is being checked as automatic as possible. An overview of the phases and the test framework is shown in Figure 4.1.

![Figure 4.1: Overview of the test framework and the phases](image_url)

The phases are defined as follows:

1. Create specification
CHAPTER 4. TEST MECHANICS

2. Check & build
3. Generate system
4. Generate test suite
5. Run test suite

We will describe each phase in detail in the next sections. Additionally, we will define some evaluation criteria which will be used to evaluate the test framework. The Reflexivity property will be used to demonstrate each phase. More specifically: the case of Reflexivity when using equality, called ReflexiveEquality throughout this thesis. The definition of ReflexiveEquality is shown in Table 4.1.

<table>
<thead>
<tr>
<th>Formula</th>
<th>Variable</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = x )</td>
<td>( x )</td>
<td>Money</td>
</tr>
</tbody>
</table>

Table 4.1: Property definition of ReflexiveEquality

4.1.1 Create specification

The generator requires a consistent Rebel specification in order to generate a system. Coincidently, the Rebel syntax is strong enough to write down the defined properties in Rebel. This opens the possibility to create a Rebel specification containing the property definitions and to reuse the generator when generating the tests. This way, we can be sure that we check exactly the same code as which would be used when using a real Rebel specification. A real specification means, for example, a specific bank account specification written in Rebel. This is a convenient choice, because the generator is being checked this way and it opens the ability to use the same approach with different generators.

We can use the event blocks in Rebel to write down the properties. An event describes a transition from one state to another and accepts parameters. Additionally, it can have pre- and post conditions, where the post conditions state what happens when the transaction is being executed. In Listing 4.3 the event definition for the ReflexiveEquality property written in Rebel is shown.

```
event reflexiveEquality(x: Money) {
    postconditions {
        new this.result == ( x == x );
    }
}
```

Listing 4.3: The event definition for the ReflexiveEquality property.

The event name and the parameters are used to generate a test case from this event definition. To check whether the property was fulfilled given a certain set of parameters, we store the result in a data field called result\(^1\). The test suite uses the value of this field, to determine the result. In case the result value is false during testing, a bug has been found.

Besides the event definition, we need to write the actual Rebel specification to be able to generate a system from it. The specification describes the fields, the events it uses and the life cycle of the state machine. Since we are only interested in testing the events, we can hold the specification itself to a minimum. The life cycle consists of 2 states, the initial and final state. The transition between these states is the event we defined, ReflexiveEquality. In Listing 4.4 a specification used for one property is shown. In the case of multiple properties, we can add these to the events block. In the life cycle, we can comma separate the transitions.

\(^1\)The new keyword is used to indicate that the field should be updated when the transition takes place.
module gen.specs_money.MoneyExample
import gen.specs_money.MoneyExampleLibrary

specification MoneyExample {
  fields {
    id: Integer @key
    result: Boolean
  }
  events {
    reflexiveEquality []
  }
  lifeCycle {
    initial init -> result: reflexiveEquality
    final result
  }
}

Listing 4.4: The event definition for the ReflexiveEquality property.

4.1.2 Check & build
Now that there is a specification containing the properties, the specification can be built. This results in a CST of the specification that the test framework can use to generate the tests. This is done by using the existing toolchain that is available for Rebel.

Building the specification means that the specification is being checked and returns a CST of the specification when the specification is consistent. Building also does more, like flattening the specifications, but this is not relevant for this thesis. The CST required in order to generate a system from it by using the generator. The CST is also used by the test framework to generate the test suite.

4.1.3 Generate system
The generator will be used to generate a system from the specification that we have created. This system which will be used to check each property. Note that the generated system is assumed to be runnable. As otherwise the test suite, that will be generated by the test framework (in the next phase), cannot be run against the generated system.

4.1.4 Generate test suite
The test suite requires some configuration to work with the generated system. The test framework first initializes the test suite, then generates the test cases.

Although this generator is the most mature and often used in experiments within ING, it is still used as a prototype. The resulting system is not production ready, as this requires some more actions. One of these is that the resulting system should have tests which test the generated system. The generated system doesn’t contain anything that’s related to testing yet. So to make use of the testing libraries in Scala, we will need to add the test dependencies to the build file of the project and add a configuration file for Akka. This is done when we initialize the test suite and can be found in the source. We do not cover this in detail here since there are no custom settings in there, rather it is default configuration that is only required to make the messaging layer work for the test suite. The configuration can be found in the source of the test framework.

The test framework is build up such that it first starts the generated system, followed by running the test suite against it. Thus when running the test suite (next phase), the generated system will automatically be started.
CHAPTER 4. TEST MECHANICS

The test framework can traverse the CST and generate a test case for each event. A test case is generated by using the templating feature of Rascal, where we fill in event specific data as shown in Listing 4.5. The resulting test case of ReflexiveEquality is shown in Listing 4.6.

The functions generateParamList() and checkAction() are utility functions that are defined in the template that is used for a test file. The generateRandomParamList() method generates tuples of random values that are used as parameters. checkAction() is a method that executes the given event and checks whether the resulting value of the result field was true. A test file consists of the utility functions and all of the snippets that were generated.

4.1.5 Run test suite

The test suite can be run with SBT by using sbt test. The log shows detailed information about the tests and shows a summary when the test suite has finished. When running the test framework with the specification that we created in Subsection 4.1.1 the test suite finishes successfully, as shown in Listing 4.7.
Looking at the run time of this specific run, it shows us that the ReflexiveEquality test case was executed within 4 seconds. While running the whole test suite took almost 37 seconds. This difference is due to the fact that the generated system is being started first, as described in Subsection 4.1.4.

The log clearly shows which test cases were run and whether these failed or not. Now that we have a working case, how does this work in case of a test failed? We can simulate a bug by modifying the generator that we use. Let’s say that we have a translation error in the generator, such that the equality (==) operator would be translated to a not equal (!=) operator in the generated system. The results show a detailed stack trace of what went wrong along with the input values, such that the issue can be reproduced. Listing 4.8 shows the output after modifying the generator.

Listing 4.7: Log output of the test suite concerning ReflexiveEquality.

```
[info] MoneySpec
[info] - should work with ReflexiveEquality (3 seconds, 686 milliseconds)
[info] ScalaTest
[info] Run completed in 36 seconds, 957 milliseconds.
[info] Total number of tests run: 1
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 1, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
> Done testing
> ** Tests successful! **
```

Listing 4.8: Log output after modifying the generator.

```
[info] MoneySpec
[info] - should work with ReflexiveEquality *** FAILED *** (1 second, 278 milliseconds)
(Data(None,Some(true))), found CurrentState(Result,Initialised(Data(None,Some(false))))
: With command: ReflexiveEquality(-940003591.28 EUR)
[info] at scala.Predef$.assert(Predef.scala:170)
[info] at akka.testkit.TestKitBase$class.expectMsg_internal(TestKit.scala:388)
[info] at akka.testkit.TestKitBase$class.expectMsg(TestKit.scala:382)
[info] at MoneySpec.expectMsg(MoneySpecSpec.scala:15)
[info] at MoneySpec.checkAction(MoneySpecSpec.scala:86)
[info] at MoneySpec$anonfun$1$$anonfun$apply$mcV$sp$1$$anonfun$apply$mcV$sp$2.apply(
  MoneySpecSpec.scala:174)
[info] at MoneySpec$anonfun$1$$anonfun$apply$mcV$sp$1$$anonfun$apply$mcV$sp$2.apply(
  MoneySpecSpec.scala:173)
[info] at scala.collection.immutable.List.foreach(List.scala:381)
[info] at MoneySpec$anonfun$1$$anonfun$apply$mcV$sp$1.apply$mcV$sp$1.apply$mcV$sp$1(MoneySpecSpec.scala:172)
[info] at MoneySpec$anonfun$1$$anonfun$apply$mcV$sp$1.apply(MoneySpecSpec.scala:172)
[info] ...
[info] ScalaTest
[info] Run completed in 35 seconds, 883 milliseconds.
[info] Total number of tests run: 1
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 0, failed 1, canceled 0, ignored 0, pending 0
[info] *** 1 TEST FAILED ***
> Done testing
> ** Some tests failed! **
```
CHAPTER 4. TEST MECHANICS

4.1.6 Test framework evaluation

The tests are generated based on the defined properties. After running the test framework, we evaluate the results and check what can be improved. The evaluation is done to be sure that the properties that we have defined are being tested (automatically) by the test framework and that we are able to find bugs with this approach. We define the following criteria to evaluate the test framework after each iteration:

Coverage

The coverage metric is used to check whether the properties are actually being tested. In case something in the system is not covered by the tests, it could be that the properties were not being tested. For example, this could happen if our test framework generates an incorrect test suite in the fourth phase (Subsection 4.1.4). Checking the coverage shows that the properties are being tested by the test suite.

An open-source library called Scoverage [29] is used to determine the test coverage. It creates a detailed report about the coverage. Since the generated system uses SBT as build tool, we use the open-source plug-in sbt-scoverage\(^2\) to integrate the tool with SBT. The tool could report the test coverage incorrectly, causing it to be a threat for our evaluation. We also tried to use another tool, JaCoCo [30] but this tool didn’t determine the coverage as expected, resulting in reporting 0% coverage every time.

Mutation coverage could also be a way to evaluate the test framework. PIT is a known tool for mutation testing, but it currently cannot meaningfully mutate Scala [31]. We stick with using test coverage.

We are using random input data in our experiments, which can affect the results of the coverage reports. However, multiple runs often report the same coverage. In our experiments, we will run the test framework multiple times and take the coverage report that is equal in most of these runs. This report is then used for the evaluation.

The coverage report shows how many statements exist in the generated system and how many of those were covered. Additionally, it does the same for branches, which is the number of different execution paths that could be taken. We will use the statement coverage and the total percentage of coverage. The coverage report also shows which parts of each statement have been executed, it shows green highlighting for covered parts and red highlighting for uncovered parts. The coverage highlighting for the ReflexiveEquality property described in this chapter highlights everything green, meaning that the whole statement was executed, as shown in Figure 4.2.

![Figure 4.2: Test coverage example for ReflexiveEquality](https://github.com/scovrage/sbt-scoverage)

The logic of a specification is defined in one Class in the generated system, which is called Logic and prefixed by the Rebel specification name. We will only look at these classes to check to which extent the properties have been tested, using the highlighting that shows the coverage.

\(^2\)https://github.com/scovrage/sbt-scoverage
The generated system also contains some other logic that is more related to how it communicates with other instances when it is deployed, which is not something that is covered by the properties we defined in Chapter 3. As a result, we will not be able to bring the test coverage to 100%. However, all files in the generated system will be used to determine the overall coverage percentage. Since we use the same generated system to determine the coverage, the higher the coverage, the more complete it tests the defined properties in the generated system.

**Number of bugs**

The number of bugs found by an experiment also describes how effective the experiment was. Although this cannot be a hard criterion, as it can vary per case. Consider that the system was already tested thoroughly, such that the bugs that this test suite would have found are already solved. This would mean that the number of bugs found would remain 0, thus wouldn’t have any effect as criteria. It is still an interesting part, as the number of bugs found proofs that the test framework is able to find bugs. Because of this, we will report on this criterion and take it into account.

**4.2 Conclusion**

The research question for this chapter lead as follows:

How can we test each property as automatically as possible to find bugs in the generator?

Existing approaches often require the system under test to be written in the same language. This was not possible when testing the generator in our case. In our case, the generator is being used to generate a system, against which the test suite will run. Our approach is similar to QuickCheck in that we use random input values and test each property multiple times. It differs in that we use a Rebel specification and the generated system to check whether the properties hold when using the generator. Unlike QuickCheck, we do not use mitigation techniques to pinpoint the edge cases.

We demonstrated a full cycle based on one property, which indicated that this approach works to check a property. A full cycle consists of the following 5 phases:

1. Create specification
2. Check & build
3. Generate system
4. Generate test suite
5. Run test suite

The first step is done manually by translating the properties to a Rebel specification. The test framework is able to execute the other phases, which can be found in the `Main.rsc` file in the source code.

For the experiments, all the properties defined in Chapter 3 will be used. This results in a bigger specification, which can be used to test the generator automatically by using the test framework. After running the test framework, we evaluate it on the coverage and number of bugs found metrics. The event definitions of each property defined in Chapter 3 can be found in the Appendix A. The event definitions for the additional properties that were added in the third experiment (Chapter 7) can be found in Appendix B.

---

3Library files are not included when determining the coverage
4.3 Threats to validity

Uncompilable system

When the generated system is unable to compile, the test framework cannot proceed. Because it cannot run the generated test suite against the generated system in that case. Although such errors could be detected by the test framework, it is out of scope for this thesis. It is hard to argue whether the compilation error would be a bug or something else, as it can have many causes.

When running the test framework, it can still occur that the generated system is unable to compile. We will see this back in Chapter 5. The test framework terminates when the generated system cannot compile. Affecting the bugs that we could find to answer the third research question: What kind of bugs can be found using this approach and how many?

Random input

A threat to our approach is that we are probably not generating enough random input to detect all bugs. This affects the amount of bugs we find in each experiment as well as the kind of bugs that we might find to answer the third research question: What kind of bugs can be found using this approach and how many? Also, in case we do not find any bugs, it could be that there are no bugs in the system anymore. Or that we do not generate enough random input to detect additional bugs. Or that our set of properties are not complete, as described in Chapter 3. Since we do only check for the properties we have defined, the first option is not the case. This approach does not find all the bugs in the system, but only checks the system with the properties that we have defined.

One system

Only one generator is being used throughout this thesis. However, it would be useful to make the test framework compatible with the other generators and generated systems too. This enables reasoning about the different implementations and its generators. Some changes are required to make the test framework compatible with these systems. By doing so, every generated system for which a generator is built by ING can be checked based on the same properties, resulting in that the defined properties are checked thoroughly on every system and that inequalities can be detected between the different generators. Testing multiple generators can result in finding more bugs, which can affect our results to the third research question: What kind of bugs can be found using this approach and how many?

A threat in doing so is that one of the other generators might not support some translations of each expression that is used in the specification that we created. Thus the test framework can also be used to check whether every expression variant is taken into account by the generator. Unfortunately, an error in this translation would be blocking, in that it can lead to a generated system that is not able to compile. Resulting in that the test framework cannot proceed to run the test suite on the generated system. This could be used as a way to check the generators too. Although compilation errors were not the aim of the project, as compilation errors can have many causes, the test framework can still be used to detect those to a certain extent.

Whitebox implementation

We use property-based testing as testing technique and implemented the required functionality ourselves, resulting in a white-box implementation. This means that we expect that our values generation is working correctly too. In case this isn’t working correctly, a fix is required. This is a threat to our second research question: How can we test each property as automatically as possible to find bugs in the generator? Because this would mean that a property cannot be tested automatically and that this can be improved (by fixing it). In turn, it can also affect the bugs that we could find to answer the third research question.

Another way how this could be done was to check how the custom types were generated to Scala. And
then generate a Scala test project using the same types. Writing property tests for each type could achieve the same goal when it comes to checking the implementation of this component in the generated system. However, if we would follow this approach, we wouldn’t use the generator to translate the Rebel expressions to Scala. This results in that the generator itself is still not being tested. With our approach, we test the generator and are able to find errors in the generator. Although we cannot conclude that the generator is implemented correctly when the generated test suite runs successfully, rather we can conclude that the properties it checks for are satisfied.
Chapter 5

Experiment 1: Using random input

The properties that we defined in Chapter 3 are translated into test cases as described in Chapter 4. In this experiment, we expect to find some bugs that were unknown before by using the test framework. When we have triggered some bugs, an investigation is needed to check what the cause is of that bug. Next, we can categorize the bugs found to partially answer the third research question: What kind of bugs can be found using this approach and how many?

5.1 Method

In the first experiment, each property will be tested 100 times with random input values. This means that if the property holds for 100 tests, it is reported to be successfully satisfying the property. This is a similar approach compared to what QuickCheck does when checking properties. Unlike QuickCheck, the test framework does not shrink the input values to come with minimum values for which the case fails. Instead, it will just report the values that were used when the property failed.

5.2 Results

Two runs are being done to detect bugs in the generator in this experiment because the first run terminated quickly. The test framework was unable to proceed in testing every property in this run.

5.2.1 First run

The first run results into a termination of the run due to a compile error in the generated system. Although we made the assumption that the generated system should be compilable, this error came from a property definition that was expected to hold, namely AssociativeMultiplicationInteger1. Which is why we can consider this as an error that is found when using the test framework. The error describes that an overloaded method cannot be applied to the Money type, as shown in Listing 5.9.
CHAPTER 5. EXPERIMENT 1: USING RANDOM INPUT

The error log does not clearly indicate what exactly went wrong. It doesn't show clearly which property is causing this error. Also, it does not describe what the types of the variables were. Investigating the generated system reveals that both errors were happening when dealing with the AssociativeMultiplicationInteger1 property. This means that the variables $x$, $y$ and $z$ are of type Integer, Integer, Money respectively. Temporarily disabling this property allows the test framework to proceed further.

5.2.2 Second run

After disabling the AssociativeMultiplicationInteger1 property, the test framework was able to run completely. This results in 7 failing tests. For each test, the input values for which the property doesn’t hold are logged such that the error can be reproduced. In Table 5.1 an overview of the failing properties, along with its input values ($x$, $y$ and $z$) are shown.
CHAPTER 5. EXPERIMENT 1: USING RANDOM INPUT

Table 5.1: Overview of failing tests along with its input values

<table>
<thead>
<tr>
<th>Property name</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>DistributivePercentage1</td>
<td>0.51</td>
<td>-311254801.77 EUR</td>
<td>-707194075.77 EUR</td>
</tr>
<tr>
<td>DistributivePercentage2</td>
<td>0.93</td>
<td>2089630160.75 EUR</td>
<td>-1316628389.49 EUR</td>
</tr>
<tr>
<td>DistributiveInteger2</td>
<td>-883022216</td>
<td>-298435082.93 EUR</td>
<td>715725888.96 EUR</td>
</tr>
<tr>
<td>AssociativeMultiplicationPercentage2</td>
<td>840296462</td>
<td>1771903729.60 EUR</td>
<td>0.53</td>
</tr>
<tr>
<td>DistributiveInteger1</td>
<td>-1790274467.41 EUR</td>
<td>1691684272</td>
<td>1449321647</td>
</tr>
<tr>
<td>AssociativeMultiplicationInteger2</td>
<td>-1852801029.34 EUR</td>
<td>-1309504561</td>
<td>1880170895</td>
</tr>
<tr>
<td>AssociativeMultiplicationPercentage1</td>
<td>-352883323.42 EUR</td>
<td>0.27</td>
<td>294211708</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0.51</td>
<td>Percentage</td>
</tr>
<tr>
<td>y</td>
<td>-311254801.77 EUR</td>
<td>Money</td>
</tr>
<tr>
<td>z</td>
<td>-707194075.77 EUR</td>
<td>Money</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Formula</th>
<th>Scala result</th>
<th>Expected result</th>
</tr>
</thead>
<tbody>
<tr>
<td>x*(y+z) == (y<em>x)+(z</em>x)</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>x*(y+z)</td>
<td>-519408927.54539996 EUR</td>
<td>-519408927.5454 EUR</td>
</tr>
<tr>
<td>(y<em>x)+(z</em>x)</td>
<td>-519408927.5454 EUR</td>
<td>-519408927.5454 EUR</td>
</tr>
<tr>
<td>y+z</td>
<td>-1018448877.54 EUR</td>
<td>-1018448877.54 EUR</td>
</tr>
<tr>
<td>y*x</td>
<td>-158739948.9027 EUR</td>
<td>-158739948.9027 EUR</td>
</tr>
<tr>
<td>z*x</td>
<td>-360668978.6427 EUR</td>
<td>-360668978.6427 EUR</td>
</tr>
</tbody>
</table>

Table 5.2: DistributivePercentage1: Precision error when multiplying a Percentage with Money

5.3 Analysis

For each failed test we investigate what went wrong. The first four tests reveal precision problems when using the Money type in calculations. The latter three tests were also failing because of these precision problems. However, these tests were also failing after the precision errors were fixed. Thus, for the latter 3 tests, another version of the generated system was used, which contains the fixes for the precision problems. This is done such that we are able to reveal the other errors that these properties can reveal.

DistributivePercentage1

This property uses a Percentage value and two Money values for its tests. The values are named x, y and z respectively. To check this failing test, we check the results of the intermediate calculations in the formula that is being used. In Table 5.2 the values are shown for which the test case failed, along with the intermediate calculations. The intermediate calculations seem to be fine, as the results are almost the same when we compare the results of the Scala evaluation and the expected result. The resulting left-hand side of the expression contains a precision error, which is caused when multiplying a Percentage (the x variable) with a Money type (the result of y+z in this case).
CHAPTER 5. EXPERIMENT 1: USING RANDOM INPUT

**DistributivePercentage2**

This test case looks similar as **DistributivePercentage1**. It uses the same type of variables, but the expression is slightly different. In Table 5.3 the result and the intermediate calculations of a failing case are shown. What can be seen here is that the precision error occurs when the Money type is multiplied by the Percentage type. While with **DistributivePercentage1** it was the other way around.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0.93</td>
<td>Percentage</td>
</tr>
<tr>
<td>y</td>
<td>2089630160.75 EUR</td>
<td>Money</td>
</tr>
<tr>
<td>z</td>
<td>-1316628389.49 EUR</td>
<td>Money</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Formula</th>
<th>Scala result</th>
<th>Expected result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(y+z)<em>x == (y</em>x)+(z*x)</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>(y+z)*x</td>
<td>718891647.2718 EUR</td>
<td>718891647.2718 EUR</td>
</tr>
<tr>
<td>(y<em>x)+(z</em>x)</td>
<td>718891647.2718001 EUR</td>
<td>718891647.2718 EUR</td>
</tr>
<tr>
<td>y+z</td>
<td>773001771.26 EUR</td>
<td>773001771.26 EUR</td>
</tr>
<tr>
<td>y*x</td>
<td>1943356049.4975002 EUR</td>
<td>1943356049.4975 EUR</td>
</tr>
<tr>
<td>z*x</td>
<td>-1224464402.2257001 EUR</td>
<td>-1224464402.2257 EUR</td>
</tr>
</tbody>
</table>

Table 5.3: **DistributivePercentage1**: Precision error when multiplying a Money with Percentage

**DistributiveInteger2**

This case uses **Integer** in conjunction with the Money type. Earlier cases showed that there was a precision error when using the Percentage and Money types. Since the Percentage type is translated to a Double in the generated system, it can be expected that there would be precision problems occurring. As this is a known issue with types that use floating-point arithmetic [27]. This case reveals that a precision error also occurs when multiplying Money with an Integer. In the intermediate calculations when investigating a failing test with its values are shown in Table 5.4. The last two rows, in boldface, show that a precision error occurs when Money is multiplied by an Integer.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>-883022216</td>
<td>Integer</td>
</tr>
<tr>
<td>y</td>
<td>-298435082.93 EUR</td>
<td>Money</td>
</tr>
<tr>
<td>z</td>
<td>715725888.96 EUR</td>
<td>Money</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Formula</th>
<th>Scala result</th>
<th>Expected result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(y+z)<em>x == y</em>x + z*x</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>(y+z)*x</td>
<td>-368477052257036762.48 EUR</td>
<td>-368477052257036762.48 EUR</td>
</tr>
<tr>
<td>y<em>x + z</em>x</td>
<td>-368477052257036796 EUR</td>
<td>-368477052257036796 EUR</td>
</tr>
<tr>
<td>y+z</td>
<td>417290806.03 EUR</td>
<td>417290806.03 EUR</td>
</tr>
<tr>
<td>y*x</td>
<td>263524808260992384 EUR</td>
<td>263524808260992372.88 EUR</td>
</tr>
<tr>
<td>z*x</td>
<td>-632001860518029180 EUR</td>
<td>-632001860518029135.36 EUR</td>
</tr>
</tbody>
</table>

Table 5.4: **DistributiveInteger2**: Precision error when multiplying Money with an Integer
**CHAPTER 5. EXPERIMENT 1: USING RANDOM INPUT**

**AssociativeMultiplicationPercentage2**

The earlier cases already shown a precision error when using *Double* and *Integer* in conjunction with *Money*. This case triggers the same problem, but also reveals that the same thing happens when multiplying an *Integer* with *Money*. While with *DistributiveInteger2* it was the other way around. The intermediate calculations are shown in Table 5.5, the calculation of multiplying an *Integer* with *Money* is shown in boldface. Additionally, this case shows that the small precision errors can cause a noticeable difference, which lead to a difference of 130 EUR in this case (in the *Scala* results).

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>840296462</td>
<td>Integer</td>
</tr>
<tr>
<td>y</td>
<td>1771903729.60 EUR</td>
<td>Money</td>
</tr>
<tr>
<td>z</td>
<td>0.53</td>
<td>Percentage</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Formula</th>
<th>Scala result</th>
<th>Expected result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x<em>y)<em>z == x</em>(y</em>z)</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>(x*y)*z</td>
<td>789129950543366910 EUR</td>
<td>789129950543366877.856 EUR</td>
</tr>
<tr>
<td>x*(y*z)</td>
<td>789129950543366780 EUR</td>
<td>789129950543366877.856 EUR</td>
</tr>
<tr>
<td>x*y</td>
<td>1488924434987484670 EUR</td>
<td>1488924434987484675.2 EUR</td>
</tr>
<tr>
<td>y*z</td>
<td>939108976.688 EUR</td>
<td>939108976.688 EUR</td>
</tr>
</tbody>
</table>

Table 5.5: AssociativeMultiplicationPercentage2: Precision error causing bigger differences

**DistributiveInteger1**

This case also uses three variables: *x*, *y* and *z*. Which are of type *Money*, *Integer* and *Integer* respectively. In Table 5.6 the different values are shown of the calculation between *Scala* and the expected result. In boldface, it shows how the addition of two (positive) integers results in a negative value. This is because the resulting value of the addition would be bigger than the maximum value of what an *Integer* type can hold. This results occurrence is called integer overflow [32]. The operation that is being done does not check or prevent this overflowing behaviour. Although this could be expected when using the *Integer* type, it can lead other errors classes of vulnerabilities, including stack and heap overflows [33], when this is not being prevented.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>-1790274467.41 EUR</td>
<td>Money</td>
</tr>
<tr>
<td>y</td>
<td>1691684272</td>
<td>Integer</td>
</tr>
<tr>
<td>z</td>
<td>1449321647</td>
<td>Integer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Formula</th>
<th>Scala result</th>
<th>Expected result</th>
</tr>
</thead>
<tbody>
<tr>
<td>x*(y+z) == (x<em>y)+(x</em>z)</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>x*(y+z)</td>
<td>20635097589620385223.57 EUR</td>
<td>-5623262698769382599.79 EUR</td>
</tr>
<tr>
<td>(x<em>y)+(x</em>z)</td>
<td>-5623262698769382599.79 EUR</td>
<td>-5623262698769382599.79 EUR</td>
</tr>
<tr>
<td>y+z</td>
<td>-1153961377</td>
<td>3141005910</td>
</tr>
<tr>
<td>x*y</td>
<td>-302857915908673575.52 EUR</td>
<td>-302857915908673575.52 EUR</td>
</tr>
<tr>
<td>x*z</td>
<td>-2594683539688709024.27 EUR</td>
<td>-2594683539688709024.27 EUR</td>
</tr>
</tbody>
</table>

Table 5.6: DistributiveInteger1: *Integer* overflows when using addition
CHAPTER 5. EXPERIMENT 1: USING RANDOM INPUT

**AssociativeMultiplicationInteger2**

For this case three variables are used: $x$, $y$ and $z$, which are of type *Money*, *Integer* and *Integer* respectively. In Table 5.7 the values of a failing test case are shown, along with the intermediate formula steps. On the left-hand side of the expression, we see the expected results, while on the right-hand side there is a big difference between the *Scala* result and the expected result (shown in boldface in Table 5.7). The result value of the operation would become smaller than the minimum value of what an *Integer* value can be. When this happens, an integer underflow occurs which results in the value being “wrapped” to the maximum value \[^{[32]}\], and then being used to calculate further. This underflow results in an unexpected amount, which we can see back in the results. The operation neither checks for underflowing an *Integer* value nor does it prevent it.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>-1852801029.34 EUR</td>
<td>Money</td>
</tr>
<tr>
<td>$y$</td>
<td>-1309504561</td>
<td>Integer</td>
</tr>
<tr>
<td>$z$</td>
<td>1880170895</td>
<td>Integer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Formula</th>
<th>Scala result</th>
<th>Expected result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(x\times y)\times z = x\times (y\times z)$</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>$(x\times y)\times z$</td>
<td>4561767263499657218201... EUR(^1)</td>
<td>4561767263499657218201... EUR</td>
</tr>
<tr>
<td>$x\times (y\times z)$</td>
<td>3877739486117270379.94 EUR</td>
<td>4561767263499657218201769467.30 EUR</td>
</tr>
</tbody>
</table>

| $x\times y$         | 2426251398546224819.74 EUR | 2426251398546224819.74 EUR |
| $y\times z$         | -2092906591               | -2462092362461952095     |

Table 5.7: *AssociativeMultiplicationInteger2*: *Integer* underflows when using multiply

**AssociativeMultiplicationPercentage1**

In this case there are three variables: $x$, $y$ and $z$, which are of type *Money*, *Percentage* and *Integer* respectively. In Table 5.8 the values and intermediate calculations are shown of a failing case, such that we can reason about the results. The row in boldface shows a precision error when comparing the results of *Scala* and the expected result with each other. This issue is caused by the *Percentage* that is being used. In the implementation, the *Percentage* is being translated into a *Double* value. In this case, this *Double* value is then being multiplied with an *Integer*. This results in a *Double* value containing a precision error, which is related to the problems with floating-point arithmetic \[^{[27]}\].

\(^1\)This value has been truncated for readability. The exact value is “4561767263499657218201769467.30 EUR”. The same counts for the “Expected result” column.
### 5.4 Evaluation criteria

When looking at the coverage results, it is notable that the condition for the if-clause of the implicative properties is often not being triggered. Resulting in that it always returns `true`, as this is how it was specified in the specification (the else-clause of an implicative property). This is caused by the random values that are being used as input. An example of this is the TransitiveEquality property \((x = y \land y = z \implies x = z)\). When relying on random data, there is a seldom chance that 3 values are equal to each other. Thus we could optimize the random values such that the condition holds, such that we also test these properties such that the if-clause is triggered. In Figure 5.1 the coverage for the TransitiveEquality property, green highlighting indicates the statements that are executed, while red highlighting indicates statements that were not checked at all.

```scala
case TransitiveEquality(x, y, z) => {
    checkPostCondition((nextData.get.result.get -- {
        if ((x == y && y == z)) x == z
        else true
    }))
    "new this.result == ((x == y && y == z) ? x == z : True )"
}
```

**Figure 5.1:** Test coverage of an implicative property (TransitiveEquality)

The first criteria to evaluate an experiment was to determine the test coverage. The implicative properties are not covered when using random values as input data. The other properties, which do not use implication, are fully tested though. In Figure 5.2 the results of the coverage report are shown, with a total of 87.80% coverage. The file with its name ending with “Logic” contain the...
implementation of the properties. The other 12.20% code is not tested, but this part is also not related to the implementation of the properties. Because of this, it is not required to achieve the full 100% coverage. Note that the coverage does not include the coverage over the libraries that the system uses. Some implementation details are depended on the libraries that the generated system use, thus it can not be concluded that 87.80% of the system is now tested.

![Figure 5.2: Test coverage report of the first experiment](image)

The branch coverage is reported to be 66.67%. This is caused by the implicative properties that are being translated to *if-else* statements, where the *else* branch is translated to *true*. Furthermore the coverage could be increased as not every part of the condition of implicative properties are being checked, as we have seen in Figure 5.1.

The second criterion that we defined is the number of bugs that we have found by doing the experiment. By using this approach we found a total of 8 bugs. A compilation error (1x), overflow/underflow errors (2x) and precision errors (5x). Many of these precision errors originated from the library that is used for the *Money* type, for which we created an issue on Github\(^2\). An improvement to the test framework, such that the implicative properties are also covered, might result in more bugs that can be found.

### 5.5 Conclusion

In this first experiment, we tested each property that was defined in Chapter 3 100 times with using random values as input. First, the test suite terminated due to a compilation error. After disabling the causing property (temporarily), a total of 7 tests were failing. In this experiment, we managed to find precision errors and overflow/underflow errors. Additionally, we found a compilation error when using a property which was expected to hold. The precision errors that were related to the *Money* type were reported. These bugs are fixed in the next version of the generator.

Although many properties were tested successfully, the test framework also indicates that the implicative properties were satisfied. However, when looking at the statement coverage of the implicative properties, we saw that the if-clause is often not being triggered. Meaning that it would call the else-clause which simply returns true. When relying on random data, there is a seldom chance that the if-clause is being triggered. Thus we could optimize the generated input values such that these satisfy the condition for the if-clause.

### 5.6 Threats to validity

**Fixed amount of tries**

The 100 tries to check a property is a fixed number that is being used, but why exactly this number and not a higher or lower number? It might be the case that some errors are not triggered because of this fixed amount. Running more cases might be revealing an additional error, or it might not.

\(^2\)https://github.com/typelevel/squants/issues/265
In case it doesn’t, it means that the test framework just requires more time to run the whole test suite, while it does not have an effect on the results. 100 seems to be an amount that works such that it consistently reports the same amount of failing tests however, this has checked by running it the test runs multiple times, also with using numbers like 300 or 50 for the amount. Also QuickCheck uses this amount to check a property. During this thesis, we stick with 100 as the number of tries. Finding the optimal number of tries is left as future work, thus remaining as a threat to validity in this approach.

The optimal amount of tries would be a useful addition when answering our second research question: How can we test each property as automatically as possible to find bugs in the generator? Although we focus on automatically testing the properties, not mentioning the efficiency. Increasing the efficiency of the test framework would result in less time required to run it. Also, adding more properties would increase the time required to run the test suite. It might also be the case that increasing the number of tries would result in finding additional bugs. Thus leading to a threat to answering the third research question: What kind of bugs can be found using this approach and how many?

Amount of test runs

The test framework has been executed several times for this experiment. With each run, 7 tests were failing, which consistently were the same tests on every run. Because of this, we assumed that the causes of these failures were the same, as the same tests failed on each run. However, the causes might not have been the same for these runs, thus remaining as a threat to validity to answering our third research question. Because the causes of the failing tests could be different and might reveal additional bugs.

Unfixed issue

Unfortunately, the compilation error has not been fixed throughout this project however, it is an open issue on Github\(^3\). The property causing this issue is disabled in order to continue running the test framework. This prevents the test framework from detecting bugs that could be found by this property.

---

\(^3\)https://github.com/typelevel/squants/issues/281
Chapter 6

Experiment 2: Improving coverage

Some bugs were found by using random input values in the first experiment. However, the implicative properties were not effectively checked in terms of triggering the if-clause when using random input values. This is what we aim to improve in this experiment, expecting to detect more bugs.

6.1 Method

We can separate the properties that we have in 2 different categories: those using implication ( \( \Rightarrow \) ) and those that do not. The defined properties are being separated over two specifications according to which category these belong. We name these specifications MoneyExpressions and MoneyConditionals. For the MoneyConditionals specification (the implicative properties), another way of generating the random values would be more useful. The random input values will be optimized such that the condition of the if-clause of these properties are being satisfied. For the other specification (Money-Expressions), the earlier approach (random values as input) can still be used. We do not need to change this functionality for these cases since the properties can be used with random values.

In the MoneyConditionals specification, the condition to trigger the if-clauses will be added to the preconditions of each event definition in Rebel, such that these can be used to generate the values matching this clause. The updated event definition of the Symmetric property is shown in Listing 6.10 for example. Where the preconditions have been added to the event definition.

```java
1 event symmetric(x: Money, y:Money) {
2   preconditions {
3     x == y;
4   }
5   postconditions {
6     new this.result == ( (x == y) ? y == x : False );
7   }
8 }
```

Listing 6.10: The updated event definition of the Symmetric property

When generating the test suite, the events are being traversed. In case an event with some preconditions is found, it generates a list of tuples containing the values that satisfy the condition to trigger the if-clause. Which is different compared to the generated tests in Chapter 5. The size of the tuples depends on the arity of the event from which the test is being generated.

The first difference is that it now uses our custom generator to determine the input values, instead of the built-in random generator in Java. A list of tuples, containing values which satisfy the if-clause of the implication, is being generated. Our custom generator is a simple proof of concept in order to check if this will actually result in more failing tests. This custom generator basically consists of multiple methods which are being called based on the event name. In Listing 6.11 this behaviour
is shown for the Symmetric and Division1 event. The String parameter of these methods is a way how we can pattern match on the event name in Rascal. In case the event couldn’t be handled, an exception is thrown.

```
private list[Expr] genTestValueForEvent("Symmetric") {
  Expr moneyValue = genRandomMoney();
  return [moneyValue, moneyValue];
}

private list[Expr] genTestValueForEvent("Division1") {
  real moneyAmountX = genRandomDouble();
  real intAmountY = genRandomInteger();
  real moneyAmountZ = moneyAmountX * intAmountY;
  str currency = genRandomCurrency();
  return [convertToMoney(currency, moneyAmountX), convertToExpr(intAmountY), convertToMoney(currency, moneyAmountZ)];
}

private default list[Expr] genTestValueForEvent(str eventName) {
  throw "genTestValueForEvent not implemented for event <eventName>";
}
```

Listing 6.11: Values generation for Symmetric and Division1, including the fall-back case.

This means that the way how we determine these values is basically hard-coded, requiring to have knowledge about the if-clause itself. Note that this doesn’t make this approach very dynamic, but the result will consist of a list of tuples that satisfy the if-clause. These tuples will be used as input for the test case that will be generated.

However, the input values that are generated now are fixed when we use them directly in a test case, which completely removes the randomness of the input values when running the test suite multiple times. It would be better to keep the randomness, such that the input values are different on each run. To solve this problem, we mutate the values in the list with a random operation, such that the values are sort of random again. The tuples still have to satisfy the condition to trigger the if-clause, as this was the actual intention. So the second difference compared to the first experiment, is that for each tuple in the list, we will generate a random operation and use that operation to mutate the values inside the tuple. To ensure that the tuple values still satisfy the condition of the if-clause, each value in the tuple will be mutated by the same operation. In Listing 6.12 an example of a generated test case is shown.
"work with Antisymmetry" in {
    Seq((USD(1593.62), USD(1593.62)), (USD(2869.78), USD(2869.78)),
        (EUR(4676.80), EUR(4676.80)), (USD(1850.29), USD(1850.29)),
    // ... // More values in the list
        (USD(9501.16), USD(9501.16)), (- EUR(149.67), - EUR(149.67)),
        (- EUR(159.67), - EUR(159.67)), (EUR(8015.77), EUR(8015.77))
}.foreach {
    data: (Money, Money) => {
        val randomOperation = genRandomOperation(genRandomOperator("Money", true),
            generateRandomMoney(data._1.currency), generateRandomInteger(true),
            generateRandomInteger(false), generateRandomPercentage(true),
            generateRandomPercentage(false), Random.nextInt(10))

        checkAction(Symmetry(
            randomOperation(data._1),
            randomOperation(data._2)
        )
    }
}

Listing 6.12: Resulting test case with semi-random values. Omitted some input tuples for readability.

The list of values are generated by using our custom generator, the number of tuples in the list can be defined when generating the test suit. A method `genRandomOperation()` has been added to the template, which is used to mutate the fixed values in the list. After all the `checkAction()` method is being called to check the result.

Now that the input values for the implication events should always satisfy the condition of the if-clause, we can also update the specification such that the else-clause of the expression always returns `False`. This can be seen in Listing 6.10. This results in a failing case again in case the precondition was not met. When this happens, it indicates that there's a problem with either our value generator or with the generator itself.

### 6.2 Results

Running the test framework with these changes results in 2 additional failing tests compared to the first experiment (Chapter 5). An overview of the failing properties and the used input values are shown in Table 6.1. The log of the test run reports that the precondition was not met when using these input values, as shown in Listing 6.13.

<table>
<thead>
<tr>
<th>Property name</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Division1</td>
<td>-16729.90 USD</td>
<td>830</td>
<td>-20.16</td>
</tr>
<tr>
<td>Division2</td>
<td>-44.68 USD</td>
<td>870</td>
<td>-38870.47</td>
</tr>
</tbody>
</table>

Table 6.1: Failing tests overview along with its input values
CHAPTER 6. EXPERIMENT 2: IMPROVING COVERAGE

[info] MoneyConditionals
[info] - should work with Additive4params (7 seconds, 224 milliseconds)
[info] - should work with AntisymmetryLET (5 seconds, 493 milliseconds)
[info] - should work with Symmetric (5 seconds, 344 milliseconds)
[info] - should work with Division2 *** FAILED *** (23 milliseconds)
[info] java.lang.AssertionError: assertion failed: expected CommandSuccess(Division2(-16729.90 USD,830,-20.16 USD)), found CommandFailed(NonEmptyList(PreConditionFailed(x == z*y)))
[info] - should work with Division1 *** FAILED *** (127 milliseconds)
[info] java.lang.AssertionError: assertion failed: expected CommandSuccess(Division1(-44.68 USD,870,-38870.47 USD)), found CommandFailed(NonEmptyList(PreConditionFailed(x*y == z)))

// ...

Listing 6.13: Precondition failed error in Division1 and Division2.

6.3 Analysis

The values used in the test case should be correct since we generated these values such that they satisfy the condition of the if-clause and thus they should satisfy the preconditions. Note that the conditions of the if-clause were added as preconditions in the MoneyConditionals specification, which causes the error. As the PreConditionFailed error is thrown by the system when the input values do not satisfy the preconditions.

For Division1 it states that the condition \( x*y == z \) failed. The values used for \( x, y \) and \( z \) were -44.68 USD, 870 and -38870.47 USD respectively. The result of \( x*y = -38871.60 \) USD. This should be equal to \( z \). In fact, the input of \( z \) was slightly different, -38870.47 USD.

Remember that the input values are being mutated by a random operation that we have added to the test cases. This difference is caused by the precision error when operating with the Money type, which was found in Chapter 5. The random operation that was done was causing this behaviour. The same goes for the error with Division2, where \( x == z*y \) should hold. The values of \( x, y \) and \( z \) are -16729.90 USD, 830, -20.16 USD respectively. The result of \( z*y = -16732.80 \) USD, which is not equal to the expected value -16729.90 USD (the value of \( x \)).

The first experiment already described the precision problem and how it could be fixed. To solve this problem, we modify the generator such that the precision error is fixed when generating the system. Then the test framework is being executed again to check whether both tests are succeeding. This resulted in the same amount of tests that were failing, which means that we found a different case now. In Table 6.2 an overview of the used input values are shown\(^1\). The log reported that one case still fails on the precondition check, while the other case just reports values for which the result is false, as shown in Listing 6.14.

<table>
<thead>
<tr>
<th>Property name</th>
<th>( x )</th>
<th>( y )</th>
<th>( z )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Division1</td>
<td>1.5043478260... USD</td>
<td>-779</td>
<td>-1171.8869565217... USD</td>
</tr>
<tr>
<td>Division2</td>
<td>-3328.8254545454... USD</td>
<td>-129</td>
<td>25.8048484848... USD</td>
</tr>
</tbody>
</table>

Table 6.2: Failing tests overview, after fixing precision errors

\(^1\)The decimals have been truncated for readability, Listing 6.14 shows the exact values.
CHAPTER 6. EXPERIMENT 2: IMPROVING COVERAGE

| [info] MoneyConditionalsSpec: |
| [info] MoneyConditionals |
| [info] - should work with Additive4params (7 seconds, 24 milliseconds) |
| [info] - should work with AntisymmetryLET (3 seconds, 66 milliseconds) |
| [info] - should work with Symmetric (4 seconds, 361 milliseconds) |
| [info] - should work with Division2 *** FAILED *** (670 milliseconds) |

```java
java.lang.AssertionError: assertion failed: expected CommandSuccess(Division2
(-3328.825454545454545454545454545455 USD,-129,25.80484848484848484848484848484848 USD))
, found CommandFailed(NonEmptyList(PreConditionFailed(x == z*y)))))
```

| [info] - should work with Division1 *** FAILED *** (316 milliseconds) |

```java
java.lang.AssertionError: assertion failed: expected CurrentState(Result,Initialised
(Data(None,Some(true)))), found CurrentState(Result,Initialised(Data(None,Some(false))))
: With command: Division1(1.504347826086956521739130434782609 USD
,-779,-1171.886956521739130434782608695652 USD)
```

// ...

Listing 6.14: Precondition failed error in Division1 and Division2.

The test concerning Division2 shows that the precondition check fails. If we look at the input values, it can be seen that the Money values are a fractional number. As it contains many decimals and it rounds up at the end. When operating with this rounded value, the resulting value is also slightly different. As the generated system is implemented such that the preconditions are being checked first, the PreConditionFailed exception is thrown. This leads to the issue of the division problem in which a number cannot be equally divided. When defining the properties in Chapter 3, we did not take this into account. Instead, we defined that it should contain the exact value, which cannot be done in this case. This should be defined in order to determine whether this is expected or incorrect.

When looking at Division1, we see another case as the input values passed the precondition checks. This indicates that the values satisfy the condition to trigger the if-clause of the property. However, the result of the if-clause returns false, showing us that the property does not hold when using these input values. Thus a case has been found for which the Division1 property doesn’t hold. The investigation of the intermediate calculation steps are shown in Table 6.3. Note that the Division1 property is defined as \( x \cdot y = z \implies x = \frac{z}{y} \).

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>1.504347826086956521739130434782609 USD</td>
<td>Money</td>
</tr>
<tr>
<td>Y</td>
<td>-779</td>
<td>Integer</td>
</tr>
<tr>
<td>Z</td>
<td>-1171.886956521739130434782608695652 USD</td>
<td>Money</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Formula</th>
<th>Scala result</th>
<th>Expected result</th>
</tr>
</thead>
<tbody>
<tr>
<td>x*y == z</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>x == z/y</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

x*y = -1171.886956521739130434782608695652411 USD
z/y = 1.504347826086956521739130434782608 USD

Table 6.3: Division1: Difference in rounding

In the results, we can see that the expected values do not match the property either. Although in Scala the first expression is considered true. Since the expected results also return false for the intermediate calculations, the input values might not fully satisfy the condition to trigger the if-clause. Which could be an implementation error in our values generator. However, it’s notable that in Scala the condition was first considered to hold since it passed the precondition checks. This indicates that
there is a rounding error happening in the system, which triggered this case.

Unfortunately, we are unable to trace back how the input values used for this tests were exactly determined. As these are build up by using randomly generated values and then mutating these by a random operation (as described in Section 6.1). Nevertheless, the results show that there is also an unexpected rounding going on when executing $x \times y$ in Scala. As the expected value contains some additional decimals compared to the result from Scala.

6.4 Evaluation criteria

To evaluate this experiment, we use the version of the generator in which the precision problems related to the Money type are fixed. This is done because otherwise the precision errors would cause more tests to fail. Additionally this would cause that the rounding errors found Section 6.3 would not be triggered.

When looking at the coverage report concerning a specific implicative property, it can be seen that the else-clause of the implication is not being triggered anymore. This was also the intention of the modification done in this experiment, as the if-clause is actually what we wanted to check in this experiment. In Figure 6.1 the coverage highlighting of TransitiveEquality is shown.

```scala
case TransitiveEquality(x, y, z) => {
    CheckPostCondition((nextData.getS supposedResult) get == {
        if ((x == y && y == z)) x == z
        else false
    })), "new this.result == ( (x == y && y == z) ? x == z : false )"
}
```

**Figure 6.1:** Test coverage for TransitiveEquality in second experiment

The expectation was that the test framework could be improved, such that the test coverage on the generated system would become higher. In the first experiment we found that the implicative properties were not tested thoroughly. This is what has been improved in this experiment. The coverage report does not indicate a huge difference compared to the first experiment (Chapter 5). The total test coverage is 88.87%, which is slightly higher (1.07%) than the results in the first experiment. The report is shown in Figure 6.2. Note that the properties have been categorized in this experiment, this is also visible in the coverage report (there are now two “Logic” files, one for each category).

**Figure 6.2:** Test coverage report of the second experiment

It is notable from the coverage report that the branch coverage is exactly 50%. This amount is lower than what it was in the first experiment. However, in this experiment the implicative properties are being tested thoroughly, which was not the case in the first experiment. Considering that the else-clause of the implicative properties is not being triggered, the test coverage will never become
100%. Which is not a problem in that sense, as we don’t intend to test the else-clause. We are more interested in the result of the if-clause of these implicative properties. This is also true for the branch coverage, which can be expected to be 50%. Although the first experiment reported a higher amount for this, it was not testing the implicative properties good enough in the first experiment.

The other criteria we use was the number of bugs that we have found. Using this approach 2 more tests were failing compared to the first experiment in Chapter 5. The division problem was not taken into account when defining the properties, which resulted in these failing cases.

When defining the properties, it was stated that the value should be the exact value, while this is not possible with division. This results in a threat to the definition of this property. Because of this, we consider the bugs that were found as false positives. Although, because of this it might be the case that the actual bugs it might be able to find, are being hidden because of this. Resulting in false negatives: bugs that exist in the system, but are not found by the test framework.

6.5 Conclusion

In this experiment, we generated the input values such that the condition of the implicative properties are satisfied. This revealed 2 additional failing cases that are triggering the division problem and indicating a rounding issue. The generated system tries to hold the exact value, which triggers this situation. It is reasonable that the system tries to hold the exact value, but it is also unclear from the property definitions what should happen in this case.

When defining the properties in Chapter 3, we said that a value should be precise. However, it is not possible to do this for all division cases. This depends on the input values when testing the property. Because of this, we consider the bugs that were found as false positives. The definition of the property has to be updated in order to specify what should happen in case such a division error occurs. Due to this error, it might be the case that existing bugs in the generator are not being detected with this approach.

In this experiment we found false positives and haven’t found additional bugs when testing the implicative properties. The use of implicative properties might not be as effective as using properties that do not. Although more functionalities from the generator are being used, and thus being tested, by this approach. Which wouldn’t be the case when the implicative properties are being removed. The if-statements and preconditions were not being used in the first experiment Chapter 5. The property definitions should be updated such that it is known what should happen in this case of division. In the next experiment we will focus on improving this.

6.6 Threats to validity

Incorrect value generation

We have implemented a custom value generator to generate values for each test case. A random operation is being done on these values to make these random again. There could be an error in the implementation that incorrect values are being created. In Chapter 4 we described a similar threat concerning incorrect value generation. In this case, the traceability of how the values were created is hard because of the random operation.

We have seen such an error when running this experiment multiple times. It sometimes happened that there were additional tests failing. Although these do not trigger in every run, leaving it as a threat to this approach. The error that caused these test to fail was that the preconditions are not met. This was only the case for some properties that are using “smaller than” (<) and “greater than” (>) in its definitions. The input values are set to all zeros, which results in the condition 0 > 0, which results in false. There is a high probability that this is caused by the random operation that is being
done to mutate the input values. As mentioned earlier, the traceability for this is hard and since it does not happen regularly, fixing this issue is left as future work. Leaving it as a threat to our results.
Chapter 7

Experiment 3: Automatic value generation

In the second experiment (Chapter 6), we found that the property definition for using division with the `Money` type wasn’t clear enough in that it couldn’t be satisfied the way how it was specified. In this experiment we aim to improve this, by updating the existing definition such that the property can now be checked.

A limitation in the second experiment was that the value generation was not very dynamic when new property definitions were being added. It should be possible to add additional properties to the specification, such that these properties are being tested automatically by the test framework. To do this, the value generator should be updated too, such that it uses the preconditions in the event definitions to determine the input values. Additional properties can then be added to test the generator.

In this experiment we will use the updated version of the generator, in which the precision errors that were found in the first experiment are fixed. This is done, such that we can focus on possible additional errors that might occur.

7.1 Method

The property definitions of `Division` is being updated for this experiment because the first definition did not take the division problem into account. There was also no definition for rounding the value, as the value was expected to hold the exact value. We will update the definition by implementing a rounding function, such that these properties can be tested by the test framework.

Besides updating the property definitions for `Division`, additional property definitions are being added to test more of the generator. However, as we have seen in the second experiment, the values generator was not very dynamic. Which resulted in that it had to be modified in case an implicative property is being added to the specification. This is the second thing that should be improved, such that additional (implicative) properties do not require modifications to the values generator.

7.1.1 Updating property definitions

Initially, there were only 2 property definitions that were using division, which both are being updated. In order to check for the values such that the property definition can be used, we implement a `round()` method in the `Rebel` specification. This `round` method is intended to return a value which can be used to define the expected behaviour when using division with the `Money` type.

In addition to updating the existing properties that are using division, more properties can be added which were not defined earlier. Additional properties might also lead to additional bugs that the test
framework can detect. For example, properties of inequality when using division, as the ones that were defined for division only used equality. We add more properties to the existing set of property definitions that we defined for Rebel: Properties for division, multiplication, additivity and subtraction.

The additional property definitions for division, multiplication, additivity and subtraction fall under the “Properties of equality and inequality” category and are defined in Subsection 3.2.3, along with the updated definitions. To sum up, the updated and added property definitions are: divisionEquality, divisionInequality, additiveEquality, additiveInequality, subtractiveEquality, subtractiveInequality, multiplicativeEquality and multiplicativeInequality.

Some properties using division are now using the round() method in its definition. Rebel does not provide a way to round a value, which is why we need to define the function in the specification. In Rebel, a function is defined as an expression that is being executed whenever the function is being called. Unfortunately, there is currently no way in Rebel to write down the Scala implementation of this round function. As a workaround, we define the implementation as a String and modify the generator such that the content of the String (removing the quotes) will be the implementation of the function. The round method rounds the Money value to a maximum of 4 decimals.

In Java (and thus in Scala), there are different rounding methods available. These consist of the rounding modes described in the IEEE 854 standard and additional rounding modes as described in [34]. The fifth decimal in our round() method is being rounded by using the “HALF_UP” rounding mode (described in [34]). We chose for the “HALF_UP” as this is more in line with our expectations and this mode is perhaps the most commonly used rounding mode when rounding financial numbers. However, we also analyse the effects of using some of the other available rounding modes. The implementation of the round() method in the Rebel specification is shown in Listing 7.15.

```scala
function round(money: Money): Money =
  "money.currency(money.amount.setScale(4, RoundingMode.HALF_UP))";
```

Listing 7.15: The round() function in the Rebel specification.

7.1.2 Improving dynamicality

The additional properties that are being added also use implication in their definitions. In the second experiment (Chapter 6), the value generator was not dynamic enough in that it requires modifications to the implementation for each implicative property that is being added. In this experiment, we aim to improve this, by using the defined preconditions to determine the input values.

A custom value generator is being created that uses the preconditions to determine the input values. Note that this can be seen as an update to the earlier value generator that was created in Chapter 6. This value generator parses the preconditions and intends to generate values based on these conditions. Since the expressions inside the preconditions can become complex, we focus on a limited version of it. Limited in the sense that we support all of the properties that we defined in Chapter 3, but other expressions that were not used in our definitions might be unsupported.

Most properties are using single variables in its precondition statements. Some are using expressions on the left-hand or right-hand side of a statement, but not on both sides. The value generator that we implement will not support using expressions on both the left-hand and right-hand side, as generating values matching the condition can result in complex formulas. With expressions, we mean a combination of operators with literals and variables. Instead, the value generator requires having at least a variable on one side of the expression.

The code to generate the tuples of input values is shown in Listing 7.16. We can separate this
process into the following steps:

1. Initialize value generation data (Line 7)
2. Traverse and handle statements (Line 9-13)
3. Generate values for remaining, unassigned, variables (Line 15-18)
4. Add values to resulting list (Line 21)

```java
public list[Expr] genValues(str eventName, Preconditions? preconditions, list[Parameter] transitionParams, int amount) {
    println("\> Generating values for event <eventName>");
    list[Expr] valueList = [];
    for (int i <- [0..amount]) {
        calculatedParams = (); // Clear old data
        paramGenData = ("<p.name>" : <p.tipe, -9999.00, 9999.00, true> | p <- transitionParams);
        // Calculating
        for (Statement s <- preconditions) {
            <lhs, rhs, operator> = extractStatementData(s.expr);
            handleConditionStatement(operator, lhs, rhs);
        }
        // Check whether all are determined, if not, determine those using the randomValueProps data
        for (Parameter p <- transitionParams, !calculatedParams["<p.name>"]?) {
            calculatedParams["<p.name>"] = calculateExpression(p.name);
        }
        // Add to list
        valueList += [[getExprForVar("<p.name>")] | p <- transitionParams];
    }
    return valueList;
}
```

Listing 7.16: Code to generate the list of input values for an event.

In the following sections we describe each step in detail.

1. **Initialize value generation data (Line 7)**

To generate a single value, some data is being held to keep track of the conditions which the generated value should fulfill. These conditions are the minimum value, the maximum value and whether the zero value is allowed. Additionally, the result type of the variable is stored, used when generating the final value. This data is stored in a tuple and called RandomValueProps by using an alias in Rascal. This definition is shown in Listing 7.17.

```java
// Minimum: including. So: if 0, then 0 can be a result value when determining it random.
// Maximum: including. So: if 10, then 10.00 is max result value when determining random.
alias RandomValueProps = tuple[Type valueType, real min, real max, bool allowZero];
```

Listing 7.17: The definition of RandomValueProps in Rascal.

The value generator initializes the RandomValueProps for each input variable. Setting the minimum and maximum value to a default value and allowing zero by default.

2. **Traverse and handle statements (Line 9-13)**

Each statement in the preconditions block is being checked. The handleStatement() method handles each statement. The actions done by this method depend on the operators used in the statement that
is being handled.

In case of an expression that only contains variables and uses equality (for example, $x == y$), the value generator assigns a random value to $x$ and assigns the same value to $y$. In case of inequality (for example, $x < y$ or $x > y$), the value generator also assigns a random value to $x$ and adjusts the minimum or maximum bounds of the $y$ value such that it satisfies the condition.

In case there is an expression on one side (for example, $x * y == z$), the expression will be evaluated first. In this case, random values will be assigned to $x$ and $y$. Next, the expression can be evaluated and the result of that is being assigned to $z$. The same is done with inequality relations, but then the minimum or maximum bounds are being set based on the operator.

As mentioned earlier, having expressions on both the left-hand and the right-hand side of the expression is unsupported. The handleStatement() method will throw an error in case this happens.

3. Generate values for remaining, unassigned, variables (Line 15-18)

When handling each expression, some variables already get an assigned value. However, some variables might only have their RandomValueProps updated but do not have an assigned value yet. In this step, the variables that do not have an assigned value, are being assigned a value based on their RandomValueProps.

4. Add values to resulting list (Line 21)

The values that have been determined are being added to the list of generated input values. In the end, the list is being returned, containing all the generated input values that match the preconditions of the event.

7.2 Results

Running the test framework with the addition of the properties defined in (Subsection 3.2.3) results in no additional failing tests compared to the first experiment. Remember that the test framework is run with the generator in which the precision problems related to the Money type are fixed (otherwise there would be additional failing tests, due to the precision problems).

There are still 3 failing tests in total, which are not fixed in the generator yet. These bugs have already been reported in the first experiment and thus are not new in this experiment.

7.3 Analysis

The tests are succeeding due to the implementation of the round() method. It rounds the value of Money to 4 decimals using the “HALF_UP” rounding mode. The number 4 is chosen here to ensure that the property definitions should hold up to a precision of 4 decimals when using the round method. However, this can be modified in case a bigger precision is preferred. Changing the precision number to 6 or 8 decimals does not change the results when looking at the number of failing tests. Also, changing the rounding mode to “DOWN” does not affect our results. Although, when using a bigger precision in combination with another rounding mode, such as “FLOOR” or “TOP”, can result in some failing tests. This is being caused by an edge case in which the difference would be slightly off between the two values.

We stay with the implementation of “HALF_UP” as rounding mode, as this is more in line with our expectations and this mode is perhaps the most commonly used rounding mode when rounding financial numbers. This rounding mode is also described as a “requirement for many financial calculations” [34]. One might want to use the “DOWN” rounding mode because a bank would prevent
losing money because of this rounding issue. When this is the case, it can always be changed in the Rebel specification when needed, we stick with “HALF_UP” as rounding mode.

For this experiment, additional property definitions have been added to test the generator. The value generator now determines the values based on the preconditions of the properties. Thus the additional properties used in this experiment did not require more effort than writing these down in the Rebel specification. Compared to the second experiment (Chapter 6) this is a huge improvement, as it does not require the developer to modify the value generator when implicative properties are being added.

Another way how the values could be generated is by using existing solvers. Since the Rebel toolchain already makes use of a bounded model checker to check a specification, this could be used to translate an expression and retrieve values for which the condition holds.

We have looked into this, by using the Z3 solver. However, the solver always returns the same number when executing it multiple times. Which means that the 100 values that we would ask from the generator, will be exactly the same. A workaround would be to then add the number that was received earlier as an additional constraint, such that 100 unique values are being retrieved. But the problem still remains, as executing the same script multiple times will result in the same values. Another possibility would be to change the seed of the random generator that is being used to generate the values, resulting in different values, however, then a random seed should be used each time the solver is being run to make the generated values unique. In order to integrate the solver with the test framework, the value generator has to be changed.

Checking a Rebel specification by using the existing tool chain can already take up some time. Which is probably related to the translations that have to be done and actually running the solver. We have not measured the exact duration of each step in this case, but we expect that generating 100 random input values by using this existing implementation in the tool chain requires some time. Resulting in a overall run time increase for the test framework.

Other solvers might work better for this approach, but these still require an update to the values generation part of the test framework to integrate with such solvers. We stick with our own implementation. The properties that are defined until now can be tested by the test framework automatically with this implementation, as well as additional property definitions in case these are supported by this implementation.

7.4 Evaluation criteria

The coverage in this experiment is almost the same as in the second experiment. The implicative properties are being checked in the same way. In this experiment, additional properties were added. The total test coverage is 88.28% (which is 0.42% lower than the second experiment), shown in Figure 7.1. This shows that adding additional properties do not increase or decrease the test coverage in big amounts. The small percentage loss could be related to the fact that we do not check the else clauses of the implicative properties.
The branch coverage is still reported to be 50%, while the number of branches has been increased. This is correct, as the additional properties are using the implication symbol, which is being translated to if-else statements. This introduces these extra branches. As described before, we do not intend to check the else-clause but are more interested in checking the if-clause.

Unfortunately, no additional bugs have been found in this experiment. Additional properties were being used in this experiment compared to the other experiments, but these did not reveal more bugs. The implicative properties are checked in the same way as in the second experiment, but the values are now generated based on the preconditions of the event definitions in the Rebel specification.

The properties that have been added did not require changes in the value generator anymore. Instead, these only had to be written down in the Rebel specification. The test framework automatically determines the input values for it. This is an improvement compared to the second experiment, as there it would be required to update the value generator whenever an implicative property is added. There are still some limitations on the value generator, for example, it does not support complex expressions as preconditions. This is also not needed for the current set of property definitions and might or might not be needed in the future. Because of this, we leave the implementation for supporting complex expressions in the value generation as future work.

7.5 Conclusion

This experiment focused on improving the value generation, such that it is not required to update the test framework whenever an implicative property is being added to the Rebel specification. The value generator now uses the statements in the precondition block to determine the input values, which hereby satisfy the preconditions. We have shown that this increased the dynamicality by adding additional properties. Adding these properties to the test framework was only a matter of translating those property definitions to events in the Rebel specification.

This experiment did not reveal extra bugs in the generator, but we have shown that adding more property definitions does not require an update to the test framework when it comes to implicative properties. However, there are still some limitations when it comes to the value generation. One of the limitations is that it does not work with complex expressions as preconditions. The value generator will throw an exception when an expression is given that is unsupported.

7.6 Threats to validity

In Chapter 4 and Chapter 6 we described the threat concerning generating incorrect input values. In this experiment we brought changes to the way how these input values are being determined. However, the threat still remains as the current implementation could contain some errors too. When running the test framework, we haven’t encountered issues with this, other than the one described in Chapter 6.
Chapter 8

Discussion

In this chapter we discuss the sub research questions.

RQ 1: Which properties are expected to hold on the semantics of the generated code?

There were no existing properties defined for Rebel that were expected to hold. We have defined many properties for Rebel. With the focus on the Money type, which is considered the most important type for a bank.

Many properties have been defined, but are these all the properties? Is each definition correct? Some properties might be considered incorrect or overlapping with others. We have seen this with the definition of division that was incorrect, which was encountered in the second experiment. In the third experiment the property definitions for division have been updated. Additionally, extra properties were added to the list for the third experiment, additivity, subtraction and multiplication. This showed how additional properties could be added to the test framework to test the generator. In case of incorrect, missing or overlapping property definitions, the current set of property definitions could be updated as we have done with the third experiment. There can be many combinations among the different types of Rebel and the supported operators. Therefore, the set of properties that we have defined in Chapter 3 is not the complete set of expected properties in Rebel. Although we focused on the Money type in Rebel, there could be more properties for this. Thus, the set of properties that we have defined is also not complete when only focusing on the Money type.

Each property definition is defined as an expression, the test framework is currently limited to checking the generator on these kind of properties. We can also think of other properties that should hold in Rebel. For example, the behaviour of sync block: what are the properties of the sync block in Rebel? Can these be described in a Rebel specification? If so, the current test framework would not support that definition yet because it can currently only cope with expression-based properties.

RQ 2: How can we test each property as automatically as possible to find bugs in the generator?

We have described a way how the generator can be tested by using property-based testing. In order to check the generator, a Rebel specification is created containing the property definitions. This specification is used by the test framework to generate tests and run these tests against the generated system.

The initial version of the test framework used random values to test each property. The first experiment showed that this doesn’t work for the implicative properties. The if-clause of these implicative properties were not being triggered when using random values. In the second experiment the test framework was improved. The input values where determined such that the condition of the implicative property was being satisfied. Additionally, these values were being mutated by a random
operation, to keep the randomness of the input values. This improvement to the test framework revealed some incorrect property definitions. Also, the implementation of how the input values were determined was not very dynamic. Because every time an implicative property was being added to the specification, an update was required to the test framework.

In the third experiment the input values were determined in a more dynamic way, by using the preconditions to determine the input values. This requires a property definition to define its preconditions. As a result, it was not required to update the test framework when implicative properties were being added. This is shown in the third experiment.

We evaluated the test framework in each experiment by looking at the test coverage and the number of bugs that have been detected. The coverage details showed that in the first experiment the if-clause of the implicative properties was not being triggered. Which lead to an improved version in the second and third experiment. Furthermore the branch coverage reported a consistent 50% over the second and third experiment, which is caused by the implicative properties. The else-clause is not being tested, but that is also not the intention of that property.

The overall test coverage for each experiment was roughly the same (87.80%, 88.87% and 88.28%). This shows that despite of the improvements made to the test framework, the test coverage did not change. However, each experiment is different, some of the differences have an impact on the coverage results:

- The first experiment used random input values for each property. The else-clause of the implicative properties were defined as true, resulting that each implicative property was considered correct.
- In the second experiment the else-clause of the implicative properties were set to false. The input values are generated such that these satisfy the condition of the if-clause. These values are also being mutated when running the test, to keep the actual values random.
- During the second experiment a fixed version of the generator was being used. This version contains fixes for the issues that were related to the Money type. The coverage has been determined with the use of the fixed version of the generator.
- In the third experiment some incorrectly defined properties were updated and additional properties have been added.
- Note that the test framework has been updated throughout the experiments. Meaning that the third experiment contains the changes of the second experiment.

The test coverage of the first experiment would be considerably lower in case the else-clause of the implicative properties would be false. This would result in that every test concerning an implicative property would fail. Although the result of this would find many false-positives, because it just returns false in its definition.

The test coverage over the second and third experiment are roughly the same (only 0.59% difference). The third experiment tested 17 more properties than the second experiment. This shows that adding additional properties does not have much impact on the test coverage. The additional properties are being checked automatically by the test framework because these properties are defined in the Rebel specification.

RQ 3: What kind of bugs can be found using this approach and how many?

Multiple bugs were found using property-based testing to check the generator. The generator failed to satisfy a total of 8 properties that we have defined. Some properties triggered different kind of bugs. We can categorize the bugs found as follows:

Precision errors: Errors causing an unexpected outcome value when using calculations.
CHAPTER 8. DISCUSSION

Overflow/underflow errors: Errors happening because of a value limit that has been reached on specific types.

Compilation errors: Errors that make the generated system unable to compile, resulting that the generated system cannot be used.

Despite of the bugs that were found, we also found that two of the initial property definitions were incorrect. This was not found in the first experiment because of the random values. In the second experiment, this finding was the result of using a fixed version of the generator in which the precision errors related to the Money type were fixed. This shows that some errors can hide other bugs, this is also known as finding false-negatives. We do not consider the invalid property definitions as bugs. This was not due to an error in the generator, but an error in the property definitions.

The test framework should also be able to find other kind of bugs which did not exist in the generator that we have used. We can think of certain operations that are not correctly implemented. For example, when addition would act like subtraction. This would cause each the test case that use addition to fail.

Also, we perhaps have missed some property definitions that would trigger more bugs in the generator. The use of other types and operators that exist in Rebel can lead to more bugs. Rebel also supports data structures like map and set which we did not cover during this thesis. Therefore we cannot conclude that the test framework would only find the kind of bugs that we found. But we can conclude that these kind of bugs can at least be found, which we have shown in the experiments.

Below we provide a discussion about the different kind of bugs that have been found, precision errors, overflow/underflow errors and compilation errors.

Precision errors

As we have seen, the Money precision errors both occurred when using Percentage values as well as when using Integer values to operate with the Money type. We were able to reproduce the issue in a clean REPL environment and concluded that the problem existed in the open-source library, called Squants [21]. This library is being used for the Money type. In order to solve this problem, we created an issue on Github1 related to the precision problems on the Money type. A contributor responded and fixed the issue within a day, the change will be included in the next version of the library (1.4).

The generator should be updated to use this version of the library in order to fix these precision errors. This is what has been done in the second experiment, where we use the fixed version of the generator. So would updating the library fix all the precision errors that have been found? No, in the first experiment one of the precision errors found was being caused by multiplying a Percentage with an Integer. This precision error is not related to the precision errors when using the Money type and still remains to exist in the generator. There might be more precision errors when using the generator, but then we did not define properties for these cases and thus did not trigger those. Thus, adding more property definitions might lead to more bugs, including precision errors.

Overflow/underflow errors

The overflow/underflow errors are caused by using the Integer type. On one hand, this could be prevented by checking the operations beforehand for overflow errors. On the other hand, this could be the expected behaviour when an Integer is being used in Rebel. As Integers are known have such limits that are also dependent on the platform the application is run [33]. However, in Rebel there is currently no other type that can be used to hold a bigger number. For example in Java there is Long for a larger number or BigDecimal for even bigger numbers. This would mean that Rebel does not support such big numbers or that a custom type must be used for this. Although Rebel does support custom types, the generator does not support custom types yet. Meaning that custom types will not

1https://github.com/typelevel/squants/issues/265
work with the current implementation.

Considering that Rebel does not provide another type for bigger numbers, we conclude that the Integer type should also hold bigger numbers. In this case, Integer is being used to represent a number value in Rebel. Since the specification is about banking products and it probably could happen that a big number is needed. After all, we cannot know this for sure, as Rebel does not provide a specification yet of each of type in Rebel. Maybe it is not needed or required to support bigger numbers than an integer, resulting in that our conclusion is incorrect. However, what would be the minimum and maximum value it can hold then? Would it be the value of a 32 bit Integer, a 64 bit Integer or perhaps even arbitrary size numbers? In case our conclusion is incorrect, this should be defined too. The property definition could be updated to describe the actual bounds of the input values.

**Compilation errors**

In the first experiment, the test framework was initially being terminated because of a compilation error. Although one assumption was that the generated system should be able to compile, another assumption was that the specification was consistent. The specification containing all the properties is consistent, as Rebel did not report any syntactic or semantic errors with the type checker. Because of this, it was expected that the generated system could compile and run.

The test framework is able to find such compilation errors, as we have seen in the first experiment. However, the test framework cannot proceed with running the generated tests in this case, thus hiding other bugs that can exist. The test framework does not specifically check for compile errors and whether these are expected or not. This is out of the scope of this thesis, because there can be many causes of compilation errors in the generated system and detecting the actual causes of those is hard, if not impossible.

The cause of the compilation error we found in the first experiment was an implementation error in the open-source library that is being used for the Money type called Squants [21]. We created a Github issue\(^2\) describing this problem. When this issue has been fixed, the generator should be updated in order to fix this compilation error. Since this issue has not been fixed during this thesis, we temporarily disabled the property causing this problem.

\(^2\)https://github.com/typelevel/squants/issues/281
Chapter 9

Conclusion

In this thesis, we have shown a way how the generator can be tested by using property-based testing. This is done by generating tests based on the Rebel specification and making use of the generator to generate the tests. The Rebel specification is build up based on a set of defined properties of Rebel. These property definitions were not defined earlier, thus we defined many expected properties in Rebel.

With test framework we have found some bugs in the generated system that were unknown before. This proves that this approach already worked to identify some problems in the generator that were not known before. Additionally, we contributed to an open-source library called Squants, by issuing two reports of bugs that existed in the library.

To answer the main research question, we defined and answered the three sub research questions, which have been discussed in Chapter 8. The main research question was as follows:

How can the generator in the Rebel toolchain be tested automatically, by using the generated system, to check whether the implementation works as expected?

A Rebel specification is created from the defined properties. Next, the existing generator is being used to generate the generated system and to translate the properties (in the Rebel specification) to test cases. When running the test framework, we found some errors by using the generated system. Additionally, we were able to detect a compilation error and incorrectly translated formulas.

We conclude that, by using this approach, the semantics of the generated code can be checked automatically on whether it satisfies the defined properties. With this approach, we have found bugs in the generated code that were unknown before. The full source of our test framework is available on Github\textsuperscript{1}.

9.1 Future work

Complete property definitions

In this thesis, we defined some properties on the Rebel language, which we consider to hold when using Rebel. This list of property definitions on Rebel is not complete, there can be many more properties and there are other types available in Rebel which we did not cover. Defining these and adding them to the set of property definitions is left as future work.

Values generation by using solvers

The current value generation works for the properties that we defined throughout this thesis. When additional properties are being added, the value generator might or might not have to be updated. This depends on what preconditions have to be parsed for that property and whether the current

\textsuperscript{1}https://github.com/alexkok/master-thesis
implementation supports these operations.

Another way how the value generator could work is to use some tools to determine the random values. As we have discussed in Chapter 7, the SMT solver would theoretically be an option. Although in our attempts this resulted in having the same input values every time the test framework is run. To counter this problem, other solvers might be more useful to implement this behaviour, such as SageMath [35]. SageMath can parse expressions and determine the conditions to which each variable in the expression should hold. When using this, it would mean that the value generator has to be modified such that it integrates with this tool. Another approach could be used for this too, such as concolic testing [36] to determine the input values. We did not use this in this thesis, due to time constraints.

Multiple generators
Throughout this thesis, we have only tested the Scala/Akka generator which is developed by ING. Since there are more generators available, the test framework can be improved such that it is compatible with the other systems that can be generated by using one of the other generators that are available within ING. By doing this, the same property definitions can be tested on different kind of generated systems. This can be used to detect inequalities among the generated systems. Although the properties will remain the same in this case, generating the test suite requires some modifications for each generator.

*Templating* is used to generate the test cases. The way how the tests should be written down depends on the generated system. When a generated system uses a different setup or is written in another language then Scala, an update is needed for this part of the test framework. The other parts of the test framework can be reused, as the test framework also uses the generator from which the generated system is created.

Mutation testing
Another metric to evaluate the effectiveness of the test framework would be to use mutation testing. The mutation coverage could be used to measure how effective the test framework would be (the number of mutants created and the number of killed mutants). Unfortunately, there is currently a limited support for mutation testing for Scala systems. PIT for example, cannot meaningfully mutate Scala code [31]. Because of this, we did not use this as metric to evaluate the test framework.

*Scala* compiles to Java bytecode, mutating on bytecode level could be used too. However, some mutants might not be relevant in that the modifications would not affect the implementation, which can lead to false-positives.

Other components
Besides completing the set of property definitions, the test framework could also be extended such that it can also support property definitions concerning other components of the system. Such as the sync block definitions, which defines actions that should happen synchronously. Or performance measures when interacting with data in the system. Currently, such components are not being tested and are unsupported, while these components are also important for the bank. This is left as future work.

It might not be possible to check each and every component by using this approach. For example, it might be unable to check how multiple generated systems would integrate with each other and if this is done correctly. The generated system is configured to work with multiple distributions of the system, but is not being tested thoroughly yet.
Acknowledgements

First of all I would like to thank my supervisor, Jurgen Vinju, for his help and guidance throughout the thesis. Despite of his busy schedule, he always found a way to help, to discussing about the topic and to provide feedback throughout my thesis.

Also I would like to thank my company supervisor, Jorryt-Jan Dijkstra, for his help, discussions and feedback throughout this thesis. Thanks to my colleagues at the ING and my co-intern on the same topic, Thanusijan Tharumarajah for the discussions we had and for the feedback throughout this thesis.

I would like to thank my fellow students of the master Software Engineering at the University of Amsterdam, for the great times, support and insights throughout past year. The teachers of this masters programme for the interesting and informative courses. Thanks to Ana Oprescu, whom was always available during this master programme for questions, discussions and information about the master programme.
Bibliography


Appendix A

Property definitions in Rebel

```plaintext
event reflexiveEquality(x: Money) {
    postconditions {
        new this.result == ( x == x );
    }
}

event reflexiveInequalityLET(x: Money) {
    postconditions {
        new this.result == ( x <= x );
    }
}

event reflexiveInequalityGET(x: Money) {
    postconditions {
        new this.result == ( x >= x );
    }
}

event symmetric(x: Money, y:Money) {
    postconditions {
        new this.result == ( (x == y) ? y == x : True );
    }
}

event transitiveEquality(x: Money, y: Money, z: Money) {
    postconditions {
        new this.result == ( (x == y && y == z) ? x == z : True );
    }
}

event transitiveInequalityLT(x: Money, y: Money, z: Money) {
    postconditions {
        new this.result == ( (x < y && y < z) ? x < z : True );
    }
}

event transitiveInequalityGT(x: Money, y: Money, z: Money) {
    postconditions {
        new this.result == ( (x > y && y > z) ? x > z : True );
    }
}
```

Listing A.18: The property definitions as Rebel specification
event transitiveInequalityLET(x: Money, y: Money, z: Money) {
    postconditions {
        new this.result == ( (x <= y && y <= z) ? x <= z : True );
    }
}

event transitiveInequalityGET(x: Money, y: Money, z: Money) {
    postconditions {
        new this.result == ( (x >= y && y >= z) ? x >= z : True );
    }
}

event additive(x: Money, y: Money, z: Money) {
    postconditions {
        new this.result == ( (x==y) ? x+z == y+z : True );
    }
}

event additive4params(x: Money, y: Money, z: Money, a: Money) {
    postconditions {
        new this.result == ( (x == y && z == a) ? x+z == y+a : True );
    }
}

event commutativeAddition(x: Money, y: Money) {
    postconditions {
        new this.result == ( x+y == y+x );
    }
}

event commutativeMultiplicationInteger1(x: Integer, y: Money) {
    postconditions {
        new this.result == ( x*y == y*x );
    }
}

event commutativeMultiplicationInteger2(x: Money, y: Integer) {
    postconditions {
        new this.result == ( x*y == y*x );
    }
}

event commutativeMultiplicationPercentage1(x: Percentage, y: Money) {
    postconditions {
        new this.result == ( x*y == y*x );
    }
}

event commutativeMultiplicationPercentage2(x: Money, y: Percentage) {
    postconditions {
        new this.result == ( x*y == y*x );
    }
}

event associativeAddition(x: Money, y: Money, z: Money) {
    postconditions {
        new this.result == ( (x+y)+z == x+(y+z) );
    }
}

event associativeMultiplicationInteger1(x: Integer, y: Integer, z: Money) {
    postconditions {
        new this.result == ( (x*y)*z == x*(y*z) );
    }
}

Listing A.19: The property definitions as Rebel specification (continued)
event associativeMultiplicationInteger2(x: Money, y: Integer, z: Integer) {
    postconditions {
        new this.result == ( (x*y)*z == x*(y*z) );
    }
}

event associativeMultiplicationPercentage1(x: Money, y: Percentage, z: Integer) {
    postconditions {
        new this.result == ( (x*y)*z == x*(y*z) );
    }
}

event associativeMultiplicationPercentage2(x: Integer, y: Money, z: Percentage) {
    postconditions {
        new this.result == ( (x*y)*z == x*(y*z) );
    }
}

event distributiveInteger1(x: Money, y: Integer, z: Integer) {
    postconditions {
        new this.result == ( x*(y+z) == x*y + x*z );
    }
}

event distributiveInteger2(x: Integer, y: Money, z: Money) {
    postconditions {
        new this.result == ( (y+z)*x == y*x + z*x );
    }
}

event distributivePercentage1(x: Percentage, y: Money, z: Money) {
    postconditions {
        new this.result == ( x*(y+z) == x*y + x*z );
    }
}

event distributivePercentage2(x: Percentage, y: Money, z: Money) {
    postconditions {
        new this.result == ( (y+z)*x == y*x + z*x );
    }
}

event additiveIdentity1(x: Money) {
    postconditions {
        new this.result == ( x + EUR 0.00 == x );
    }
}

event additiveIdentity2(x: Money) {
    postconditions {
        new this.result == ( EUR 0.00 + x == x );
    }
}

event multiplicativeIdentity1(x: Money) {
    postconditions {
        new this.result == ( x*1 == x );
    }
}

event multiplicativeIdentity2(x: Money) {
    postconditions {
        new this.result == ( 1*x == x );
    }
}

Listing A.20: The property definitions as Rebel specification (continued)
event additiveInverse1(x: Money) {
    postconditions {
        new this.result == ( x+(-x) == EUR 0.00 );
    }
}

event additiveInverse2(x: Money) {
    postconditions {
        new this.result == ( (-x)+x == EUR 0.00 );
    }
}

event antisymmetryLET(x: Money, y: Money) {
    postconditions {
        new this.result == ( (x <= y && y <= x) ? x == y : True );
    }
}

event antisymmetryGET(x: Money, y: Money) {
    postconditions {
        new this.result == ( (x >= y && y >= x) ? x == y : True );
    }
}

event division1(x: Money, y: Integer, z: Money) {
    postconditions {
        new this.result == ( (x*y == z) ? (x == z/y) : True );
    }
}

event division2(x: Money, y: Integer, z: Money) {
    postconditions {
        new this.result == ( (x == z*y) ? (x/y == z) : True );
    }
}

event multiplicativeZeroProperty1(x: Money) {
    postconditions {
        new this.result == ( x*0 == EUR 0.00 );
    }
}

event multiplicativeZeroProperty2(x: Money) {
    postconditions {
        new this.result == ( 0*x == EUR 0.00 );
    }
}

event anticommutativity(x: Money, y: Money) {
    postconditions {
        new this.result == ( x-y == -(y-x) );
    }
}

event nonassociativity(x: Money, y: Money, z: Money) {
    postconditions {
        new this.result == ( (x-y)-z != x-(y-z) );
    }
}

event trichotomy(x: Money, y: Money) {
    postconditions {
        new this.result == ( x < y || x == y || x > y );
    }
}

Listing A.21: The property definitions as Rebel specification (continued)
Appendix B

Additional property definitions

```
event divisionEquality1(x: Money, y: Integer, z: Money) {
    preconditions {
    round(x*y)) == round((z));
    }
    postconditions {
    new this.result == ( (round(x*y) == round(z)) ? (round(x) == round(z/y)) : False );
    }
}

event divisionEquality2(x: Money, y: Integer, z: Money) {
    preconditions {
    round((x)) == round((z*y));
    }
    postconditions {
    new this.result == ( (round(x) == round(z*y)) ? (round(x/y) == round(z)) : False );
    }
}

event divisionEquality3(x: Money, y: Money, z: Integer) {
    preconditions {
    x == y;
    z != 0;
    }
    postconditions {
    new this.result == ( (x == y && z != 0) ? (x/y == y/z) : False );
    }
}

event divisionInequalityLT1(x: Money, y: Money, z: Integer) {
    preconditions {
    x < y;
    z > 0;
    }
    postconditions {
    new this.result == ( (x < y && z > 0) ? (x/z < y/z) : False );
    }
}
```

Listing B.22: Additional property definitions as Rebel specification
event divisionInequalityLT2(x: Money, y: Money, z: Integer) {
    preconditions {
        x < y;
        z < 0;
    }
    postconditions {
        new this.result == ( (x < y && z < 0) ? (x/z > y/z) : False );
    }
}

event divisionInequalityLGT1(x: Money, y: Money, z: Integer) {
    preconditions {
        x > y;
        z > 0;
    }
    postconditions {
        new this.result == ( (x > y && z > 0) ? (x/z > y/z) : False );
    }
}

event divisionInequalityLGT2(x: Money, y: Money, z: Integer) {
    preconditions {
        x > y;
        z < 0;
    }
    postconditions {
        new this.result == ( (x > y && z < 0) ? (x/z < y/z) : False );
    }
}

event additiveEquality(x: Money, y: Money, z: Money) {
    preconditions {
        x == y;
    }
    postconditions {
        new this.result == ( (x==y) ? x+z == y+z : False );
    }
}

event additiveEquality4params(x: Money, y: Money, z: Money, a: Money) {
    preconditions {
        x == y;
        z == a;
    }
    postconditions {
        new this.result == ( (x == y && z == a) ? x+z == y+a : False );
    }
}

event additiveInequalityLT(x: Money, y: Money, z: Money) {
    preconditions {
        x < y;
    }
    postconditions {
        new this.result == ( (x<y) ? x+z < y+z : False );
    }
}

event additiveInequalityGT(x: Money, y: Money, z: Money) {
    preconditions {
        x > y;
    }
    postconditions {
        new this.result == ( (x>y) ? x+z > y+z : False );
    }
}
event subtractiveEquality(x: Money, y: Money, z: Money) {
    preconditions {
        x == y;
    }
    postconditions {
        new this.result == ( x==y ? (x-z == y-z) : False );
    }
}

event subtractiveInequalityLT(x: Money, y: Money, z: Money) {
    preconditions {
        x < y;
    }
    postconditions {
        new this.result == ( x<y ) ? (x-z < y-z) : False ;
    }
}

event subtractiveInequalityGT(x: Money, y: Money, z: Money) {
    preconditions {
        x > y;
    }
    postconditions {
        new this.result == ( x>y ) ? (x-z > y-z) : False ;
    }
}

event multiplicativeEquality(x: Money, y: Money, z: Integer) {
    preconditions {
        x == y;
    }
    postconditions {
        new this.result == ( x==y ? (x*z == y*z) : False );
    }
}

event multiplicativeInequalityLT1(x: Money, y: Money, z: Integer) {
    preconditions {
        x < y;
        z > 0;
    }
    postconditions {
        new this.result == ( x<y && z > 0 ) ? (x*z < y*z) : False ;
    }
}

event multiplicativeInequalityLT2(x: Money, y: Money, z: Integer) {
    preconditions {
        x < y;
        z < 0;
    }
    postconditions {
        new this.result == ( x<y && z < 0 ) ? (x*z > y*z) : False ;
    }
}

event multiplicativeInequalityGT1(x: Money, y: Money, z: Integer) {
    preconditions {
        x > y;
        z > 0;
    }
    postconditions {
        new this.result == ( x>y && z > 0 ) ? (x*z > y*z) : False ;
    }
}

Listing B.24: Additional property definitions as Rebel specification (continued)
APPENDIX B. ADDITIONAL PROPERTY DEFINITIONS

Listing B.25: Additional property definitions as Rebel specification (continued)