ATUR: Automated Testing Using Rebel

Testing a generator using its input language and output system

Alex Kok
alex.kok@student.uva.nl

September 6, 2017, 53 pages

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

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Contents

Todo's

Chapter 1

Introduction

— Short about Rebel

Large systems often suffer from domain knowledge that is implicit, incomplete, out of date or contains ambiguous definitions.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

— Aim of project

Generators are being used to generates a 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 program is not checked against the specifications, meaning that the generated program 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 program against *Rebel* specifications.

1.1 Problem statement

— Translation from generator and why automatically

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 [?]. 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 we automatically test the generator in the *Rebel* toolchain, 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.

— Possibility: PBT + short explanation

Property-based testing is an approach to validate whether an implementation satisfies its specification [?]. It describes what the program should or should not do. As [?] describes: "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 [?, ?, ?], 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.

— Doing it too, worked already on x, x and x

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.

— Hypothesis, we will detect

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

RQ 1: Which properties are expected to hold on the generator?

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 generator takes a *Rebel* specification as input. Which contains the properties that are expected to hold in the generated system. Next, *Scala* tests are being generated based on the properties, using the existing generator to translate the expressions used in the properties. These tests will be run against the generated system, to check whether each property holds. In case a test of a failing test, a bug has been found. There are multiple generators available within *ING*. Throughout this thesis, we will use only one generator, which is the Scala/Akka generator. This generator is often used for other experiments too and is currently considered the most mature generator.

— In short: how. Details are in CH3

In order to run the test suite, we assume that the generated system can be compiled and that it can be run. Furthermore, 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 error about the specification.

— Assumptions

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 found 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.

— Not detecting everything, but checking properties

1.2 Research method

We will start off with defining the properties that are expected to hold on the generator. Then we describe how these properties can be tested on the generator, using one property to demonstrate the working of the test framework. We can then run the tests suite against the generated system and check if this method actually works to detect bugs.

— First defining properties, then small example

Next, we generate tests for each of these properties and run these against the generated system. After running the test suite, the result is being evaluated. When one or more tests are failing, a bug is found. However, we need to investigate the failing case such that we discover what the actual bug is. After evaluating we improve our test framework and continue to run and evaluate the results again.

— Generate, run, evaluate results and improve again

1.3 Contribution

We provide definitions of the properties that are expected to hold when using the *Rebel* language and its toolchain. Allowing to reason about the generator and whether the implementation satisfies these properties. There are no property definitions of *Rebel* yet, 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. We come with a solution that uses the input that is required for the generator and the output of the generator in order to check the generator. This process should be as automatic as possible, such that it doesn't require much time to make use of it. Our test framework will combine the required steps to detect bugs as automatically as possible.

The generator that is being used is expected to contain bugs. Therefore, we expect to detect bugs in the generator that aren't known yet. These bugs are then known bugs and can be fixed. The bugs found will then also indicate what kind of bugs we can detect in a generator by using this approach.

1.4 Related work

1.4.1 Random testing

— Describe random testing (FDRT)

Random testing is a technique in which random values are being used as input for the test cases. *QuickCheck* [?] and *Randoop* [?] 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.

— Describe FDRT

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.

— About Randoop, shortly how it works

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.

— Our case, why existing approaches can't be used

Approaches like *QuickCheck* [?] and *Randoop* [?] 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 Outline

Write later, when outline is more final.

Chapter 2

Background and context

...

2.1 Rebel

— About Rebel

Rebel is a domain specific language that focuses on the banking industry [?]. Banking products can be specified in the language, with the use of types like *Money* and *IBAN*. 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* [?]. *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*.

— What Rebel exactly checks

Checking a *Rebel* specification is based on Bounded Model Checking [?]. 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 (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 certain 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.

— Generating system from specification

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. Although the specification is being checked by using bounded model checking, the generated system is not being checked against the specification. Unfortunately, it is not possible to also use the bounded model checker to test the generator or the generated system. As the generated system is written in *Scala*, while the existing checker only supports checking the *Rebel* specification itself.

2.2 The Scala/Akka generator

— Different generators, we use 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*. *Rebel* defines the states and the transitions between the states in the *lifecycle* block, which can be seen as the Finite State Machine definition of the specified product. The Scala/Akka generator is one of these generators. It generates a system that is written in *Scala*, uses *Akka* [?] as actor system and uses Cassandra [?] as database. A resulting system of this generator is also tested thoroughly with performance tests, to reason about how well this system performs with its architecture. However, this does not check the implementation of the generated system against the *Rebel* specification. 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. Throughout this thesis, we will only make use of this generator.

A specification can be defined in terms of a Labeled Transition System [?], 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 a Labeled Transition System. 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.

— Short explanation on implementation generated system + why

Rebel introduces custom types, such as *IBAN*, *Percentage* and *Money*, these types are not supported natively in *Scala*. For these cases, a library or own implementation is used. An example is the *Money* type, which is available in the *Squants* [?] 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*.

— Diff types: known/expected problems. We have to specify expected properties

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.

— About property-based testing

A well-known tool which is based on property-based testing is *QuickCheck*, which is written for *Haskell* [?]. 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 continues to the next property. Note that these values of 100 and 1000 are the default values, these can be adjusted when needed.

— About QuickCheck

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*) [?] and *ScalaCheck* (for *Scala*) [?]. *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 use reflection to test this [?].

— About the ports of it

There is no *QuickCheck* implementation for *Rebel*. As 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.

— No QuickCheck for Rebel, not using ScalaCheck

We can also modify it to our needs when we want to improve our test suite. One of the things that we might 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 ⟹ 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*.

— Custom modifications

2.4 Terminology

— Confusing terminologies and abstractions

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 ($\Longrightarrow$) 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 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.

Test suite
The collection of generated test cases, 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 can be seen as transitions in a Labeled Transition System.

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 [**?**]. But what are the expected properties of the generator? In this chapter, we will try to answer the first research question:

— Specific types in Rebel

RQ 1: Which properties are expected to hold on the generator?

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.

3.1 Determining the properties

Currently, there are no definitions available of what the properties are of each type and operation 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*. Thus we are not able to define all the properties that exist in the *Rebel* language, as there can be countless of properties and combinations among the different types that *Rebel* supports. 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.

— Properties not defined for Rebel. We must define these

For types like *Integer*, the axioms of algebra can be used to determine whether the implementation is correct. These are most likely translated to *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 [**?**, **?**, **?**]. We provide an explanation of why a certain property is expected in *Rebel*.

— Types and axioms

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 computation properties of Floating-Point Arithmetic, as defined in the IEEE standards 754 or 854? Since the *Rebel* is intended to be a formal specification language for banking products, we don't expect that the described problems with this arithmetic are intended to exist on the *Money* type. Considering that a 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 [**?**]. Such errors should be avoided when using the *Money* type. The author of [**?**] also describes that the intention of the *Money* type is to avoid this:

— Amount, not floating-point arithmetic

> "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 [?]

In [?] 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 with 3 people, everyone would receive 33 cents, but 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 (the last cent would go to in this case) [?]. 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.

— Money operations, expecting division problem

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 [?]. However, this is not implemented in *Rebel* yet and thus it is unsupported to use operations on it when using different currencies.

— Not between 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.

— Equality, currency overrules

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. Meaning that the precision errors would have to be prevented. An exception on this is when the result would be in the form of a fraction, for example, 1/3. As this results in a value which we have to round up sometime. To fix this, we say that in this case, we use x decimals when calculating, without rounding the x+1th decimal. When using properties that contain division, this should be taken into account in case the system fails on these tests.

— Thus: precise value, rounding only for division

Define X, 4 decimals minimal?

3.2 Property definitions

The properties that we define are based on the known axioms in algebra, although not every property can be used.

— Properties based on known axioms, but has restrictions

For example, it isn't possible to multiply two *Money* types with each other in order to support the multiplicative property. This is becasue the type checker of *Rebel* would not allow multiplication between *Money* types, additionally we should think of what the result would be when two Money values are being multiplied. We consider this as an unknown result type, thus it is reasonable that we cannot use multiplication when using only *Money* types. 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 equality 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, which were added at the third experiment (Chapter 7).

3.2.1 Field properties

...

3.2.2 Properties of equality and inequality

...

3.2.3 Additional properties of equality and inequality

...

Put the right properties into the right category. And fix references among them

Reflexivity

Formula	Property name	Variable (Type)
x == x	reflexiveEquality	x: Money
x ≤ x	reflexiveInequalityLET	x: Money
x ≥ x	reflexiveInequalityGET	x: Money

Table 3.1: Reflexivity when using *Money*

The reflexive property is defined as a relation of a type with itself [?]. 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.

— Equality, currency and amount

Symmetry

Formula	Property name	Variable (Type)
x == y ⟹ y == x	symmetric	x: Money
		y: Money

Table 3.2: 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 [?].

— Equality, currency and amount

Antisymmetry

Formula	Property name	Variable (Type)
x ≤ y && y ≤ x ⟹ x == y	antisymmetryLET	x: Money
		y: Money
x ≥ y && y ≥ x ⟹ x == y	antisymmetryGET	x: Money
		y: Money

Table 3.3: 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* (≤) and *greater or equal than*

(≥) relations fit in this category, as shown in Table 3.3. This antisymmetric relation is also expected to hold, as the *Money* type supports these operations.

Commutativity

Formula	Property name	Variable (Type)
x + y == y + x	commutativeAddition	x: Money y: Money
x * y == y * x	commutativeMultiplicationInteger1	x: Integer y: Money
x * y == y * x	commutativeMultiplicationInteger2	x: Money y: Integer
x * y == y * x	commutativeMultiplicationPercentage1	x: Percentage y: Money
x * y == y * x	commutativeMultiplicationpercentage2	x: Money y: Percentage

Table 3.4: Commutativity when using *Money*

These properties are based on the commutative law [?]. 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 § 3.2 there is no known value for multiplying two *Money* variables, 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

Formula	Property name	Variable (Type)
x - y == -(y - x)	anticommutativity	x: Money y: Money

Table 3.5: 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 3.5.

Transitivity

Formula	**Property name**	**Variable (Type)**
$x == y \&\& y == z \implies x == z$	transitiveEquality	x: Money y: Money z: Money
$x < y \&\& y < z \implies x < z$	transitiveInequalityLT	x: Money y: Money z: Money
$x > y \&\& y > z \implies x > z$	transitiveInequalityGT	x: Money y: Money z: Money
$x \leq y \&\& y \leq z \implies x \leq z$	transitiveInequalityLET	x: Money y: Money z: Money
$x \geq y \&\& y \geq z \implies x \geq z$	transitiveInequalityGET	x: Money y: Money z: Money

Table 3.6: Transitivity when using *Money*

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

Associativity

Formula	**Property name**	**Variable (Type)**
(x + y) + z == x + (y + z)	associativeAddition	x: Money y: Money z: Money
(x * y) * z == x * (y * z)	associativeMultiplicationInteger1	x: Integer y: Integer z: Money
(x * y) * z == x * (y * z)	associativeMultiplicationInteger2	x: Money y: Integer z: Integer
(x * y) * z == x * (y * z)	associativeMultiplicationPercentage1	x: Money y: Percentage z: Integer
(x * y) * z == x * (y * z)	associativeMultiplicationpercentage2	x: Integer y: Money z: Percentage

Table 3.7: Associativity when using *Money*

The law of associativity is known on addition and multiplication [**?**]. It defines that the order in which certain operations are done does not affect the result of the whole expression. As described in § 3.2, it is not possible to use multiplication with only *Money* types. In Table 3.7 we define possible combinations for this property, which are accepted by the type checker of *Rebel*.

Non-associativity

Formula	**Property name**	**Variable (Type)**
(x - y) - z != x - (y - z)	nonassociativity	x: Money
		y: Money
		z: Money

Table 3.8: 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 3.8 subtraction is a relation where this property holds. An exception to this would be when each argument is zero.

Distributivity

Formula	**Property name**	**Variable (Type)**
x * (y + z) == x * y + x * z	distributiveInteger1	x: Money
		y: Integer
		z: Integer
(y + z) * x == y * x + z * x	distributiveInteger2	x: Integer
		y: Money
		z: Money
x * (y + z) == x * y + x * z	distributivePercentage1	x: Percentage
		y: Money
		z: Money
(y + z) * x == y * x + z * x	distributivePercentage2	x: Percentage
		y: Money
		z: Money

Table 3.9: Distributivity when using *Money*

The law of distributivity is another well-known law [**?**]. 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, as described in § 3.2. Thus the variable types are an important part of these properties, in Table 3.9 property definitions are shown based on this property.

Identity

Formula	Property name	Variable (Type)
x + 0 == x	additiveIdentity1	x: Money
0 + x == x	additiveIdentity2	x: Money
x * 1 == x	multiplicativeIdentity1	x: Money
1 * x == x	multiplicativeIdentity2	x: Money

Table 3.10: Identity when using *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 *Money* value, the 0 showed in Table 3.10 must be defined in a *Money* format. Thus it must have the same currency as the parameter, with the amount of 0. The *Integer* type can be used for multiplication.

Inverse

Formula	Property name	Variable (Type)
x + (-x) == 0	additiveInverse1	x: Money
(-x) + x == 0	additiveInverse2	x: Money

Table 3.11: Inverse when using *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 *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*(1/x) == 0`), it is not possible to use this definition in our case. As we cannot divide something by a *Money* type, which is why we only define the inverse relation using addition.

Additivity

Formula	Property name	Variable (Type)
$x == y \implies x + z == y + z$	additive	x: Money y: Money z: Money
$x == y$ && $z == a \implies x + z == y + a$	additive4params	x: Money y: Money z: Money a: Money

Table 3.12: 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 3.2, it should not break the equality property on the resulting values.

Property of Zero

Formula	Property name	Variable (Type)
x * 0 == 0	multiplicativeZeroProperty1	x: Money
0 * x == 0	multiplicativeZeroProperty2	x: Money

Table 3.13: 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.

Incorporate (define X)

Division

Formula	Property name	Variable (Type)
x * y == z $\implies$ x == z / y	division1	x: Money
		y: Integer
		y: Money
x == z * y $\implies$ x / y == z	division2	x: Money
		y: Integer
		y: Money

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 dived 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

Formula	Property name	Variable (Type)
x < y || x == y || x > y	trichotomy	x: Money
		y: Money

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.

3.3 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 generator?

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*.

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". An overview of the defined properties is shown in ..., the ellipsis is used to indicate that the property names are cut off for readability. Add table

// **ADD TABLE**

The properties that we have 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 a language, the properties defined here 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. This leads to some threats to validity as there exist other types in *Rebel* too.

The properties that we have defined are based on the axioms of algebra. Many axioms exist, which brings a threat to this approach such that we may have missed certain properties that can trigger even more bugs.

Invalid definitions

We provided properties that should hold on *Rebel* types, specifically when using the *Money* type. We already discussed the lack of properties, but it could be the case that 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. However, when this is the case, the properties can be updated when needed.

Alternative properties

We defined many properties in this chapter. 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 such that it can be substituted. For this thesis, we used some well-known properties and defined those for the *Rebel* types.

Chapter 4

Test mechanics

— Intro, describing contents of chapter

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. But 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

— Describing phases

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.

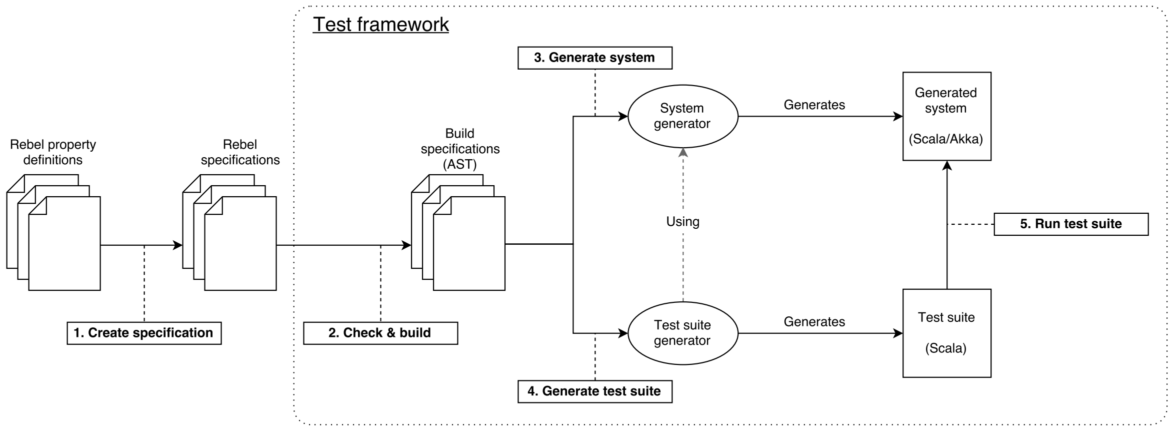


Figure 4.1: Overview of the test framework and the phases

The phases are defined as follows:

1. Create specification

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 *Reflexitivity* 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.

Formula	**Variable**	**Type**
x == x	x	Money

Table 4.1: Property definition of *ReflexiveEquality*

4.1.1 Create specification

— Generator requires spec

The generator requires a consistent *Rebel* specification in order to generate a system. This means that we have to translate the properties (which are defined in Chapter 3) to a *Rebel* specification. A *Rebel* specification consist of a life cycle definition together with its event definitions.

— Property to event translation

A test case should be able to pass values as parameters to test a specific property for a certain number of tries. We can use the events for this in the *Rebel* definition. 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 4.1 the event definition for the *ReflexiveEquality* property written in *Rebel* is shown.

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

Listing 4.1: The event definition for the *ReflexiveEquality* property.

— Further explanation, parameters, result field

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*. 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. (The *new* keyword is used to state how the field changes when the transition is taking place.)

— Also need to define the specification itself

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 4.2 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.

```
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 4.2: 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*.

— Build specification

Building the specification means that the specification is being checked and returns a CST of the specification when the specification is consistent. This is 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.

— Building to CST

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.

— 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.

— Init and generate

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 thus 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.

— Adding test configuration, but not relevant for the project

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.

— Test suite initialization

The test framework can traverse the CST and generate a test case for each event. A test case is

— Test case generation based on event

generated by using the templating feature of *Rascal*, where we fill in event specific data as shown in Listing 4.3. The resulting test case of *ReflexiveEquality* is shown in Listing 4.4.

```
public str snippetTestCase(str eventName, list[str] params, int tries) {
  return "\"work with <eventName>\" in {
          generateRandomParamList(<convertParamsToList(params)>, <tries>).foreach {
            data: List[Any] =\> {
              checkAction(<eventName>(
                <for (i <- [0..size(params)]) {>
                    // Iterate over params. Use getMappedType for the casting again
                    data(<i>).asInstanceOf[<getMappedTypeForParam(params[i])>]
                    // Add a comma if needed
                    <if (i != size(params)-1) {>,<}>
                  <}>
                )
              )
            }
          }
        }";
}
```

Listing 4.3: Test case snippet

```
    "work with ReflexiveEquality" in {
      generateRandomParamList(List("Money"), 100).foreach {
        data: List[Any] => {
          checkAction( ReflexiveEquality(data(0).asInstanceOf[Money]) )
        }
      }
    }
```

Listing 4.4: An example of a generated test

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.

Explanation of a complete test file

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 § 4.1.1 the test suite finishes successfully, as shown in Listing 4.5.

How to run the test suite + result

```
[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 4.5: Log output of the test suite concerning *ReflexiveEquality*.

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 § 4.1.4.

— Notable start up time explanation

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 4.6 shows the output after modifying the generator.

— Modify generator, demonstrate failing case

```
[info] MoneySpec
[info] - should work with ReflexiveEquality *** FAILED *** (1 second, 278 milliseconds)
[info]   java.lang.AssertionError: assertion failed: expected CurrentState(Result,Initialised
    (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(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! **
```

Listing 4.6: Log output after modifying the generator

4.1.6 Test framework evaluation

— Evaluation points

The tests are generated based on the defined properties. After running the test framework, we evaluate the results and check what can be improved. We define the following criteria to evaluate the test framework after each improvement:

Coverage

— Tool used for determining the coverage

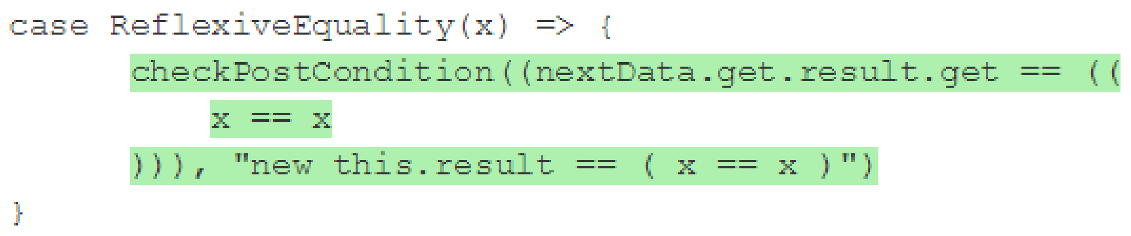
The test coverage of the generated system that is being tested by the test suite. To determine the coverage, we use an open-source library called *Scoverage* [?], which can create a report of the test coverage after running the tests. Since the generated system uses *SBT* as build tool, we use the open-source plug-in *sbt-scoverage*[1] to integrate the tool with *SBT*.

— Same specification for comparing results

For every evaluation, the same specification and generated system is used to determine the coverage. Note that the first experiment, for example, uses a smaller specification. While the second experiment separates the defined properties into two categories and added preconditions to one of the categories. In order to determine the coverage, the same specification will be used for both experiments, such that the generated system is equal and that the results from each experiment can be compared. In this example, the specifications of the secondexperiment will be used to determine the coverage of the first experiment.

— Which data exactly from reports

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. Since we are not sure how these paths are determined, we will not use this criterion for evaluation. Instead, 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.



```
case ReflexiveEquality(x) => {
        checkPostCondition((nextData.get.result.get == ((
            x == x
        ))), "new this.result == ( x == x )")
}
```

Figure 4.2: Test coverage example for *ReflexiveEquality*

— Property coverage

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.

— Won't reach 100%

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 criteria and take it into account.

— Not a hard criteria, still using for indication

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?

— Our approach, like QuickCheck

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 similar to *QuickCheck* in that we use random input values and test each property multiple times. But we use a *Rebel* specification and the generated system to check whether the properties hold when using the generator.

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:

— Combining all steps

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 Appendix A.

— Larger specification for experiments

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. However, when running the test framework this might still occur, which is a threat to this approach.

Accuracy

To evaluate the test framework we use coverage as a metric. *Scoverage* is being used to determine the coverage and to generate a report from it. The report could report the test coverage incorrectly,

— Possibly incorrect results, due to Scoverage

[1] https://github.com/scoverage/sbt-scoverage

causing it to be a threat for our evaluation.

We are using random data as input. Because of this, the results of the test coverage can fluctuate by small amounts in each run. However, we can still reason about the differences when there is a big difference between certain experiments. Additional to the coverage we used the number of bugs that were found as another metric. This metric depends on which system the test suite is being run and if the system already fixed the bugs that the test suite would find. It is also the case that after fixing the bugs that were found earlier, this metric can be seen as unnecessary, as it would result in 0 then.

One system

— Only one generator

Only one generator is being used throughout this thesis. However, it could 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. But 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.

— Probably causing compile errors

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

— ScalaCheck, not testing generator then

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.

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 if the generated test suite runs successful, rather we can conclude that the properties it checks for are satisfied.

Other components

The current setup is intended to be used to test a certain component of the generated system. Namely, the *Rebel* types and the operations among these. More properties can be added to check every type that *Rebel* supports. The test framework could also be extended such that it can also test other components of the system. Such as the sync block definitions, defining actions that should happen synchronously. Or performance measures when interacting with the data in the system, which uses a database implementation. Currently, such components are not being tested, while these components are also important for a bank. This is left as future work.

Although the test framework can be extended to also test other components of the generated system, it might not be possible to check all the components by using this approach. A generated system, for example, exposes its actions via a Rest API, which should be used to interact with the system. The tests that are done by using this approach do not check the implementation of this interface. Also, it is not able to check how, multiple generated systems would integrate with each other and if this is done correctly.

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 come to an answer to this research question: What kind of bugs can be found using this approach and how many?

—
Props to test cases, implication returns True

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.

—
Like QuickCheck, recall cycle shortly

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 5.7.

—
Termination, compile error due to library

Up arrow is incorrectly aligned in the listing due to layout

```
[error] MoneySpec.scala:316: overloaded method value * with alternatives:
[error]  (x: Double)Double <and>
[error]  (x: Float)Float <and>
[error]  (x: Long)Long <and>
[error]  (x: Int)Int <and>
[error]  (x: Char)Int <and>
[error]  (x: Short)Int <and>
[error]  (x: Byte)Int
[error] cannot be applied to (squants.market.Money)
[error]         Initialised(Data(result = Some(((((x * y)) * z) == (x * ((y * z)))))))
[error]                                                         ^
[error] MoneySpec.scala:441: overloaded method value * with alternatives:
[error]  (x: Double)Double <and>
[error]  (x: Float)Float <and>
[error]  (x: Long)Long <and>
[error]  (x: Int)Int <and>
[error]  (x: Char)Int <and>
[error]  (x: Short)Int <and>
[error]  (x: Byte)Int
[error] cannot be applied to (squants.market.Money)
[error]          checkPostCondition((nextData.get.result.get == (((((x * y)) * z) == (x *
    ((y * z)))))), "new this.result == ( (x*y)*z == x*(y*z) )")
[error]                                                                          ^
[error] two errors found
[error] (compile:compileIncremental) Compilation failed
[error] Total time: 79 s, completed 4-aug-2017 13:03:45
> Done testing
> ** Some tests failed! **
```

Listing 5.7: Log output first test run resulting in a termination.

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, as described in ??. Temporarily disabling this property allows the test framework to proceed further.

— Investigation, found property and var types

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.

— Failing tests, describing each

Property name	**x**	**y**	**z**
DistributivePercentage1	0.51	-311254801.77 EUR	-707194075.77 EUR
DistributivePercentage2	0.93	2089630160.75 EUR	-1316628389.49 EUR
DistributiveInt2	-883022216	-298435082.93 EUR	715725888.96 EUR
AssociativeMultiplicationPercentage2	840296462	1771903729.60 EUR	0.53
DistributiveInt1	-1790274467.41 EUR	1691684272	1449321647
AssociativeMultiplicationInteger2	-1852801029.34 EUR	-1309504561	1880170895
AssociativeMultiplicationPercentage1	-352883323.42 EUR	0.27	294211708

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

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).

Variable	**Value**	**Type**
x	0.51	Percentage
y	-311254801.77 EUR	Money
z	-707194075.77 EUR	Money
Formula	**Scala result**	**Expected result**
x*(y+z) == (y*x)+(z*x)	false	true
x*(y+z)	**-519408927.54539996 EUR**	**-519408927.5454 EUR**
(y*x)+(z*x)	-519408927.5454 EUR	-519408927.5454 EUR
y+z	-1018448877.54 EUR	-1018448877.54 EUR
y*x	-158739948.9027 EUR	-158739948.9027 EUR
z*x	-360668978.6427 EUR	-360668978.6427 EUR

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

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.

Variable	**Value**	**Type**
x	0.93	Percentage
y	2089630160.75 EUR	Money
z	-1316628389.49 EUR	Money
Formula	**Scala result**	**Expected result**
(y+z)*x == (y*x)+(z*x)	false	true
(y+z)*x	718891647.2718 EUR	718891647.2718 EUR
(y*x)+(z*x)	718891647.2718001 EUR	718891647.2718 EUR
y+z	773001771.26 EUR	773001771.26 EUR
y*x	**1943356049.4975002 EUR**	**1943356049.4975 EUR**
z*x	**-1224464402.2257001 EUR**	**-1224464402.2257 EUR**

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

DistributiveInt2

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 [?]. 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*.

Variable	**Value**	**Type**
x	-883022216	Integer
y	-298435082.93 EUR	Money
z	715725888.96 EUR	Money
Formula	**Scala result**	**Expected result**
(x*y)*z == x*(y*z)	false	true
(x*y)*z	-368477052257036740 EUR	-368477052257036762.48 EUR
x*(y*z)	-368477052257036796 EUR	-368477052257036762.48 EUR
y+z	417290806.03 EUR	417290806.03 EUR
y*x	**263524808260992384 EUR**	**263524808260992372.88 EUR**
z*x	**-632001860518029180 EUR**	**-632001860518029135.36 EUR**

Table 5.4: DistributiveInt2: Precision error when multiplying *Money* with an *Integer*

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 DistributiveInt2 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).

Variable	Value	Type
x	840296462	Integer
y	1771903729.60 EUR	Money
z	0.53	Percentage
Formula	**Scala result**	**Expected result**
(x*y)*z == x*(y*z)	false	true
(x*y)*z	789129950543366910 EUR	789129950543366877.856 EUR
x*(y*z)	789129950543366780 EUR	789129950543366877.856 EUR
x*y	**1488924434987484670 EUR**	**1488924434987484675.2 EUR**
y*z	939108976.688 EUR	939108976.688 EUR

Table 5.5: AssociativeMultiplicationPercentage2: Precision error causing bigger differences

DistributiveInt1

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 [?]. 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 [?], when this is not being prevented.

Variable	Value	Type
x	-1790274467.41 EUR	Money
y	1691684272	Integer
z	1449321647	Integer
Formula	**Scala result**	**Expected result**
x*(y+z) == (x*y)+(x*z)	false	true
x*(y+z)	2065907589620385223.57 EUR	-5623262698769382599.79 EUR
(x*y)+(x*z)	-5623262698769382599.79 EUR	-5623262698769382599.79 EUR
y+z	**-1153961377**	**3141005919**
x*y	-3028579159080673575.52 EUR	-3028579159080673575.52 EUR
x*z	-2594683539688709024.27 EUR	-2594683539688709024.27 EUR

Table 5.6: DistributiveInteger1: *Integer* overflows when using addition

AssociativeMultiplicationInteger2

For this case thrFor 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 [?], 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.

Variable	Value	Type
x	-1852801029.34 EUR	Money
y	-1309504561	Integer
z	1880170895	Integer
Formula	**Scala result**	**Expected result**
(x*y)*z == x*(y*z)	false	true
(x*y)*z	4561767263499657218201769467.30 EUR	4561767263499657218201769467.30 EUR
x*(y*z)	**3877394861172703799.94 EUR**	**4561767263499657218201769467.30 EUR**
x*y	2426251398546224819.74 EUR	2426251398546224819.74 EUR
y*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 [**?**].

Variable	Value	Type
x	-352883323.42 EUR	Money
y	0.27	Percentage
z	294211708	Integer
Formula	**Scala result**	**Expected result**
(x*y)*z == x*(y*z)	false	true
(x*y)*z	-28032049433190944 EUR	-28032049433190942.3672 EUR
x*(y*z)	-28032049433190948 EUR	-28032049433190942.3672 EUR
x*y	-95278497.3234 EUR	-95278497.3234 EUR
y*z	**79437161.16000001**	**79437161.16**

Table 5.8: AssociativeMultiplicationPercentage1: A precision error when using *Percentage*

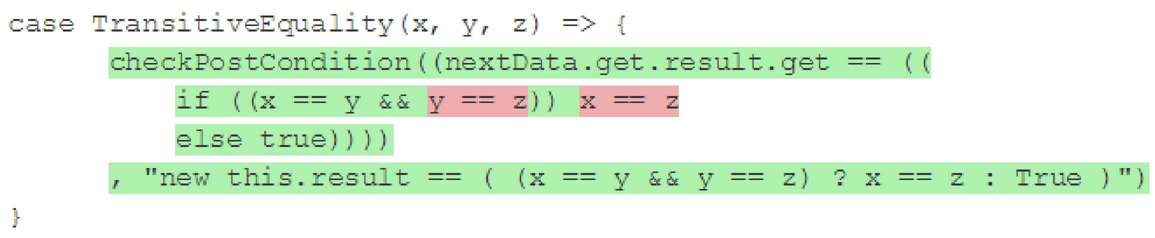
Additionally, we can see a difference in the results on the left-hand and right-hand side of the expression evaluation in *Scala*. Whereas the intermediate step for the left-hand side is calculated correctly. This also hints to the bug in the *Money* type which we already found when testing the *DistributiveInteger2* property. For the right-hand side, we cannot say this immediately, as there is already an error in the intermediate step.

This property revealed a precision error when the *Percentage* type is being used. The *Percentage* is being translated to a *Double* value, causing operations with it to have precision errors. In this case the *Percentage* is being multiplied by an *Integer*.

5.4 Evaluation criteria

Not tested implicative properties

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 due to the random values that are being used as input. An example of this is the *TransitiveEquality* property (`x == y && 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.

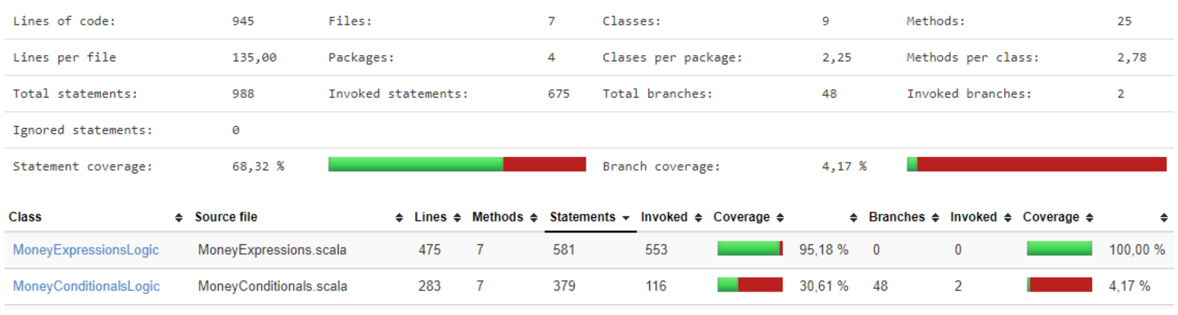


```
case TransitiveEquality(x, y, z) => {
      checkPostCondition((nextData.get.result.get == ((
          if ((x == y && y == z)) x == z
          else true))))
      , "new this.result == ( (x == y && y == z) ? x == z : True )")
}
```

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

Test coverage, 30% for implicative, 100% for others

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. This can be seen in Figure 5.2, where the test coverage of the implicative properties only covers roughly 30%. The files with a name ending with "Logic" contain the implementation of the properties, as well as the precondition checks. The test coverage concerning the other properties (those that do not use implication), reports 95% coverage. When investigating further, the other 5% are not related to the properties that we test, thus it is not required for this project to achieve 100% coverage on the Logic files.



Lines of code:	945	Files:	7	Classes:	9	Methods:	25
Lines per file	135,00	Packages:	4	Clases per package:	2,25	Methods per class:	2,78
Total statements:	988	Invoked statements:	675	Total branches:	48	Invoked branches:	2
Ignored statements:	0						
Statement coverage:	68,32 %			Branch coverage:	4,17 %		

Class	Source file	Lines	Methods	Statements	Invoked	Coverage	Branches	Invoked	Coverage
MoneyExpressionsLogic	MoneyExpressions.scala	475	7	581	553	95,18 %	0	0	100,00 %
MoneyConditionalsLogic	MoneyConditionals.scala	283	7	379	116	30,61 %	48	2	4,17 %

Figure 5.2: Test coverage report of the first experiment

The overall coverage of the generated system is 68%. Note that the libraries that are being used by the generated system are not included in the coverage report. We expect that this number can be improved by generating values such that the implicative properties are also triggering the if-clause. Which is currently not the case, as we have seen when checking the coverage of an implicative property in Figure 5.1.

— Overall 68%, cannot be 100, but can be improved

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 7 bugs. A compilation error, overflow/underflow errors and precision errors. Some precision errors originated from the library that is used for the *Money* type, for which we created an issue on *Github*[1]. An improvement of the test suite to also cover the implicative properties might result in more bugs that can be found.

— # of bugs, 7

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.

— Recap, found 7 bugs using this approach

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. 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.

— Implicative properties not tested

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. 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.

—
Fixed amount of tries not substantiated

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.

—
Multiple tries, each time 7 failed, assumed same causes

Unfixed issue

Unfortunately, the compilation error has not been fixed throughout this project however, it is an open issue on *Github*[2]. Some precision errors originated from a library used in the generated system, called *Squants* [?]. An issue was created covering these precision errors, which were fixed in the next release of that library. As for the overflow and underflow errors, these occurred when using the *Integer* type in *Rebel*. When using the *Integer* type, this might have been expected behaviour which causes this to happen. However, the generated system does not check whether this happens, nor does it prevent this. We consider the overflow and underflow errors as unexpected, as the *Rebel* language does not support other numeric types to hold a bigger value than an *Integer* supports.

—
Compile error not fixed

[1]https://github.com/typelevel/squants/issues/265
[2]https://github.com/typelevel/squants/issues/281

Chapter 6

Experiment 2: Smarter values generation

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.

— Context - result of experiment 1, what aim is now

6.1 Method

We can separate the properties that we have in 2 different categories: those using implication ($\Longrightarrow$) 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 is more useful. The random input values are being optimized such that the condition of the if-clause of these properties are being satisfied. For the other specification (*MoneyExpressions*), 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.

— 2 categories separation

In the *MoneyConditionals* specification, the condition to trigger the if-clauses will be added to the preconditions of each the event definition, 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 6.8 for example. Where the preconditions have been added to the event definition.

— Adding preconditions

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

Listing 6.8: 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 value tuples 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.

— Generating checks for preconditions

The first difference is that it now uses our custom generator to determine the input values, instead of the built-in Java random generator. A list of tuples, containing values which satisfy the

— Diff1: custom generator

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 6.9 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), converToExpr(intAmountY), convertToMoney(currency,
        moneyAmountZ)];
}
private default list[Expr] genTestValueForEvent(str eventName) {
    throw "genTestValueForEvent not implemented for event <eventName>";
}
```

Listing 6.9: Values generation for *Symmetric* and *Division1*, including the fall-back case.

Not very dynamic

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.

Mutating values with random operation

However, the values that are generated now are fixed when we use them directly in a test case, which completely removes the randomness of the values when running the tests. It would be better to keep the randomness, such that the values are different on each run. To solve this problem, we mutate the values in the list 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 6.10 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 6.10: 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 of the test.

— Small explanation about the new test case

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 6.8. This results in a failing case again in case the precondition was not met. When this happens, it could indicate that there's a problem with either our custom generator or in the generator.

— Also: else now returns false

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 6.11.

— Failing tests: Division

Property name	x	y	z
Division1	-16729.90 USD	830	-20.16
Division2	-44.68 USD	870	-38870.47

Table 6.1: Failing tests overview along with its input values

```
[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 6.11: 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 $x * y$ = *-44.68 USD* * *870* = *-38871.60 USD*. This should be equal to z. In fact, the input of z was slightly different, *-38870.47 USD*.

— Describe precision error happening at first

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 *-16729.90* USD.

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 6.12.

— Next (when fixed precision), division problem

Property name	x	y	z
Division1	1.5043478260... USD	-779	-1171.8869565217... USD
Division2	-3328.8254545454... USD	-129	25.8048484848... USD

Table 6.2: Failing tests overview, after fixing precision errors

[1]The decimals have been truncated for readability, Listing 6.12 shows the exact values

```
[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)
[info]  java.lang.AssertionError: assertion failed: expected CommandSuccess(Division2
    (-3328.8254545454545454545454545455 USD,-129,25.8048484848484848484848484848 USD))
    , found CommandFailed(NonEmptyList(PreConditionFailed(x == z*y)))
[info] - should work with Division1 *** FAILED *** (316 milliseconds)
[info]  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 6.12: Precondition failed error in *Division1* and *Division2*.

— Division2 triggers division problem

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. In Chapter 3, we defined that the precision in this case should be of X decimals. However, the test framework does not specifically check for this precision yet.

SRC division problem?

Define X

— Division1 triggers difference in rounding problem

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*y == z ⟹ x == z/y`.

Variable	Value	Type
X	1.504347826086956521739130434782609 USD	Money
Y	-779	Integer
Z	-1171.886956521739130434782608695652 USD	Money
Formula	**Scala result**	**Expected result**
x*y == z	true	false
x == z/y	false	false
x*y	-1171.886956521739130434782608695652 USD	-1171.886956521739130434782608695652**411** USD
z/y	1.504347826086956521739130434782608 USD	1.504347826086956521739130434782608 USD

Table 6.3: Division1: Difference in rounding

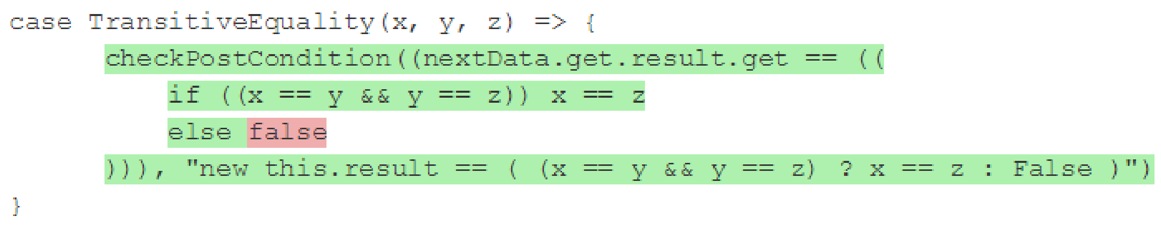
In the results, we can see that the expected values do not match the property either. Although in *Scala* the first expression is considered *true*. Since the expected results also return *false* for the intermediate calculations, the input values might not fully satisfy the condition to trigger the if-clause. Which could be an implementation error in our values generator. However, it's notable that in *Scala* the condition is considered to hold, which triggered this case. 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 § 6.1). Nevertheless, the results show that there is also an unexpected rounding going on when executing *x*y* in *Scala*. As the expected value contains some additional decimals compared to the result from *Scala*.

6.4 Evaluation criteria

— Property coverage

When looking at the coverage report concerning a specific property, it can be seen that the else-clause of the implication is not being triggered anymore. In Figure 6.1 the coverage of *TransitiveEquality* is shown, note that only the else condition (which was translated to *false*) is not triggered by the test suite. This was also the intention of the modification used in this experiment, as the if-clause is actually what we wanted to check in this experiment.

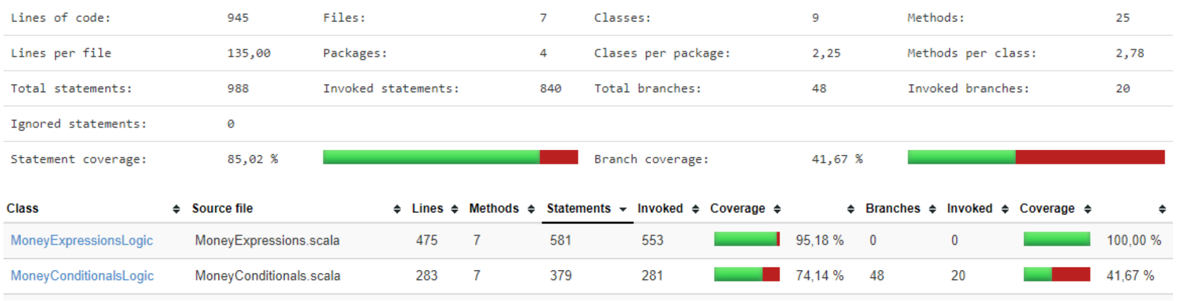


```
case TransitiveEquality(x, y, z) => {
      checkPostCondition((nextData.get.result.get == ((
          if ((x == y && y == z)) x == z
          else false
      ))), "new this.result == ( (x == y && y == z) ? x == z : False )")
}
```

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

— 74% on conditionals. Others remain the same - image

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. In this experiment these properties are triggering the if-clause of the properties using implication, thus we expect the test coverage to be higher when looking at the coverage of the properties. This also follows from the results, as we can see in Figure 6.2, the test coverage when looking at the logic file of the implicative properties is 74%.



Lines of code:	945	Files:	7	Classes:	9	Methods:	25
Lines per file	135,00	Packages:	4	Clases per package:	2,25	Methods per class:	2,78
Total statements:	988	Invoked statements:	840	Total branches:	48	Invoked branches:	20
Ignored statements:	0						
Statement coverage:	85,02 %			Branch coverage:	41,67 %		

Class	Source file	Lines	Methods	Statements	Invoked	Coverage		Branches	Invoked	Coverage	
MoneyExpressionsLogic	MoneyExpressions.scala	475	7	581	553		95,18 %	0	0		100,00 %
MoneyConditionalsLogic	MoneyConditionals.scala	283	7	379	281		74,14 %	48	20		41,67 %

Figure 6.2: Test coverage report of the first experiment

— Coverage, 85% (overall), better

The total test coverage on the generated system is reported to be 85%. As we have discussed in the evaluation of the first experiment, we do not expect to reach the 100% coverage, as there are certain components in the generated system which we do not test with this approach. Also, 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.

— # of bugs, 2 more

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 bug found was when using division with the *Money* type, which is expected to be rounded off at the Xth decimal . However, as we have seen in this experiment, the generated system does not take this into account. We can categorize this bug into one category, rounding errors. These are different from precision errors as the precision errors are caused by having an incorrectly calculated value, where rounding errors are basically not working with the rounding method that is defined.

Define X

Rounding method

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. For example: when dividing 1 by 3. 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, however, the rounding method is not taken into account here. Furthermore, it is not clear from the specification how this rounding should be done.

When defining the properties in Chapter 3, we said that a value should be precise. When rounding is needed, it must have X decimals . However, the test framework does not yet take this rule into account when testing these properties.

Define x

6.6 Threats to validity

Incorrect value generation

We have implemented a custom value generator to generate values for each test case. Furthermore, 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, which are expected to be correct values. When this is the case, the traceability of how the values were created is hard. This might affect the results or make some errors hard to trace back. We have seen this in this experiment.

Detecting precision

In this chapter, we triggered the division problem and found that this can cause problems in the generator. It's important to know what the expected result would be in this case. A possibility is to define this on the *Rebel* language or to make such rounding and precision definitions part of the specification. Currently, it is unclear what should happen in this situation. We concluded that the generator doesn't take the rules on precision and rounding, that we have defined in Chapter 3, are not taken into account. And that it currently uses a lower precision. The precision we defined is perhaps not expected for *Rebel* specifications, leaving it as a threat for this approach.

Dynamicallity

The implicative properties are now being tested such that the condition of the if-clause is being satisfied. However, the values generator that is being used for this is not very dynamic. As it simply checks for the event name and throws an exception in case this is not defined for the property yet. This means that adding new property definitions to the test framework requires a modification to the value generator in case of an implicative property. This makes the test framework less dynamic when adding new properties that should be tested on the generator.

— Not very dynamic

To fix this, it would be better to generate the values by interpreting the preconditions such that random values can be determined based on a certain condition. Since the *Rebel* toolchain already makes use of a bounded model checker to check a specification, this could be used to simply translate an expression and retrieve values for which the condition holds.

— Fix: using the preconditions

We have looked into this, by using the *Z3* solver. However, the solver always returns the same

— Checked using Z3, but returns same values all the time 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 another constraint, such that 100 unique values are being retrieved. But the problem still remains, as executing the same script multiple times results in the same values. When changing the seed of the random generator that is being used, it will return different values. In order to make the test framework execute this behaviour, the value generator has to be changed to integrate with the solver. Additionally, this could have a huge effect on time increase that the test framework needs to successfully finish.

There are other solvers available too, or other methods to generate values that match the condition. It would be useful to make the test framework more dynamic when such properties are being used.

— Other possibilities, future work

Implicative properties effectiveness

The use of implicative properties might not be as effective as using properties that do not. If the properties could be rewritten such that random values could be used to check the same thing, the implicative properties might be unnecessary. On the other hand, 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. If-statements and preconditions were not being used in the first experiment.

— Uneffective? Checking more though

Chapter 7

Experiment 3: Improving the value generation

...

7.1 Method

...

7.2 Results

...

7.3 Analysis

...

7.4 Evaluation

...

7.5 Threats to validity

...

Chapter 8

Discussion

In this chapter we discuss the research questions.

8.1 RQ 1: Which properties are expected to hold on the generator?

Rebel did not have any definitions of which properties are expected to hold on a specification. Since we are using property-based testing to check the generator, it was required to define the expected properties first. The definitions of each property can be found in Chapter 3.

Many properties were defined during this thesis, but these are certainly not all the properties that exist for *Rebel*. We focused on the *Money* type, which is considered the most important type for a bank. Additional properties can always be added to the test framework.

8.2 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 could be tested by using property-based testing. In order to check the generator, a *Rebel* specification was required. The generated system was used to determine the results, allowing to reason about the generator. The test setup could be divided into the following phases:

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

The first phase had to be done manually. For the other phases, we introduced our test framework to automate these steps in order to detect the bugs as automatically as possible.

8.3 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 9 properties that we have defined. Some properties triggered different kind of bugs. The bugs that were found can be separated into the following categories:

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

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

Precision errors: Errors causing an unexpected outcome value when being calculated.

Add division

Compilation errors

Compilation error, Squants issue

The fact that the test framework initially was being terminated was because of a compilation error. Although one assumption was that the generated system should be able to compile, another assumption we made was that the specification was consistent. The specification we created for all the properties is consistent, as *Rebel* did not report any syntactic or semantic errors with the type checker. The test framework is thus able to find such compilation errors. However, there can be many more compilation errors for which we do not check, which is also out of the scope of this thesis. The cause of this error was actually caused by an implementation error in an open-source library that the generated system used, called *Squants* [**?**]. To fix this, we created a *Github* issue[1] describing the problem. So that this can be fixed in the next release of the library.

Overflow/underflow errors

Overflow/unde errors, discussion and unclear definition

The overflow/underflow errors are caused because of the use of 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 [**?**]. 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 big numbers, or that a custom type must be used for this. Considering that *Rebel* does not provide another type for bigger numbers, the *Integer* type is considered to also hold bigger numbers. 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*.

Precision errors

Precision errors, Squants issue

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. Since we were able to reproduce the issue in a clean *REPL* environment, the problem existed in the open-source library, called *Squants*, that was used for the *Money* type. In order to solve this problem, we created an issue on *Github*[2] 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). So it is required to update this library in order to let these tests in our test suite pass.

Add division

[1] https://github.com/typelevel/squants/issues/281
[2] https://github.com/typelevel/squants/issues/265

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*. However, these definitions were not defined earlier, thus we defined properties of *Rebel* that are expected hold. We have found found some bugs in the generated system that were unknown before, by using the test framework that we created. 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*, we issued 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 we automatically test the generator in the *Rebel* toolchain, by using the generated system, to check whether the implementation works as expected?

A *Rebel* specification was created from the defined properties. Next, the existing generator was 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 in the generator by using the generated system. Additionally, we were able to detect a compilation error and incorrectly translated formulas. Although the latter was not the case, we showed that this can be detected when modifying the generator such that a formula is being translated incorrectly.

We conclude that this approach is a way how the implementation of the generator can be checked to satisfy certain properties that have been defined. However, it does not mean that there are no implementation errors in case the test framework finishes successfully. It only means that no implementation errors were found using this test framework.

9.1 Future work

Complete property definitions

— Full definitions for Rebel

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. These should be added to have a complete set of property definitions for the *Rebel* language. This is left as future work. When additional properties are known, these properties can be added to the test framework such that these will also be tested on the generator.

— Other properties/-components

Besides the mathematical properties, which have been focused on throughout this thesis, other properties could be defined too. This requires modifications to the test framework. As well as definitions of such properties, as these are missing currently. For example, it would have a valuable meaning for the bank to thoroughly check the properties of using a sync block in *Rebel* and verify that these also holds when a system is being generated by the generator.

Dynamic values generation

The value generator in the test framework that we have created for the implicative properties can be improved. Such that random input values are generated based on the actual preconditions that are defined. Our value generator was more intended as a prototype, to check whether this approach would work to find more bugs. The current value generator requires an update whenever an implicative property is being added to check. If the value generator would be able to use the preconditions to solve this issue, it doesn't require an update anymore for each implicative property that is being added.

This is gonna be Experiment 3, rewrite or remove later

— Improve value generator

Another way how this could be done is to use some tools to determine the random values. As we have discussed, the SMT solver could be used, but in our attempts, this resulted in the same values every time, thus we would lose the randomness of the values. Other approaches might be useful too to implement this functionality, such as concolic testing [?] or using SageMath [?].

— Other tools possible

Multiple generators

Throughout this thesis, we have only tested the system that is generated by 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 system.

— Only one, support for more could be added

Mutation testing

Another approach to test the properties on the generator would be to use mutation testing. The mutation coverage could then 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. Since *Scala* compiles to *Java* bytecode, mutating on bytecode level could be used to solve the problem of limited support. However, this could lead to false-positives, meaning that some mutants might not be relevant in that the modifications would not affect the implementation. This should then be taken into account.

SRC-PIT for mutation testing in Scala, limited support

Properties based on type checker

Rebel contains a type checker, which is able to determine exactly which operations are supported by using certain combinations of operators and types. This could be used to automatically generate the property specifications in *Rebel* based on whether a type supports certain operations that are required for the property. For example, if *Percentage* supports addition, the properties for additivity, associativeAddition and commutativeAddition can be generated based on this rule. Considering that a map of all operations can be created and a map of all existing types, each combination can be tested against the type checker. If the type checker allows determines it as a correct expression, the event definition for the property can be generated. Although this might result in many definitions that are being tested, with a possible overlap between them, it would be a way to automatically generate the properties that should be tested. This would assume that the type checker does the right thing, which would be a threat for this approach. This might also cause more compilation errors in the generated system, Nevertheless, it can make it more dynamic when a new type could be added to *Rebel*.

— Generating properties based on type checker

Appendix A

Property definitions of Rebel in Rebel

```
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.13: 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.14: 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.15: 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.16: The property definitions as Rebel specification (continued)