Testing Code Generators against Definitional Interpreters

Master's Thesis

Ioannis Papadopoulos

Testing Code Generators against Definitional Interpreters

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Ioannis Papadopoulos
born in Athens, Greece

Programming Languages Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

ING Group
ING Tech Research and Development
Amsterdamse Poort Bijlmerplein
Amsterdam, the Netherlands
www.ing.nl

Testing Code Generators against Definitional Interpreters

Author: Ioannis Papadopoulos
Student id: 4565002
Email: I.Papadopoulos-1@student.tudelft.nl

Abstract

Large companies suffer from the increasing complexity that exist in their software systems. Evolving their software becomes even harder if we consider that a change in one system can affect several other parts of their software architecture. Especially banks that need to be always complied with regulations, have to constantly make changes in their software to reflect these changes. ING is a primary example that currently tries to find a solution to these problems through the use of model driven development and more specifically code generation. In particular, they have created a Domain Specific Language called Maverick to specify the requirements/business logic and through the usage of code generators to automatically generate their entire codebase from these Maverick specifications. Code generators as any other software artifact is not bug free, meaning that testing code generators is of paramount importance. However, testing code generators is not straightforward as their output is another program that besides syntactic structure also has behavior. Many formal approaches have been developed that try to formally prove the correctness of code generators. Nevertheless, the complexity and scalability issues that these approaches face make them infeasible in practice. This thesis presents a testing approach that leverages a definitional interpreter to test code generators. We evaluate and show the practicability of our approach using Maverick specifications developed by ING and we conclude that our proposed method can address many of the issues that formal approaches face.

Thesis Committee:

Chair:	Prof. Dr. E. Visser, Faculty EEMCS, TU Delft
University supervisor:	Dr. S. Erdweg, Faculty EEMCS, TU Delft
Committee Member:	Dr. M. F. Aniche, Faculty EEMCS, TU Delft

Preface

When I first decided to work on this project I knew that it will not be easy as I needed to work hard to catch up with some of the background that I was missing. However, I believe that all the things I learned during this journey is the biggest reward that I could get and what exactly I was aiming for my master thesis. In this master thesis, there were many people involved who deserve many thanks because without their support and guidance this thesis would not have been possible.

First of all, I would like to thank Eelco Visser for welcoming me in the Programming Languages research group and my supervisor Sebastian Erdweg for all his guidance and supervision during this master thesis.

Furthermore, I would like to thank ING for giving me the opportunity to work on this project. Also, I would like to thank my colleagues in ING, namely my company supervisor Jorryt-Jan Dijkstra, for his help throughout this thesis. Special thanks also to Miguel, Kevin and Robbert for sharing their knowledge and for the insightful discussions that we had regarding the semantics of the Maverick DSL.

Last, but certainly not least, I would like to thank my friends and girlfriend for their encouragement and support throughout the low and high points of the thesis. Finally, I want also to thank my family and especially my parents, without them not only this thesis would not have been possible but also this master and the person I am today would not have been possible. Through their unconditional support they encourage me to pursue and excel not only in the things that I love to do but also as a person.

Ioannis Papadopoulos
Delft, the Netherlands
November 26, 2018

Contents

List of Figures

Listings

Chapter 1

Introduction

Large companies suffer from insurmountable complexity and an astonishing amount of legacy that exist in their software systems. Evolving the software becomes extremely difficult since making a change in one of the systems that constitute the software architecture of a company can affect a lot of other systems that are also part of the architecture.

In addition to that, the strong coupling that exists between the requirements and their implementation not only prevents the evolution of the software but also makes extremely hard to distinguish any requirements from the code and trace them in the implementation. For example, a change in technology would result to reverse engineer these requirements in the already existing implementation and then re-implement them according to the new technology being used. As a result, companies, such as banks that constantly need to change the requirements of their software in order to comply with changes in laws or regulations, have to face with the same problem, that is, to reverse engineer the requirements in the implementation.

ING, a large globally operating bank, is one of the companies that suffer from these problems. In particular, figure 1.1[1] is an actual representation of a subset of the current software architecture within ING where it become obvious that a single change (red star) might affect several other systems (yellow stars).

The aforementioned reasons drive large corporations to search for alternative ways of managing their codebases [8, 32]. To abstract away this complexity, a promising solution that currently has attracted much interest in the research community is *model driven engineering* [17, 18, 31]. In model driven engineering developers focus their efforts not in developing code but in building models in which they try to capture all the functional requirements of the software that they want to generate. Subsequently, they can use these models to transform them into new models or code. The process of translating models into a programming language is called code generation and the software artifact that is developed to perform this translation is known as *code generator*. Essentially a code generator can also been seen as a compiler that translates a source language to a target language.

One way that is widely used nowadays in order to create these models is through the use of *Domain Specific Languages* (DSLs). These are small, high-level and expressive

[1] The image is blurred because of sensitive information

Figure 1.1: Visual representation of a subset of ING's current Software Architecture

programming languages which usually are tailored to a specific problem domain [51]. These characteristics enable not only developers, but also domain experts with limited experience in coding, to express their ideas in a higher level where implementation details are not important.

However, as with other software development artifacts, code generators are not free from bugs and, thus, to introduce code generation into production applications, it is essential to have a high degree of confidence in the code generator's reliability and correctness. Therefore, rigorously testing code generators becomes inevitable, especially when they are used in the development of safety critical software [20].

Nevertheless, validating that a code generator transforms correctly a model into source code is inherently more complex than validating any other individual software artifact [2]. To illustrate, a code generator's inputs and outputs are complex objects and not just values or event sequences as would be in the case of a typical software artifact. In particular, the input of a code generator is a model expressed in a high level modeling language and its output is another program which purpose is to preserve the behavior of the model. This means that the output is a complex data structure that has also behavior and, hence, verifying that the output program is the correct one is non-trivial as one has to show that the output preserves the semantics of the model.

Many researchers have focused on developing techniques in order to improve code gen-

erators' reliability and robustness. According to Sturmer et al. [48] these techniques fall apart in two categories:

- *constructive procedures*, which are related with the adoption of standards and guidelines
- *analytical procedures*, which are related with verifying and testing the code generator

The procedures that fall into the first category assure that a tool has been developed according to a systematic development process [48] such as the Software Process Improvement and Capability determination (SPiCE). Although, these procedures can guarantee that a code generator has been developed with correctness in mind, do not assure that a code generator is bug free or that any existing errors could be detected. Therefore, testing or verifying the code generator still is of paramount importance.

A strong approach that researchers have extensively studied in recent years, and falls in the second category, is to formally prove the correctness of compilers/code generators [6, 10, 23]. However, the industrial benefit of these verification approaches could not yet be shown, this can mainly be attributed to the following two reasons. Firstly, code generators are usually being developed in an environment that constantly faces technological innovation which, subsequently, leads to new versions of the modelling language and code generator to appear in shorter cycles [49]. The aforementioned reason along with the increasing complexity that theoretical approaches address, make them infeasible in practice.

Another approach that falls into the second category, as described by Sturmer et. al. [48], is testing. Despite that testing can only provide partial guarantees about the reliability of a software artifact, still remains one of the most used techniques in practice. A widely used approach in testing is *unit testing* [3]. Typically, in unit testing a test designer specifies test cases for which he checks whether the *system under test* (SUT) behaves correct. For each run of these text cases the outputs produced by the SUT are compared against the expected outputs that also have been specified by the test designer. Having said that, when applying this software testing method to a code generator the output is source code, so ultimately the comparison between the expected output and the actual output would be a string comparison, meaning that unit testing in that case focuses mostly in the syntactic aspects of the generated code. Thus, unit testing constitutes a weak approach for testing a code generator.

Consequently, several studies stressed the importance of testing generated code's behavior when it is executed rather than its concrete syntax [9, 37, 42, 47, 49]. The technique that is used in these approaches is *back to back testing* [52]. In this type of testing, two or more software systems, that are supposed to respond with similar results, are tested with the same input stimuli and the produced results are then compared to each other to confirm the correctness of the systems under test. For the case of code generators this means that the translation carried out by a code generator is tested through the execution of both the source model and generated code. More specifically, both executions produce traces from which the traces of the source model play the role of the expected output against which the traces of the generated code are tested.

For modeling languages and more specifically DSLs a description of their execution semantics is needed in place of the expected output to make feasible the testing approach that we just described above. Several approaches exist that try to formally describe the semantics of such languages [43, 35, 15] in order to avoid their ambiguity. Another way to produce a concise definition of the semantics of a programming language is through the usage of a *definitional interpreter* [38]. One of the advantages of this approach is that the interpreter can be executed which allows to observe the behavior of specific programs of the defined language.

In this thesis, we exploit the idea of definitional interpreters and we present a method that focuses on testing code generators from a semantic perspective. In a similar way with the previous studies we use a back-to-back technique established on execution semantics. However, in contrast to these studies we use a definitional interpreter to provide us with a specification of the expected output. Subsequently, we check if the execution of the generated code conforms to this output. This lead us to the main question of this thesis: *how could Code Generators be tested against a definitional interpreter?*

Moreover, we develop and evaluate this method within ING. In particular, ING is a prime example who are researching whether model driven engineering is the solution to their problems. They have created a Domain Specific Language (DSL), called *Maverick*, based on *state machines* in order to capture the requirements and formalize their business logic in these DSL specifications. Using this model their aim is to automatically generate their entire codebase. The method described in this work entails the development of a definitional interpreter for the *Maverick DSL*. Furthermore, we automatically generate input stimuli, capable of traversing different paths of a state machine, for both the generated code and the definitional interpreter. Finally, the traces produced by the execution of the generated code are tested for conformance to the traces produced by the definitional interpreter.

This thesis continues as follows: in Chapter 2 we give an overview of the approach followed along with necessary details for the DSL and code generators developed by ING. We then dive into the development of the definitional interpreter in Chapter 3. In Chapter 4, we first give some background information on the problem of generating test sequences from state machines and then we continue with the description of the approach followed in this work to tackle this problem. Afterwards, in Chapter 5 we evaluate our method using experimental state machines that were created by ING. In Chapter 6 we discuss related work and how this differs from our approach. Finally, Chapter 7 concludes this thesis, and suggests some ideas for future work.

Chapter 2

Testing approach

This chapter gives a high-level description of the testing approach that we implemented. We describe the different steps our testing approach entails in order to reliably test ING's code generator. Subsequently, we present ING's research project regarding code generators and introduce the DSL named Maverick that they have created for that purpose. This will enable the reader to better understand the next chapters.

2.1 Conformance testing of Code Generators

According to Stürmer et al. a code generator can be assumed that is working correctly if invalid test models are rejected and the code that is generated by the translation of the valid test models preserves the behavior of the model [49]. Our approach focuses on testing the second characteristic: whether the generated code conforms to the model's behavior.

To realize that, we employ a similar approach to *back-to-back testing* based on the execution semantics. The back-to-back testing entails the execution of both generated code and model, and then the comparison of their execution traces. Therefore, this approach requires the generation of test cases that will steer the testing of the code generator and will be identical for both the generated code and model. Testing can only give us confidence that a software artifact is correct up to a certain degree, and that is related to the specified test cases. Hence, it is important to generate test cases that will make the testing of the code generator as reliable as possible. In the case examined in this work, the model is represented as a state machine so our test cases should be, at least, capable of traversing the different paths of a state machine.

In addition, to perform conformance testing of the code generator we need a specification of the behavior captured in the model. We accomplish that by using a definitional interpreter which gives us a concise definition of the model's execution semantics.

Finally, the execution traces of the generated code should be checked for their correctness against the traces produced by the definitional interpreter. However, as Stürmer et al. argue, the traditional notion of correctness cannot be applied to code generators and therefore "a notion of sufficiently similar behavior" should be used instead [49]. Thus, our approach is based on this notion.

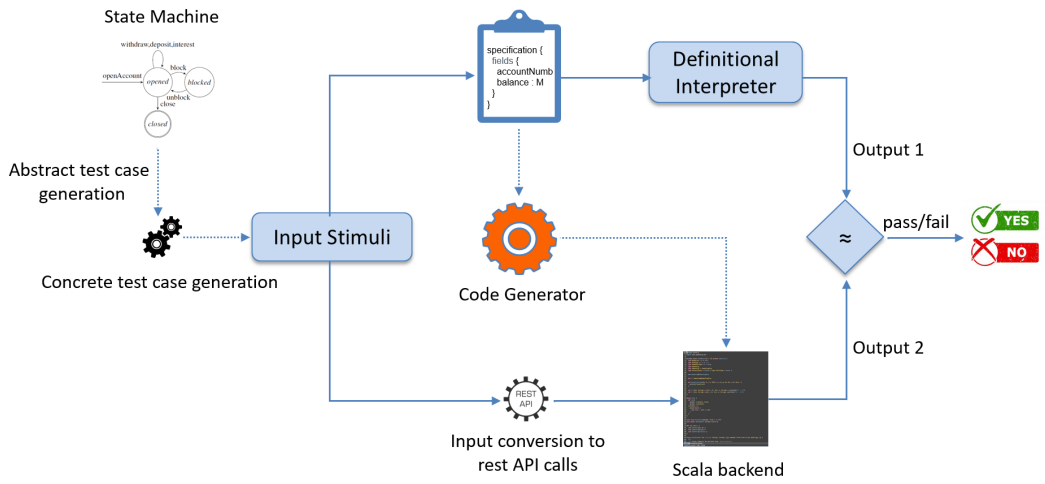


Figure 2.1: Conformance testing of the Code Generator

In more detail, as depicted in 2.1, our approach consists of the following four main parts:

- *Test Case Generation*: The first step of our approach is to generate test cases, capable of exploring different paths of our state machine model, that then will be given us an input to both the Definition Interpreter and generated code. In particular, the procedure followed in this work can be divided in the following two components:
  - *Abstract test case generation*: The aim of this process is to create paths on the model level. More specifically, as state machines are extended graphs, graph traversal algorithms can be used to find different paths that exist in a state machine. Therefore, we first convert our model to a directed graph and then we use a modified version of breadth-first search algorithm to find all the paths, up to a certain length, that exist between the origin node and a final node of a state machine. Then these paths can be used as abstract test cases which are only missing the parameters that actually make the traversal of the identified paths feasible.
  - *Concrete test case generation*: In this step, the previous generated paths, which basically contain only the names of the transitions that need to be taken, get concretized by generating actual values in a pseudo-random way which fulfill the conditions that enable the transition from one state to another.
- *Definitional Interpreter*: We use it in order to define the execution semantics of the modeling language. Consequently, the interpreter can be executed, after taking the test cases generated before as input, which enable us to observe the behavior of specific programs of the modeling language. This is simply an interpreter written using the Scala programming language which has well understood semantics and

provides us with good support to represent programs and define evaluation functions over them.

- *Execution of the generated Scala backend*: In order to execute the generated code, the test cases generated in the first step, are converted to rest API calls which perform requests to the generated Scala backend.

- *Comparison of the different traces*: Lastly, the traces produced by the execution of the generated code are tested for conformance to the traces resulted from the definitional interpreter. The comparison takes into account the notion of *sufficiently similar behavior* for time related properties of the model and generated code.

2.2 ING's Code Generator

Currently, ING is researching whether model-driven engineering, and more specifically code generation, is the solution to the increasing complexity of their software systems. Their main goal is to automate the software development process by capturing all the requirements in a model and from this model to generate software that complies to all these requirements.

In other words, their vision, as depicted in figure 2.2 is a software development process where a domain expert writes the specifications that encapsulate all the requirements of the software to be build, then a code generator takes as an input these specifications to generate the software artifacts that are required for the software to be deployed. Subsequently, after the deployment of these artifacts, users can give their feedback based on their experience. If the user experience feedback is negative, their goal is to only have to re-write the specifications and then generate the software again.

Moreover, this process indicates their purpose of not touching at all the generated code, which makes the testing of the code generator of even bigger importance.

Currently, as can be seen in figure 2.3 they generate multiple things from those specifications. More specifically, they generate visualizations for the business but also visualizations that can be used for developers and documentation which includes experiments with generating documents in English and Dutch. In addition, they generate a *Scala* backend application with a RESTful application program interface (API) in order to implement these specifications. That API is documented in *Swagger* which is typically a documentation of an API that can potentially be leveraged to generate compliance. Finally, they generate mobile apps and a portal that consume these APIs. Our main focus in this thesis is to test the Scala backend application which implements the model and constitutes the basis of other generated artifacts.

2.2.1 Maverick DSL

In this section we dive into the domain specific language named *Maverick* that ING have created to capture their business logic. Maverick is a declarative language therefore in this work we will be talking about *Maverick specifications* and not *Maverick programs*.

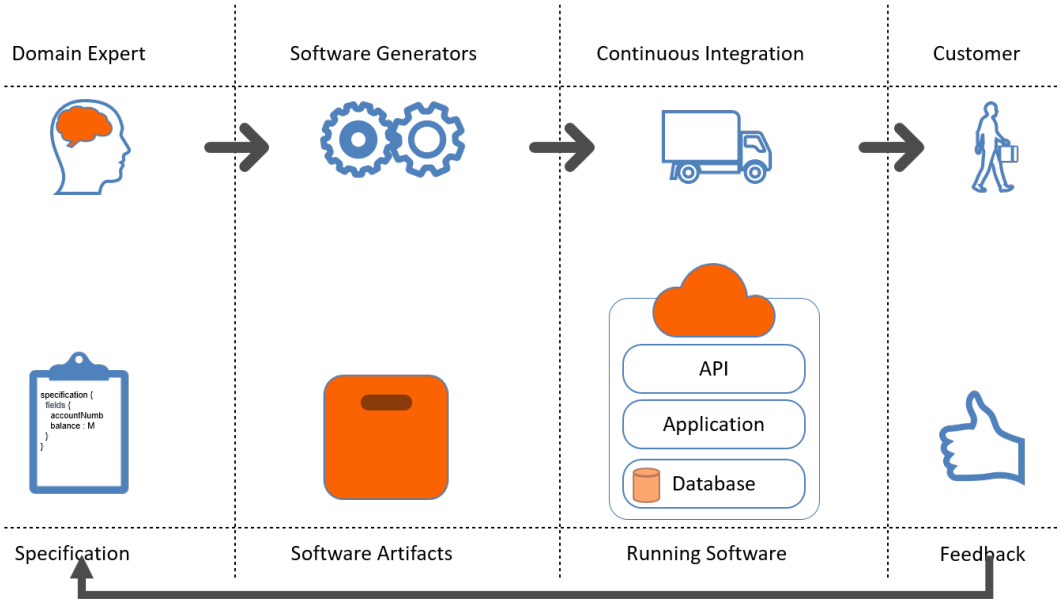


Figure 2.2: Software development process with Code Generators

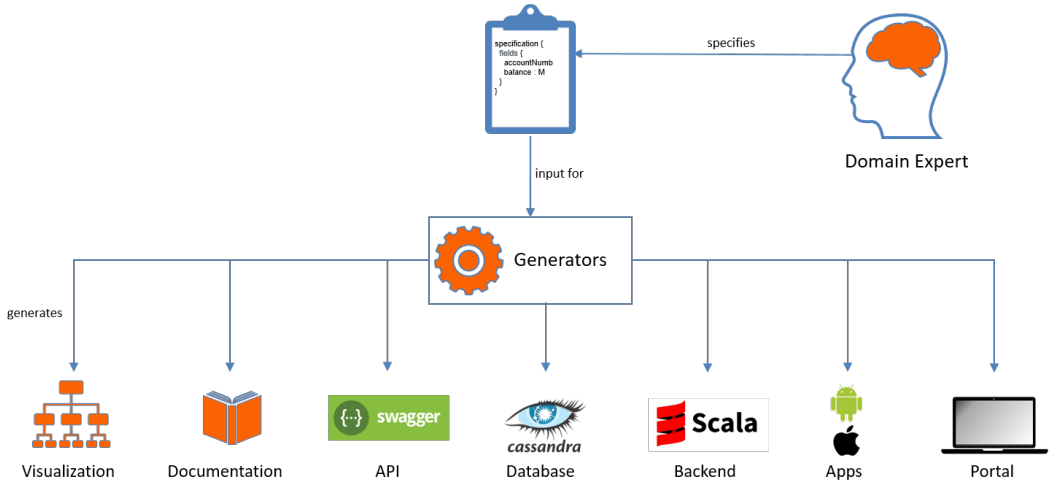


Figure 2.3: ING's Code Generators

Maverick specifications use a transition system (also known as state machines) as their core. A maverick specification can be seen as a definition of a state machine that describes its states and transitions with some extra annotations such as invariants.

An example of how a Maverick specification is formatted is given in listing 2.1, this listing illustrates a simple account specification. A maverick specification starts with a module definition which can be used by other files in order to import this specification. Then we can import other specifications files or a library file which contains function declarations.

After the import statements, the definition of the specification follows. A specification

consists of the following five main blocks:

- *fields*: Contains the definition of one or more fields. It should be noted here that a field can also be a reference to another specification.
- *identity*: This block defines a mandatory identity of at least one of the fields. The identity is globally unique and identifies an instance of a specification.
- *invariants*: Includes the definition of predicates that must always hold during the execution of a transition. They are named and can refer to other fields of the specification.
- *lifeCycle*: The life cycle defines which transitions can be taken within this state machine and the source, destination states after a specific transition is taken. This can be seen in the listing 2.1 in the form of: *source –> destination: transitionName*. In addition the keyword *initial* is a "magic" source state, an instance of a specification that does not exist yet comes from this state. Finally, the keyword *final* denotes that a state is final which means that the state machine is finalized and no more transitions can be taken.
- *transitions*: The transitions block defines the transitions that can be made. Each transition consists of a pre and post conditions block. These two blocks contain conditions that act as guards for the transition. They can contain configuration parameters which are optional and/or parameters that are mandatory. Finally, the notation <*variable*> *'(prime)* that can be seen inside a post-conditions block refers to the "new" (i.e. post-transition) expression value.

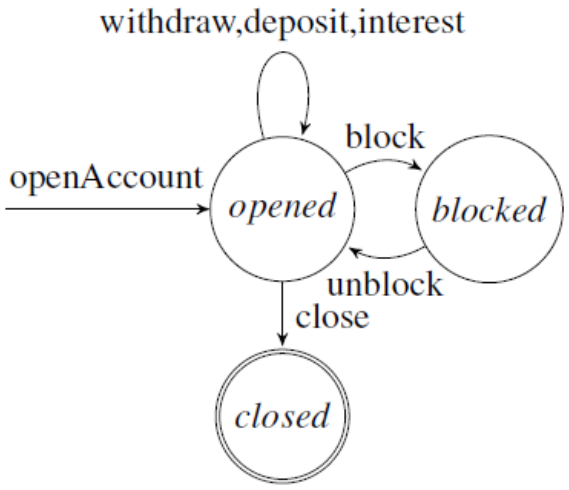


Figure 2.4: Graphical representation of a simple account specification

Each one of the aforementioned blocks can contain a block that starts with *@doc*, this block is optional and only serves the purpose of documenting or explaining something that is not clear.

Moreover, there are a number of types defined within this language, some of them are tailored specifically for the banking domain. Such examples are, the *Money* type (eg. line 14 in listing 2.1) which is a monetary value like *EUR 50* or *USD 22.35* and *IBAN* type (eg. line 13 in listing 2.1) which is a well formed IBAN such as *NL03INGB0672370210*.

An important aspect of the Maverick DSL is the capability of interaction between specifications. As mentioned before a transition of a specification contains pre and post conditions that act as guards for this transition. All these are boolean expressions that define a predicate that should hold in order for the transition to be successful. However, these expressions can also refer to other specification instances and read their properties. Accordingly, these properties can be used as if they were local fields and hence be part of any expression.

An explicit exception of what we mentioned above is the case of invoking transitions in external specifications (an example of a specification with external invocations can be seen in line 56, 57 of the listing 2.2). More particularly, if you have a reference to another specification instance, you can invoke a transition on it. In the case of an external invocation, in order for the primary transition to be valid the secondary transition must first succeed. This means that any implementation of a Maverick specification must enforce that those transitions will be taken atomically in respect of the perspective of the outside world. In other words, if a secondary transition fails then the whole initial transition fails too.

As can be seen in the graphical representation of a simple specification account (figure 2.4), it might be possible to have a current state which would allow two or more transitions to be taken. This possibility for multiple destination states or non-determinism is resolved by the fact that Maverick is actor-based. This means that any transition or change is initiated by an actor which basically enables non-deterministic options between transitions to become deterministic in practice.

Finally, to get a better understanding of the semantics of the Maverick language consider the following sequence of events, expressed in natural language and related with the account and transaction specifications listed in 2.1, 2.2 (for the reader's convenience we give trivial names to IBAN values):

1. Open account with IBAN *A1* and initial deposit of *60 EUR*.
2. Open account with IBAN *A2* and initial deposit of *100 EUR*.
3. Start a transaction of *20 EUR* from account *A1* to account *A2*.
4. Book the transaction that started before.

This sequence of events will trigger a sequence of transitions in the Maverick specifications. For each transition, first the pre-conditions will be checked and if they hold then the transition will be executed. Following, the post-conditions (along with the invariants) will be checked using the post-transition expression values explained before (eg. line 48 in

listing 2.1). If they also hold then the transition is successful. In the given events all the conditions hold, thus subsequently a sequence of state changes will be triggered in Maverick instances as following:

1. The first event triggers the transition *openAccount* in listing 2.1 (line 43). This transition creates a specification instance of an account. This instance of an account will be in state *opened* with account number equal to A1 and a balance of 60 EUR.

2. The second event also triggers the transition *openAccount* in listing 2.1 (line 43). This transition creates another specification instance of an account which will be in state *opened* with account number equal to A2 and a balance of 100 EUR.

3. The transition *start* in listing 2.2 (line 33) is triggered. A specification instance of a transaction is created that will be in state *validate* with the amount being transferred from account A1 to A2 equal to 20 EUR.

4. The transition *book* (listing 2.2 line 53) is triggered and this signals two external transitions. First a *withdraw* transition takes place for the account A1 and then a *deposit* transition occurs for the account A2. After the successful execution of these transitions the instance of a transaction is now in its final state, namely *booked*. In addition, the account A1 is in state *opened* with a balance of 40 EUR and account A2 is in state *opened* with a balance of 120 EUR.

```
1  module simple_transaction.Account;
2
3  import simple_transaction.Library;
4
5  @doc {
6    This is a specification of an Account.
7    The account can be opened, blocked and closed and can never be
         overdrawn.
8  }
9
10 specification {
11
12   fields {
13     accountNumber: IBAN;
14     balance: Money;
15   }
16
17   identity {
18     accountNumber;
19   }
20
21   invariants {
22     @doc {
23       The balance should always be positive.
```

```
24      }
25      positiveBalance {
26        this.balance >= EUR 0.00;
27      }
28    }
29
30    lifeCycle {
31      initial -> opened: openAccount[minimalDeposit = EUR 50.00];
32      opened -> opened: withdraw, deposit, interest;
33      opened -> blocked: block;
34      blocked -> opened: unblock;
35      opened -> closed: close;
36      final closed;
37    }
38
39    transitions {
40      @doc {
41        Opening an account needs a valid IBAN and some initial
              deposit.
42      }
43      openAccount[minimalDeposit: Money = EUR 0.00](initialDeposit: Money) {
44        preconditions {
45          initialDeposit >= minimalDeposit;
46        }
47        postconditions {
48          balance' == initialDeposit;
49        }
50      }
51
52      @doc {
53        Withdraw money from the account.
54      }
55      withdraw(amount: Money) {
56        preconditions {
57          amount > EUR 0.00;
58
59          balance - amount >= EUR 0.00;
60        }
61        postconditions {
62          balance' == this.balance - amount;
63        }
64      }
65
66      @doc {
67        Deposit money on the account.
68      }
69      deposit(amount: Money) {
70        preconditions {
71          amount > EUR 0.00;
```

```
72        }
73        postconditions {
74          balance' == this.balance + amount;
75        }
76      }
77
78      @doc {
79        Block the account for withdrawals and deposits.
80      }
81      block() {}
82
83      @doc {
84        Unblock the account so that withdrawals and deposits can
              happen again.
85      }
86
87      unblock() {}
88
89      @doc {
90        Close the account.
91      }
92      close() {
93        preconditions {
94          this.balance == EUR 0.00;
95        }
96      }
97
98      interest[maxInterest: Percentage = 10%](currentInterest: Percentage) {
99        preconditions {
100         currentInterest <= maxInterest;
101       }
102       postconditions {
103         this.balance' == this.balance + singleInterest(this.balance,
              currentInterest);
104       }
105     }
106   }
107 }
```

Listing 2.1: An example of a Maverick specification that specifies a simple account

```
1  module simple_transaction.Transaction;
2
3  import simple_transaction.Library;
4  import simple_transaction.Account;
5
6  @doc{
7    This is a specification of a Transaction.Via a transaction money
```

```
      can be transferred between two accounts.
}
specification {
  fields {
    id: Integer;
    amount: Money;
    from: Account;
    to: Account;
  }

identity {
    id;
  }

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

transitions {
    @doc {
       Start a new transaction.
    }
     start(amount: Money, from: Account, to: Account) {
      preconditions {
        @doc{ From account must exist.}
        from is initialized;
        @doc{ To account must exist.}
        to is initialized;
        to != from;
        amount > EUR 0.00;
        amount.currency == EUR;
      }
      postconditions {
        amount' == amount;
        from' == from;
        to' == to;
      }
    }

    @doc{
       Book the transaction.
    }
    book() {
      preconditions {}
      postconditions {
```

```
56          this.from.withdraw(this.amount);
57          this.to.deposit(this.amount);
58        }
59      }
60      ...
61    }
62  }
```

Listing 2.2: An example of Maverick specification that specifies a transaction between two accounts

Chapter 3

Definitional Interpreter

This chapter introduces the main building block of our approach, the *definitional interpreter* of the Maverick DSL. We first give some information on approaches that focus on defining the behavior of a language and subsequently we explain what a definitional interpreter is and what is necessary to know in order to understand the implementation of a definitional interpreter. Finally, we dive into the implementation details of the definitional interpreter developed for the Maverick DSL.

3.1 Background

Before start developing an implementation of a language, either this is a compiler or an interpreter, it is first essential to know how a language is supposed to behave. Although the syntax of a language give us information on how we can write a particular language and which particular symbols we can use for that purpose, it give us limited information about the meaning of these symbols and how they are supposed to work.

The meaning of the programming languages, often referred as semantics, can be divided in to two main levels: static semantics which refers to restrictions of the set of valid programs and dynamic semantics which refers to the run-time behavior of a program. In this section, we are specifically interested in the execution semantics of a programming language and which approaches exist that help us to grasp a better understanding of their behavior.

One way that is often used to describe the behavior of a general purpose programming language or a DSL is through the usage of natural languages (eg. English). However, using a natural language to describe the meaning of a programming language leads to problems related with the ambiguity and imprecision of the natural languages.

Therefore, beyond natural languages, formal approaches emerged to aid understanding of programming languages. These formal approaches usually use some sort of mathematical formalism which makes their notation more precise and less ambiguous than natural languages.The three main approaches to formal semantics, which also other approaches use or combine in some way, are the following:

- *Operational semantics*[35]: The definition of a language is given by *reduction rules*

which describe how a program from its initial state is transformed step by step into its terminal state.

- *Denotational semantics*[43]: The definition of a language is given by mapping a program into a mathematical object that is considered to represent its semantics.

- *Axiomatic semantics*[15]: The definition of a language is given by using assertions that can be proven in some logic, and which describe what conclusions can be made for the input or output of a program. Basically, this approach specifies what can be proven about a program.

Another approach for defining the semantics of a language is to write a definitional interpreter for a given language. A definitional interpreter, as introduced by Reynolds [38], is a program which aims not only to give an implementation of a specific language but to provide a definition of this language, even if this means that will have to sacrifice efficiency for the clarity and simplicity of the definition. Reynolds showed that it is possible to write a definitional interpreter that defines the semantics of the object language by relying on the semantics of the host language. In other words, a *definitional interpreter* is an interpreter written for one programming language using another one with the goal to implicitly specify the execution semantics of the object language.

Moreover, Reynolds with his seminal study made a synopsis of earlier work where he classified definitional interpreters based on two key characteristics: whether the defining language is a *higher-order language* and whether the defining language uses a *call-by-value* or *call-by-name* evaluation strategy. A programming language is higher-order if functions in that language can occur as values, as arguments in other functions or as results of other functions. In a language that use a call-by-value evaluation strategy expressions that are passed as arguments to functions are first evaluated and then the value is bound to the corresponding variable in the function. On the contrary, in a call-by-name evaluation an expression passed as an argument in a function is substituted in the function's body and only evaluated when they used in the function, in case an argument does not appear in the function body then it never gets evaluated and if it appears several times then each time is getting evaluated again.

In more detail, a definitional interpreter operates over an abstract syntax tree (AST) representation of an object program and calculate its value. An AST is a structural representation of the source code in tree form which is based on the syntax of a certain programming language, but without caring about irrelevant syntactic details (eg. parenthesis, semicolons etc).

Definitional interpreters are usually structured as recursive traversals over the aforementioned AST representation. An AST typically is modelled through the usage of algebraic data types which basically are inductive data structures that can be recursively defined. This allows the definition of a match case for each language constructor and subsequently the recursive invocation of the interpreter on the sub-trees of this constructor. Therefore, functional languages such as Scala, which provide us with algebraic data types and pattern matching constructs, are particularly appropriate for implementing a definitional interpreter.

In that way, a definitional interpreter defines the behavior of the language constructs and enables us by its execution to observe the behavior of specific programs of the defined language. In addition to that, by using as defining language one that has well-understood semantics we can produce a clear definition of the semantics of the object language which is easy to understand for any developer that knows the defining language.

3.2 Implementation of the Definitional Interpreter

As a first step of our testing approach, we implemented a definitional interpreter for the Maverick DSL. The intention behind the definitional interpreter is to act as a simulator for the Maverick specifications and provide us with an expected output. Our implementation was done in Scala and we made heavy use of algebraic data types (e.g. case classes) and pattern matching. Furthermore, throughout our implementation we tried to maintain a functional programming style by preferring immutable variables over mutable ones and recursion over loops.

First of all, to implement a definitional interpreter for a target language, as we explained in section 3.1, is necessary to have a structural representation of the object program which is usually given by an AST. This can be realized through the implementation of a parser which takes as input a string representation of a program in the target language and produces a structural parse of the object program by breaking it into its language constructs. However, in our case we will not need to build our own parser us ING have already built a parser for the Maverick DSL which is used for the code generation process. In particular, the input to our interpreter will be an AST intermediate representation of a Maverick specification, which has passed all the necessary checks related with whether a specification is well formed or not.

The starting point of our implementation is the function *simulate* as illustrated in the listing 3.1. Some observations and explanations, supplementing listing 3.1, are presented in the following list:

- As we described in section 2.2.1, Maverick DSL has two important characteristics: first, is an actor based language which means any transition or change is initiated by an actor and second, allows the interaction between different specification instances through the usage of external invocations. This explains the signature of the *simulate* function which take as arguments a program which basically is a set of Maverick specifications, a list of events which refer to these specifications, and a data structure named *CurrentState* which goal is twofold: to represent the current state of the simulation by keeping track the specification instances that currently exist and to allow the interaction between the specification instances. In particular, *CurrentState* is represented by a list of instances (instance values, see listing 3.3).

- In addition, as mentioned in section 2.2.1, there are two type of transitions: initial transitions which signal the creation of a specification instance and does not have an origin state, and intermediate transitions which trigger a transition to an already existing specification instance. For that reason, we implemented the *Event* trait which

is extended by the case classes *CreateInstance* and *TriggerTransition*. Each type of event consists of: the values of the identity of the instance, the module name of the specification that this event refer to, the name of the transition to be taken, and finally the arguments of the transition.

- An event is matched against the two aforementioned case classes. In the case of triggering a transition to an already existing instance, we first search in our data structure called *CurrentState*, which keeps track of the instances created in the simulation, and return, if found, the instance that the transition refers to (line 23). Subsequently, we check whether the instance has already reached its final state which means that no transition is permitted (line 27), if the instance has not reached its final state we proceed with the population of the transition's environment. On the contrary, in the case of creating an non existing instance we do not look for the instance in the current state of the simulation but we initialize the instance to be created with *null* values (not listed here).

- The return value of the function is an updated list of the instances that currently exist in the simulation (*CurrentState*).

```
1  def simulate(program: XlinqProgram, eventsList: List[Event], currentState:
       CurrentState): CurrentState = eventsList match {
2      case List() => currentState
3      case event :: restEvents =>
4        event match {
5          case CreateInstance(instanceIdentityValues, moduleName,
               transitionName, transitionArguments) =>
6            val spec = program.getMaverickSpec(moduleName) match {
7              case Some(s) => s
8              case None => throw new TransitionException("Specification not
                   found")
9            }
10           /* ... initialization with null values of the instance to be created
                 not listed ...*/
11           /* ... population of the transition's environment not listed ...*/
12           .
13           .
14           handleTransition(program, transition, transitionEnvironment,
                 instanceToBeCreated, currentState, destination) match {
15             case Right(ncs) => simulate(program, restEvents, ncs)
16             case Left(message) => throw new TransitionException(message)
17           }
18         case TriggerTransition(instanceIdentityValues, moduleName,
               transitionName, transitionArguments) =>
19           val spec = program.getMaverickSpec(moduleName) match {
20             case Some(s) => s
21             case None => throw new TransitionException("Specification not
                   found")
```

```
22          }
23          val instance = findFSMinstance(currentState.listOfInstances,
                formIdentity(spec.identity, instanceIdentityValues), moduleName)
                match {// search and return, if found, the instance for which
                the transition refers to
24            case Some(inst) => inst
25            case None => throw new UndefinedException("The FSM instance does
                  not exist.")
26          }
27          if (hasReachedFinalState(spec, instance.state)) //if the instance
                has reached its final state no transition can be taken
28            throw new TransitionException("Event could not get processed
                  because the FSM has reached the final state")
29          /* ... population of the transition's environment not listed ...*/
30          .
31          .
32          handleTransition(program, transition, transitionEnvironment,
                instance, currentState, destination) match {
33            case Right(ncs) => simulate(program, restEvents, ncs)
34            case Left(message) => throw new TransitionException(message)
35          }
36      }
37  }
```

Listing 3.1: Starting point of our implementation: simulate function

Consequently, the handling of an event/transition, in both type of events, is delegated to the function *handleTransition*(line 14 and 32 in listing 3.1). The function is depicted in listing 3.2 and takes as arguments the given set of specifications (*program*), the transition to be handled (*transition*), a map of variable bindings that are under scope (*environment*), the instance that this transition refers to (*fsmInstance*), the current state of the simulation (*currentState*) and the destination state of the specification instance (*destination*). Supplementing the listing 3.2 should be noted the following:

- Each transition is handled under a given environment. The environment is a map of variable bindings that are under scope and can also have a parent environment. This represents the two following scopes: the local scope of a transition which is extended with the transition arguments, and its parent scope which is the scope of the instance that this transition refers to and is extended with the definition of "this" as a sort of hidden variable (line 2).
- We see in lines 3,5,8,9 and 14 the invocation of the actual interpreter through the usage of auxiliary functions. First, the interpreter is invoked to evaluate the constant expressions of a transition which are related with the default parameters of a transition (line 3), after their evaluation the local environment of the transition is extended with their values (line 4). In the remaining lines, we first perform a short-circuit evaluation to the pre-conditions of a transition (line 5) and we proceed with the execution of the transition's actions that are not external invocations (line 8). Then, we evaluate the

external invocations that a transition might entail. Following the description given in section 2.2.1, if a secondary transition fails then the primary transition fails too (line 11). Finally, we perform a short-circuit evaluation to the post-conditions of a transition (line 13). If they also hold then a transition can be considered successful and we proceed with updating the state of the simulation (line 14).

- It is worth to mention that in case that all the external invocations succeed then they also return an updated state of the simulation (line 8). We will see how this is done in the remaining of this section.

```
1  def handleTransition(program: XlinqProgram, transition: Transition,
       environment: Environment, fsmInstance: InstanceV, currentState:
       CurrentState, destination: String): Either[String, CurrentState] = {
2      environment.parent.updateEnvironment("this", fsmInstance)
3      val constantBinds = evalConstantExpressions(transition.constants,
           program, environment, currentState)
4      constantBinds.foreach(constant =>
           environment.updateEnvironment(constant.name, constant.value))
5      if (evalConditions(transition.preconditions, program, environment,
           currentState)) {
6        val (externalInvocations, otherActions) = transition.actions.span(a =>
             isExternalInvocation(a))
7        evalExpressions(otherActions, program, environment, currentState)
8        val newCurrentState = evalExternalInvocations(program,
             externalInvocations.toList, environment, currentState) match {
9          case Right(ncs) => ncs
10         case Left(message) => throw new ExternalInvocationException(message)
11       }
12       val postconditions = filterPostConditions(transition.postconditions)
             //filter external invocations(they have already succeed or failed
             during execution of actions)
13       if (evalConditions(postconditions, program, environment,
             newCurrentState))
14         updateCurrentState(newCurrentState, fsmInstance, environment,
               destination)
15       else
16         Left("Post-conditions do not hold.")
17     }
18     else
19       Left("Pre-conditions do not hold.")
20 }
```

Listing 3.2: The function responsible for handling a transition

As we described previously, the function *handleTransition* invokes the interpreter, through the usage of auxiliary methods, for evaluating the Maverick expressions. Before, we can start to look the evaluations performed by the definitional interpreter, we need a way to represent the results of such evaluations. Therefore, we defined the trait *Value* as an algebraic

data type with seventeen possible cases, one for each value type. This definition is shown in listing 3.3.

```
1  sealed trait Value
2  final case class StringV(value: String) extends Value
3  final case class BooleanV(value: Boolean) extends Value
4  final case class IntegerV(value: Int) extends Value
5  final case class RealV(value: Double) extends Value
6  final case class CurrencyV(value: String) extends Value
7  final case class PercentageV(value: Double) extends Value
8  final case class MoneyV(currency: CurrencyV, amount: MonetaryAmountV) extends
       Value
9  final case class MonetaryAmountV(amount: BigDecimal) extends Value
10 final case class NowV() extends Value
11 final case class IbanV(value: String) extends Value
12 final case class DateV(value: LocalDate) extends Value
13 final case class TimeV(value: LocalTime) extends Value
14 final case class DateTimeV(value: DateTime) extends Value
15 final case class PairV(key: Value, value: Value) extends Value
16 final case class SetV(values: Set[Value]) extends Value
17 final case class MapV(pairs: Seq[PairV]) extends Value
18 final case class InstanceV(identity: immutable.Map[String, Value],
       moduleName: String, state: String, fields: mutable.Map[String, Value])
       extends Value
```

Listing 3.3: Definition of trait Value

While string, integer, boolean and other values are trivial, instance values are somewhat more interesting. A specification instance is characterized by its identity, the specification that is an instance of, its current state and field values, therefore is implemented as the combination of the aforementioned four components: an identity which is an immutable map of values, a module name indicating the maverick specification, the state that is currently the instance and a mutable map of field values.

The actual interpreter that implements the semantics of a Maverick expression is defined in the object *Interpreter*. More specifically, it exposes three methods which exploit Scala's method overloading - using the same name for different methods that take different types -, which are outlined in listing 3.4. They take a Maverick expression, a set of Maverick specifications, an evaluation context and the current state of the simulation and return a value (as defined in 3.3) or, in case of evaluating an external invocation, return an updated state of the simulation(line 2 in listing 3.4).

```
1 def interp(exp: Expression, program: XlinqProgram, env: Environment,
      currentState: CurrentState): Value
2 def interp(externalInvocation: ExternalInvocation, program: XlinqProgram,
      environment: Environment, currentState: CurrentState): Either[String,
      CurrentState]
3 def interp(lit: Literal, program: XlinqProgram, env: Environment,
```

```
currentState: CurrentState): Value
```

Listing 3.4: Interpreter's overloaded methods

These evaluation functions are a large match expression with cases for each node in the intermediate AST representation. Each one is responsible for evaluating a different type of expression. In particular, line 2 is responsible for evaluating external invocations, line 3 evaluates literal expressions and line 1 evaluates the rest of the expressions that do not fall under one of the above categories (all lines refer to listing 3.3).

The leaves of an expression tree are literals and their evaluation is straightforward. Therefore, we proceed with the listings in 3.5 and 3.6 where we present in more detail the interpretation of the most important cases:

- Both of the functions in listings 3.5 and 3.6 are structured as recursive traversals. Most of the expressions are recursively evaluated as they might contain more expressions in their body.

- Variable references are looked up in the variable bindings that are contained in the environment (line 2 in listing 3.5). If there is no variable defined in the environment with the requested name, then an error is thrown.

- Arithmetic expressions are evaluated by delegating to the helper function *evaluateArithmeticExpression* (line 19 in listing 3.5). In this helper function, which is not listed here, besides arithmetic operations also subtraction of dates is defined which results to an integer value. Finally, when arithmetic operations occur on operands of different numeric types, the less inclusive type (eg. Int) is converted to the more inclusive type (eg. Double).

- In a similar way, the evaluation of comparison expressions is delegated to the helper function *evaluateComparisonExpression* (line 18 in listing 3.5).

- An assignment expression occurs during the execution of the actions of a transition (line 6 in listing 3.5). The interesting part here is that it is possible to have a post condition expression that refers twice to the same named variable but with the difference that one reference is related to the variable's value before the execution of the transition's actions and the other one to the variable's value after. Therefore, after we check if the variable already exist in the environment, a post transition value of a variable is stored in the environment, with the variable's name concatenated with the prefix "new". So in that way we can retrieve, if it is needed, both pre and post transition values of a variable.

- The "PostTransitionMember" expression is used to retrieve the post transition value of a variable (line 32 in listing 3.5). According to the previous definition, a post transition variable is evaluated by looking up the variable's name in the environment after we concatenate the variable's name with the prefix "new". If the variable is not found then we look up once again in the environment as a transition might contain

no assignments related with the variable and therefore the variable might still exist in the environment without the prefix. If still the variable is not found then an error is thrown.

- The expression "LocalReference" (line 20 in listing 3.5) basically refers to the keyword "this" and therefore its value is equal with searching for the hidden variable with name "this" in the given environment.
- An "InState" expression (line 28 in listing 3.5) is checking if a variable, which should evaluate to an instance value, is in the given state and is evaluated to a boolean value accordingly.
- The expression "Initialized (line 24 in listing 3.5) is evaluated to a boolean value according to whether the interpretation of a variable would result in an instance value which is not in the origin state (which means it has not been instantiated yet as described in section 2.2.1).
- The evaluation of an external invocation is given in listing 3.6. We can have two type of external invocations: either via a reference field (line 2) or via an identity of a specification instance (line 20). The difference between the two different ways to invoke an external invocation is the following: in the invocation via field we first evaluate the field that refers to another specification (line 3) whereas in the invocation via identity we call the interpreter recursively to evaluate the identity given (line 21).
- The latter evaluation is done by first evaluating the parameters that compose its identity value and followingly we look up for an instance that has this identity value in the current state of the simulation (see line 44 in listing 3.5). While the former evaluation should evaluate the given field (which ultimately is a variable) to a specification instance already instantiated in the given environment. Subsequently, in both cases we check whether the instance has reached already its final state(line 11 and 30). In the case that the instance is in its final state an error is thrown as no transition can take place. Otherwise, we proceed with the evaluation of the transition's arguments (line 13 and 31) and we prepare accordingly the environment for the transition to be handled (lines 16, 17, 18 and 34, 35, 36). Lastly, we recursively invoke the function handleTransition that we presented earlier.

```
1  def interp(exp: Expression, program: XlinqProgram, env: Environment,
       currentState: CurrentState): Value = exp match {
2    case Variable(name) => env.lookupVariable(name) match {
3      case Some(v) => v
4      case None => throw new UndefinedException("Variable not found.")
5    }
6    case Assignment(lhs, rhs) => lhs match {
7      case Variable(name) =>
8        env.lookupVariable(name) match {
9          case Some(_) =>
```

```
10            val newValue = interp(rhs, program, env, currentState)
11            val newName = "new" + name
12            env.update(newName, newValue)
13            newValue
14          case None => throw new UndefinedException("Variable not found.")
15        }
16      case _ => throw new IllegalExpressionException("Left side of the
            assignment expression should be a variable.")
17    }
18    case ce: ComparisonExpression => evaluateComparisonExpression(ce,
          program, env, currentState)
19    case nbe: NumericBinaryExpression => evaluateArithmeticExpression(nbe,
          program, env, currentState)
20    case LocalReference() => env.lookupVariable("this") match {
21      case Some(instance) => instance
22      case None => throw new UndefinedException("Local reference not found.")
23    }
24    case Initialized(_, variable) => interp(variable, program, env,
          currentState) match {
25      case InstanceV(_,_,state,_) => if(state != "origin") BooleanV(true)
            else BooleanV(false)
26      case _ => throw new UndefinedException("Specification instance not
            found.")
27    }
28    case InState(_, variable, state) => interp(variable, program, env,
          currentState) match {
29      case InstanceV(_, _, instanceState, _) => BooleanV(instanceState ==
            state)
30      case _ => throw new UndefinedException("Specification instance not
            found.")
31    }
32    case PostTransitionMember(expression) => expression match {
33      case Variable(name) =>
34        val postTransitionVariable = "new" + name
35        env.lookupVariable(postTransitionVariable) match {
36          case Some(value) => value
37          case None => env.lookupVariable(name) match { //look again for the
                variable in the environment without the prefix "new" because
                sometimes there are no actions so no assignments at all or no
                assignments related with a variable
38            case Some(value) => value
39            case None => throw new UndefinedException("Variable not found.")
40          }
41        }
42      case _ => throw new IllegalExpressionException("Illegal expression for
            a PostTransitionMember. It can only be a variable.")
43    }
44    case IdentityOf(reference, arguments) =>
          program.getMaverickSpec(reference.fqn) match {
```

```
45        case Some(s) =>
46          val instanceIdentityValues = arguments.map(p => interp(p, program,
                env, currentState))
47          //look up for an instance with the given identity value in the current
                state of the simulation
48          findFSMinstance(currentState.listOfInstances, formIdentity(s.identity,
                instanceIdentityValues), reference.fqn) match {
49            case Some(inst) => inst
50            case None => throw new UndefinedException("FSM instance not
                  defined.")
51          }
52        case None => throw new UndefinedException("Specification not found")
53      }
54      /* ...some more cases... */
```

Listing 3.5: Evaluation of other Maverick expressions

To demonstrate how our interpreter implements the same semantics that we described in section 2.2.1, suppose that we have the same example of sequence of events as the one that we give in the preceding section (see example 2.2.1). This sequence of events will be handled as following:

- The first three events are related with the creation of a new specification instance, hence they will be matched against the case in line 5 of listing 3.1. After we find the specification and transition that the corresponding event refers to, we delegate the handling of the transition to the method *handleTransition*(line 14 in listing 3.1). In that method, the handling of a transition is performed by following exactly the same steps described in section 2.2.1. In particular, first the pre-conditions are checked (line 6 in listing 3.2), then the actions that the transition entails are executed (line 7, 8 in listing 3.2) and finally we check whether the post conditions hold.

- The actual interpreter is invoked for the evaluation of the expression trees that compose the aforementioned conditions and actions (listing 3.4). For instance, the evaluation of the expression trees that compose the post-conditions will first match against the case in line 18 of listing 3.5. This helper method will evaluate the expressions that compose a comparison expression recursively. To give you an idea, at some point a post-transition value expression will be necessary to be evaluated in the post conditions (as explained in our example in section 2.2.1), this means that the method in 3.5 will be called recursively with argument the *PostTransitionMember* expression which then will be matched against the case in line 32 of the same listing and the evaluation will continue accordingly.

- After the interpreter has recursively evaluated the conditions and actions of a transition, an updated current state of the simulation will be returned which in this particular sequence of events will include three Maverick specification instances, namely the account A1 in state *opened* and balance 60 EUR, the account A2 in state *opened*

and balance 100 EUR and a transaction instance in state *validated* with amount 20 EUR from account A1 to account A2.

- Finally, the last event triggers a transition to an already existing instance, namely the transaction instance. Thus, the event is matched against the case in line 18 of listing 3.1. Before, we delegate the handling of the transition to the method *handleTransition* we first look up for the instance in the current state of the simulation (line 23 in listing 3.1) and perform the necessary checks (eg. instance reached final state).

- The transition follows the same procedure as explained before but what should be noted here is that the *book* transition signals the external invocation of transitions in the account A1 and account A2. This interaction is handled in line 8 of listing 3.2. In more detail, a list of external invocations (withdraw 20 EUR from A1, deposit 20 EUR to A2) is passed to the interpreter (listing 3.6) for evaluation. First the *withdraw* transition will be evaluated by recursively invoking the *handleTransition* (line 19 in listing 3.6). If the *withdraw* transition fails then a error will be thrown causing also the primary transition to fail. In case that it succeeds, then the *deposit* transition will be evaluated. In that way we implement the semantics of the external invocations described in section 2.2.1 (i.e. in order for the primary transition to be valid the secondary transitions must first succeed).

- After the successful completion of the external invocations of *withdraw* and *deposit*, the *book* transition will succeed too. As a result the current state of simulation includes: a transaction instance in the final state *booked*, account A1 in state *opened* and balance 40 EUR, account A2 in state *opened* and balance 120 EUR.

```
1  def interp(externalInvocation: ExternalInvocation, program: XlinqProgram,
       environment: Environment, currentState: CurrentState): Either[String,
       CurrentState] = externalInvocation match {
2      case ExternalInvocationViaField(specification, field, transitionName,
           arguments) =>
3        val instance = interp(field, program, environment, currentState) match {
4          case i: InstanceV => i
5          case _ => throw new ExternalInvocationException("Field doesn't refer
               to a specification instance, hence external invocation is not
               possible.")
6        }
7        val spec = program.getMaverickSpec(specification.fqn) match {
8          case Some(s) => s
9          case None => throw new ExternalInvocationException("Specification not
               found during external invocation.")
10       }
11       if (hasReachedFinalState(spec, instance.state))
12         throw new ExternalInvocationException("Transition is not possible
               because the FSM has reached the final state")
13       val transitionArgumentsValues = arguments.map(arg => interp(arg,
             program, environment, currentState))
```

```
14        val (transition, destination) = getTransitionAndDestination(spec,
              transitionName, instance.state)
15        val specificationEnvironment = new Environment(null)
16        instance.fields.foreach(field =>
              specificationEnvironment.updateEnvironment(field._1, field._2))
17        val transitionEnvironment = new Environment(specificationEnvironment)
18        transitionEnvironment.populateEnv(transition.parameters.toList,
              transitionArgumentsValues)
19        handleTransition(program, transition, transitionEnvironment, instance,
              currentState, destination)
20      case ExternalInvocationViaIdentity(identity, transitionName,
            injectedParams, ownParams) =>
21        val instance = interp(IdentityOf(identity.reference, injectedParams ++
              identity.parameters), program, environment, currentState) match {
22          case i: InstanceV => i
23          case _ => throw new ExternalInvocationException("Specification
                instance not found during external invocation via identity.")
24        }
25        val spec = program.getMaverickSpec(instance.moduleName) match {
26          case Some(s) => s
27          case None => throw new ExternalInvocationException("Specification not
                found during external invocation.")
28        }
29        if (hasReachedFinalState(spec, instance.state))
30          throw new ExternalInvocationException("Transition is not possible
                because the FSM has reached the final state")
31        val transitionArgumentsValues = ownParams.map(arg => interp(arg,
              program, environment, currentState))
32        val (transition, destination) = getTransitionAndDestination(spec,
              transitionName, instance.state)
33        val specificationEnvironment = new Environment(null)
34        instance.fields.foreach(field =>
              specificationEnvironment.updateEnvironment(field._1, field._2))
35        val transitionEnvironment = new Environment(specificationEnvironment)
36        transitionEnvironment.populateEnv(transition.parameters.toList,
              transitionArgumentsValues)
37        handleTransition(program, transition, transitionEnvironment, instance,
              currentState, destination)
38    }
```

Listing 3.6: Evaluation of an external invocation

As can be seen from the above example the interpreter defines the meaning of the various components of the Maverick language and give to them a precise meaning matching the description given in the previous chapter. Furthermore, the style of the code of our interpreter reveals its primary goal, that is to define Maverick's language constructs rather than describe an efficient implementation of the Maverick language. We strive to emphasize clarity and readability in our code by using compositional, recursively-defined evaluation functions over the Maverick language constructs and maintaining a pure functional style.

Finally, we never try to emphasize in efficiency, of course this might result in a slow implementation or other performance issues but the purpose of a definitional interpreter is to produce a specification rather than usability.

Chapter 4

Test Case Generation from State Machines

In this chapter, we present our approach regarding the generation of input stimuli for both the definitional interpreter and generated code. As the Maverick DSL is based on state machines, the problem lies on generating test sequences for a state machine. To test the code generator in different circumstances and subsequently increase our confidence about whether the code generator is bug-free or not, it is necessary the generated input stimuli to be able to explore the different paths of a state machine. This chapter begins with introducing the concept of generating test cases for behavioral models along with existing approaches that try to tackle this problem. Then, we continue with our approach on generating test cases capable of traversing different paths of a Maverick specification.

4.1 Background

Although formal approaches have been developed to verify software artifacts, software testing remains one of the most widely used verification and validation technique. Software testing entails the execution of the system under test (SUT) and the comparison of the obtained behavior with an expected one. A significant factor in software testing, as testing can only provide incomplete answers about the correctness of a software system, is the specified test cases for which the behavior of the SUT will be tested. Therefore, specifying test cases that are able to rigorously test a SUT is fundamental towards the development of trustable and robust software systems.

An approach that has drawn a lot of interest in both industry and academia is model based testing (MBT) in which models are used to steer the testing process. MBT relates to a process in which tests are derived from a model of a SUT by employing a number of sophisticated methods [11]. The main idea of model based testing is that instead of creating manually test cases (eg. use cases) an algorithm derives them automatically from a model. According to Utting et al. [50] MBT usually constitutes a type of functional testing (black box) where no implementation details of the SUT are taken into account in the construction of the model. The model of a SUT can either be derived for testing purposes or, in case of

model driven development methods, the already available model can be leveraged also for the generation of test cases. Some examples of software models that have been used for testing are Finite State Machines (FSMs) [27], Markov chains [56] and Statecharts [14].

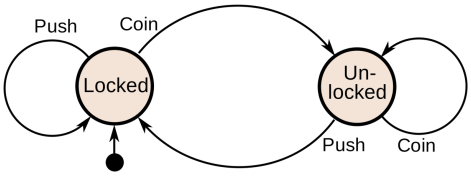
In this section we focus on the methodology that use state machines (or other variations of state machines) as a basis for generation of tests.

4.1.1 State machines

Transition systems, also known as state machines, are a fundamental concept for computer science and as such many variants exist. State machines and its variations have been widely used to model systems in many different areas [12, 16] and therefore automated test case generation from transition systems has long been studied.

A finite state machine is a model of computation based on an abstract machine which is made of one or more states (finite number of states). Only one state can be active at any given time, hence the machine must transition from one state to another state in order to carry out different actions.

(a) A turnstile



(b) State-transition diagram for a turnstile

Figure 4.1: A turnstile represented as a state machine

To better grasp the concept of a state machine we can think one of the many examples in which devices that are used in daily life exhibit the behavior of state machines. A good example to illustrate that is to consider the case of modelling as a state machine a coin operated turnstile machine which is used to control access in different type of locations (eg. subways, amusement parks) [24]. A turnstile, which consists of three rotating arms at waist level (figure 4.1a), is initially locked and the arms cannot be moved, blocking the entry in that way. Inserting a coin or a token unlocks the turnstile which enables the arms to rotate,

allowing a single customer/commuter to pass through. Once the customer passed, the arms are locked again until the insertion of the next coin.

A FSM can be represented by a state-transition diagram which can also be seen as a directed graph whose vertices correspond to the states of the machine and its edges to the transitions that connect the different states of the machine. In addition, each edge is labeled with information associated with the transition, for example, information regarding the conditions that should hold (or the input that should be given) in order for a transition to take place. In the example of a turnstile that we gave before, the state-transition diagram is given by two vertices representing the states of a turnstile (Locked, Unlocked) and edges that connect the vertices, indicating the transitions that can be taken along with the input that must be given in order for a transition to take place (4.1b).

4.1.2 Automatic test case generation from State Machines

First of all, it should be clear that with the term *test case* we refer to the description of a single test and with the term *test suite* we refer to a set of test cases. In addition, based on the characteristics of the SUT, test cases can have different forms [58]. For instance, if the characteristic of a SUT that is under consideration is functional, then a test case can have the form of pairs of input and output values [58]. On the contrary, if the SUT is a reactive system, then a test case would take the form of a sequence of events.

However, generating test cases for all possible behaviors of a SUT is not feasible. Thus, it is necessary to select an adequate subset of test cases for driving the testing process. Usually, this is achieved through the usage of *coverage criteria* which typically control the generation of test cases or measure the quality of an already existing test suite [50]. The coverage of a test suite can be defined using different criteria according to the abstraction level of the SUT such as requirements, code or model coverage.

In particular, model coverage criteria help us to identify to which extent our generated test cases resemble the modeled requirements and as a result enables us to test the SUT in different scenarios. A model coverage criterion is agnostic of the specific details of a model and can be applied to any element of a model (eg. state, event, transition) or to a combination of them. More specifically, some of the most common coverage criteria for state machines are the following[50]:

- All-states: This coverage criterion is satisfied if each state of a state machine is contained to at least one test case.
- All-transitions: Satisfying this criterion requires for each transition of a state machine to have at least one test case that contains it.
- All-n-transitions: Similar as before but with the difference that requires to traverse all transition sequences up to length n.
- All-paths: A test suite satisfies this criterion if all possible paths of a state machine are traversed. However, this coverage criterion is considered impossible to satisfy.

Other type of coverage criteria used in state machines are related with transition conditions and are known as control-flow based coverage criteria (eg. decision coverage, condition coverage).

Typically, the first step in generating test cases for state machines is to create paths on the model level based on some coverage criteria, known as *abstract test case generation*. As explained in section 4.1.1 state machines can also be seen as directed graphs. Therefore, many approaches exist that use graph traversal algorithms to find paths in state machines ([7, 33, 39, 54, 53]). Traversing a graph entails to start at a node in a graph and continue traversing edges of the graph until a stopping condition is met (eg. all states have been visited, all edges have been traversed etc). This can be done using several graph traversal algorithms such as breadth first search, depth first search or Dijkstra's shortest path algorithm.

However, abstract test cases only contain information about the sequence of the transitions that are needed to traverse a certain path, but missing information regarding the concrete parameter values that are necessary to make feasible the traversal of a path. Several approaches have been developed in combination with graph traversal techniques to produce concrete test cases. To name a few, such methods are partition testing which defines input value partitions and selects representative values from them [55, 54], boundary value analysis in which the idea is to select values at the boundaries of an input domain (eg. guard condition) [25] and using a constraint solver which basically tries to produce values that satisfy a set of constraints [13].

Finally, another approach in test case generation is *random testing* [34]. The power of random testing lies in the fact that without spending much effort someone can produce quickly a large number of test cases, and in that way, reveal flaws of the SUT early in the development phase [58]. This approach can also be combined with the graph traversal technique by selecting, for instance, randomly the next state to be visited or transition to be traversed.

4.2 Implementation of the Test Case Generation for the Maverick DSL

This study does not focus on the design of a novel test case generation for state machines. Hence, the approached implemented and described in the current section follows the general methodology described in section 4.1.2, namely first the generation of abstract test cases and subsequently their concretization with actual values. Its purpose is twofold: first to increase our confidence in code generator's reliability and correctness, and second to evaluate the proposed testing approach for code generators.

As we described in section 2.2.1, Maverick is an actor based language meaning that any transition is initiated by an actor. Therefore, we are interested in generating test cases that would have the form of a sequence of events and which will enable us to observe the behavior of the Maverick specifications and generated code.

As Maverick is based on state machines, the first step of our approach is to convert a Maverick specification into a representation of a directed graph with labeled edges, the pos-

sible states of a Maverick specification as vertices and its transitions along with the origin and destination states of a transition as labeled edges. The traits in listing 4.1 serve exactly that purpose, as can be seen in that figure the directed graph (line 1) is represented with an adjacency map that has as key a vertex in the graph and as value a set of vertices that are adjacent to that vertex. The labels of the edges are captured in a map which has as a key a tuple consisting of the origin and destination state of a transition and as value the transition itself (line 7).

```
1 trait DirectedGraph[A] extends Iterable[A] {
2   protected val adjacency: mutable.Map[A, Set[A]] = mutable.Map()
3   ...
4 }
5 trait LabeledEdges[A, T] {
6   this: DirectedGraph[A] =>
7   protected val labels: mutable.Map[(A, A), Set[T]] = mutable.Map()
8   ...
9 }
```

Listing 4.1: The traits used to represent a Maverick specification as a directed graph

Subsequently, we follow the methodology that is described in section 4.1.2 for test case generation from state machines. In particular, we first create abstract test cases using a modified version of the breadth-first search algorithm illustrated in listing 4.2. The aim of this algorithm is the following: given a directed graph of a Maverick specification to find all the paths between an origin and a destination state. However, finding all the paths in a directed graph that might contains cycles, which is the case for the Maverick specifications (see figure 2.4), is infeasible. Therefore, our algorithm finds all the paths between two states that are under a given length (line 14). At the end the algorithm returns a list of the paths found represented by the consecutive vertices/states that were visited during the graph traversal from the starting to the target node.

```
1 def bfs_findPaths(graph: DirectedGraph[String] with
      LabeledEdges[String,Transition], start: String, end: String, length:
      Int): List[List[String]] = {
2     var pathQueue = List[List[String]]()
3     var tmp_path = List[String](start)
4     var paths = List[List[String]]()
5     pathQueue = pathQueue :+ tmp_path
6
7     while (pathQueue.nonEmpty) {
8       tmp_path = pathQueue.head
9       pathQueue = pathQueue.filterNot( p => p == tmp_path)
10      val last_node = tmp_path.last
11      if(last_node == end)
12        paths = paths :+ tmp_path
13      for(link_node <- graph.directSuccessors(last_node)) {
```

```
14        if(tmp_path.size < length) {
15           val new_path = tmp_path :+ link_node
16           pathQueue = pathQueue :+ new_path
17        }
18       }
19     }
20     paths
21   }
```

Listing 4.2: Modified version of breadth-first algorithm for finding all the paths under a certain length between an origin and destination state

Following, using an auxiliary method we convert the output of the algorithm described before to the classes illustrated in listing 4.3, in which a path (line 2) resembles a list of transitions. This helper method connects the consecutive states that were produced before with possible transitions. In case that there is more than one possible transition between a start and end state, the helper function determines which transition will connect the two nodes by first looking whether there is a transition (from the set of possible transitions between two nodes) which has not been used yet. If all of them have been used then a transition from all possible transitions will be selected randomly.

The next step is to generate concrete parameters for the abstract test cases generated before. The aim is to generate parameters for the transitions that will actually make possible their execution, or in other words, parameters that satisfy the conditions required for a transition to be executed. It should be noted that this step is tailored to the experimental machines presented in chapter 2. For this step we generate pseudo-randomly values for the transitions' parameters of the experimental Maverick specifications.

```
1 case class FsmTransition(from: String, to: String, transition: Transition)
2 case class FsmPath(transitions: List[FsmTransition])
```

Listing 4.3: Definition of the path and transition classes

The method responsible for making concrete the abstract test cases is defined in the object *InputGenerator* and is presented in the listing 4.4 where we depict the overall structure of the method. Some clarifications supplementing listing 4.4 are listed below:

- The method takes us an input: a list of the generated abstract paths, the Maverick specification for which we will generate concrete test cases, a set of Maverick specifications (program), a mutable map which purpose is to store the identity values that will be generated for each test case and finally a boolean parameter that indicates whether the test cases that will be created will be valid test cases or completely random, meaning that invalid cases might be created.

- The mutable map *identitiesCreated* stores the identities created during the concretization of a test case. In that way each event contained in a test case can be populated in such way to refer to the same specification instance.

- The method is structured as an iteration over each path that was generated in the previous step. For each transition included in the corresponding path we create a concrete event. To do so, we first determine if a transition refers to an initial transition (line 8). If that is the case, we generate events for the referenced fields of the specification (if there are any, line 10) and we continue by creating values for the identity and transition arguments using the auxiliary methods in lines 11, 15, 17. In case that the transition that we want to create a concrete event is not an initial transition, we follow the same procedure but with the difference that the identity values needed for the transition will be retrieved from already existing values (line 19).

- As we mentioned earlier, the concretization step described here is specifically tailored to the needs of the experimental Maverick specifications that are illustrated in section 2.2.1 (i.e. only when the *createValid* parameter is set to *true*). To be more precise, consider the example of having a Maverick specification of a simple account (listing 2.1) and receiving an event that triggers the transition *close* (line 92 in listing 2.1). As it can be seen in the aforementioned listing, the transition *close* requires the balance of the account to be equal with zero euro (line 94 in listing 2.1) in order to be executed. To create a valid transition we keep track of the balance of the account (line 5 in listing 4.4) during the concrete creation of the events. Subsequently, we create an extra *withdraw* transition that will withdraw the remaining amount of money from the corresponding account(line 28 in listing 4.4). In that way, we assure we will have a valid *close* transition.

- Similarly, the auxiliary method in lines 13, 22 in listing 4.4) creates an amount that will make the transition possible. This is accomplished by taking into account the name of the transition that is about to get concretized. For example, in case of a withdraw transition, the method will create a pseudo-random amount which will be below the current balance (i.e. current balance is given as an argument). Finally, it will return the amount that was generated and an updated balance. In case of the withdraw transition this means that the updated balance will be equal to: *balance - generatedAmount*.

- Finally, the auxiliary methods in lines 11, 17, 24 generate values based on the type of the transition's argument and by using a primitive value randomizer which is listed in 4.5). Thus, for each type like Integer, String etc., a random generator has been implemented inside the object PrimitiveValueRandomizer. However, the *IBAN* type is more complex than the other types as its value should be compliant to the IBAN standards. This is required in the generated system which is a banking system and therefore must be compliant to the IBAN standards. For that reason the library *iban4j*[1] has been used to generate randomly valid *IBAN* values (line 5 in listing 4.5). Finally, the primitive value randomizer is instantiated by using as seed the current date time expressed in milliseconds (line 2 in listing 4.5).

[1] https://github.com/arturmkrtchyan/iban4j

```
1  def createConcreteEventSequences(paths: List[FsmPath], spec: Specification,
       program: XlinqProgram, identitiesCreated: mutable.Map[String,
       Set[List[Value]]], createValid: Boolean): TestSuite = {
2    var eventsForExternalFields: List[Event] = List()
3    var balance: BigDecimal = 0.0
4    val testCases: List[TestCase] = for(path <- paths) yield { //iterate over
         all the abstract paths
5     balance = 0.0 //keeping track of the balance of an account based on the
          events generated
6     /*...*/
7     var concreteEventSequences: List[Event] = transitions.map(tr =>
8      if(tr.from == "origin") {
9        //create events for the reference fields of a specification (if any)
10       eventsForExternalFields =
             initializeExternalFields(externalReferencedFields, program,
             identitiesCreated)
11       val identityValues = createIdentityValues(spec.identity)
12       //update the map which stores the identities that have been created
13       identitiesCreated.update(spec.module.fqn,identitiesCreated.getOrElse(spec.module.fqn,
             Set.empty) + identityValues)
14       //creates a valid amount (or a completely random amount) and updates
             accordingly the balance)
15       val (amount, updatedBal) = createValidAmount(tr.transition.name,
             balance, createValid)
16       balance = updatedBal
17       val argumentsValues =
             createTransitionArgumentValues(tr.transition.parameters.toList,amount,
             identitiesCreated)
18       CreateInstance(identityValues, spec.module.fqn, tr.transition.name,
             argumentsValues)
19     }else{
20       val identityValues = findIdentiyValues(spec.module.fqn,
             identitiesCreated)
21       //creates a valid amount (or a completely random amount) and updates
             accordingly the balance)
22       val (amount, updatedBal) = createValidAmount(tr.transition.name,
             balance, createValid)
23       balance = updatedBal
24       val argumentsValues =
             createTransitionArgumentValues(tr.transition.parameters.toList,
             amount, identitiesCreated)
25       TriggerTransition(identityValues, spec.module.fqn,
             tr.transition.name,argumentsValues)
26     }
27    )
28     if(createValid && spec.name == "Account") { // to create a valid
           "close" transition
29     /*...*/
30      val argumentsValues =
```

```
            createTransitionArgumentValues(transWithdraw.transition.parameters.toList,
            balance, identitiesCreated)
31        concreteEventSequences = concreteEventSequences :+
            TriggerTransition(identityValues, spec.module.fqn, "withdraw",
            argumentsValues)
32        concreteEventSequences = concreteEventSequences :+
            TriggerTransition(identityValues, spec.module.fqn, "close", List())
33      }
34      identitiesCreated.update(spec.module.fqn, Set.empty) //initialize back
            to empty the map for the next test case
35      TestCase(eventsForExternalFields ++ concreteEventSequences)
36    }
37    TestSuite(testCases)
38  }
```

Listing 4.4: Method responsible for the concretization of the abstract test cases

```
1  object PrimitiveValueRandomizer {
2    def apply(seed : Long = DateTime.now(UTC).getMillis):
         PrimitiveValueRandomizer = new PrimitiveValueRandomizer(seed)
3
4    /*...*/
5    //random generation of IBAN using iban4j
6    def randomIBAN(country : CountryCode = DefaultCountryCode,
7                   bankCode : String = DefaultBankCode,
8                   branchCode : String = DefaultBranchCode) : String = {
9      new Iban.Builder().
10         countryCode(country).
11         bankCode(bankCode).
12         branchCode(branchCode).
13         buildRandom.
14         toString
15   }
16
17    /*...*/
18 }
19 //instantiated as following in the InputGenerator object
20 implicit val rand: PrimitiveValueRandomizer = PrimitiveValueRandomizer()
```

Listing 4.5: Primitive value randomizer

The method described above yields a test suite related with the given Maverick specification and represented by a list of test cases (resembled in the classes displayed in listing 4.6). It should be mentioned that the concrete test cases generated after this step are in the form of the events that the method *simulate* described in section 3.2 take us an input.

```
1  case class TestSuite(testCases: List[TestCase])
```

```
2 case class TestCase(eventSequence: List[Event])
```

Listing 4.6: Definition of the test case and test suite classes

Chapter 5

Evaluation

In section 1 we introduced the following research question: *how could Code Generators be tested against a definitional interpreter?*. In the previous chapters, we presented and described our approach on testing ING's code generator using a definitional interpreter. In the current chapter, we evaluate the practical applicability of our approach with respect to whether is able to discover bugs in ING's code generation tool chain. We first describe the setup of the different experiments that we conducted and subsequently we present the results along with their interpretation.

5.1 Setup of the Evaluation

First of all, to evaluate our approach we use the experimental Maverick specifications created by ING and presented in section 2.2.1, namely the account and transaction specifications. To be more precise, we also use two modified versions of the Maverick specifications mentioned above, that still model a simple bank account and a transaction between two accounts, but with the difference that these versions contain more or different language constructs of the Maverick language than those included in listings 2.1, 2.2.

More specifically, in listing 5.1 a Maverick specification that models a transaction is depicted, it uses date/time fields (eg. lines 8, 9) and also expressions that entail operations with date/time fields (line 42). Another example is presented in listing 5.2, the specification illustrated in that listing is an account that contains other language constructs of the Maverick language such as sets and other expressions that in the other versions we did not encounter (eg. lines 8, 18).

Using different versions of Maverick specifications enable us to observe the behavior most of the language constructs that constitute the Maverick language and consecutively to test whether the code generator preserves their behavior in the generated code.

Our evaluation approach can be divided in two main scenarios. In the first scenario, each one of the different versions of the Maverick specifications is given to the code generation tool chain us an input and then the generated code produced is tested by following our approach. In the second scenario, we introduce a bug in the generated code and we then test it using our approach, during which we expect that the introduced bug will be discovered.

```
...
specification {
  fields {
    id: Integer;
    amount: Money;
    from: Account;
    to: Account;
    currentDate: Date;
    currentTime: Time;
    createdOn: DateTime;
    bookedOn: DateTime;
    endOfDayTime: Time;
    failureDate: DateTime;
  }
  ...
  transitions {
    @doc {
      Start a new transaction.
    }
     start(amount: Money, from: Account, to: Account) {
      preconditions {...}
      postconditions {
        amount' == amount;
        from' == from;
        to' == to;
        createdOn' == now;
        currentDate' == now;
        currentTime' == now;
        endOfDayTime' == 17:00;
      }
    }

    @doc{
      Book the transaction.
    }
    book() {
      preconditions {}
      postconditions {
        this.from.withdraw(this.amount);
        this.to.deposit(this.amount);
        bookedOn' == now;
        2018-08-23 - currentDate == 1;
      }
    }
    ...
  }
}
```

Listing 5.1: An example of a transaction specification that incorporates date/time fields and operations

```
...
specification {

  fields {
    accountNumber: IBAN;
    balance: Money;
    country: String;
    elements: Set[Money];
  }
  ...
  transitions {
    @doc {
      Opening an account needs a valid IBAN and some initial
          deposit.
    }
    openAccount[minimalDeposit: Money = EUR 0.00, allowedCountries: Set[String
        ] = {"Netherlands", "Germany"}](initialDeposit: Money, country :
        String) {
      preconditions {
        initialDeposit >= minimalDeposit;
        country in allowedCountries;
      }
      postconditions {
        balance' == initialDeposit;
        elements' == { initialDeposit };
        country' == country;
      }
    }
  ...
  }
}
```

Listing 5.2: An example of an account specification that contains sets

Using the approach we described in section 4, for each of the aforementioned Maverick specifications we generate test cases for both the generated code and our definitional interpreter. Our approach produce test cases in the form that our definitional interpreter accepts, hence they do not need any further processing before we supply with them our interpreter. As our approach incorporates the use of a primitive value randomizer generates paseudo-randomly test cases, we can generate fast a number of test cases, which should be noted that will be different each time that we run our input generator. To give you an idea, we present a test case (listing 5.3), as generated from our method, related with the account specification. As explained and as can be seen in the listing below, a test case is simply a sequence of events that create a new specification instance or that trigger a transition to an already existing specification instance.

An event contains a list of the identity values of a specification instance that this event refers to, the module name of the specification that the instance belongs to, the name of the transition that is triggered and the values of the transition's arguments. To illustrate, for the

creation of an account the identity values of an account is its IBAN (line 1 in listing 5.3 and the values of the transition's arguments is the initial deposit that is needed to open the account (line 4 in listing 5.3). The sequence of events presented in listing 5.3 will trigger the consecutive execution of the following transitions: openAccount, block, unblock, deposit, withdraw, interest, interest, deposit, withdraw, close. In the same way we create test cases for the transaction specification with the difference that first two events will be created that instantiate two accounts. In other words, a sequence of events for the specification transaction would look like following: open an account, open another account, start a transaction between the two accounts, book the transaction.

```
1  CreateInstance(List(IbanV("NL09INGB0604094712")),
2                 "simple_transaction.Account",
3                 "openAccount",
4                 List(MoneyV(CurrencyV("EUR"),MonetaryAmountV(466.33))))
5  TriggerTransition(List(IbanV("NL09INGB0604094712")),
6                    "simple_transaction.Account",
7                    "block",
8                    List()),
9  TriggerTransition(List(IbanV("NL09INGB0604094712")),
10                   "simple_transaction.Account",
11                   "unblock",
12                   List()),
13 TriggerTransition(List(IbanV("NL09INGB0604094712")),
14                   "simple_transaction.Account",
15                   "deposit",
16                   List(MoneyV(CurrencyV("EUR"),MonetaryAmountV(46.43)))),
17 TriggerTransition(List(IbanV("NL09INGB0604094712")),
18                   "simple_transaction.Account",
19                   "withdraw",
20                   List(MoneyV(CurrencyV("EUR"),MonetaryAmountV(38.81)))),
21 TriggerTransition(List(IbanV("NL09INGB0604094712")),
22                   "simple_transaction.Account",
23                   "interest",
24                   List(PercentageV(5.0)),
25 TriggerTransition(List(IbanV("NL09INGB0604094712")),
26                   "simple_transaction.Account",
27                   "interest",
28                   List(PercentageV(5.0)),
29 TriggerTransition(List(IbanV("NL09INGB0604094712")),
30                   "simple_transaction.Account",
31                   "deposit",
32                   List(MoneyV(CurrencyV("EUR"),MonetaryAmountV(7.84)))),
33 TriggerTransition(List(IbanV("NL09INGB0604094712")),
34                   "simple_transaction.Account",
35                   "withdraw",
36                   List(MoneyV(CurrencyV("EUR"),MonetaryAmountV(505.4875)))),
37 TriggerTransition(List(IbanV("NL09INGB0604094712")),
38                   "simple_transaction.Account",
```

```
39                  "close",
40                  List())
```

Listing 5.3: A test case for a specification account

However, as the generated code is a Scala application with a RESTful API the generated test cases need to be further processed. In particular, to invoke the generated code, and more precisely the backend, we create a rest API call for each event included in a test case. For that purpose, we developed a backend invoker where the test cases are converted accordingly to HTTP requests with JSON body that contains the parameters of a transition. For example, the event *CreateIstance* in listing 5.3 will be converted to an HTTP request as following:

http://localhost:8080/simple_transaction/Account/NL09INGB0604094712/OpenAccount with JSON body *{ "initialDeposit": "EUR 60.0" }*

The identity of the instance, the module name of the specification that the instance belongs to and the name of the transition to be triggered construct the URL of the request and the parameters of the transition constitute the JSON body of the request. Equivalently, the rest of the events included in the test case above are converted to HTTP requests that will be sent consecutively in order to invoke the backend.

As is evident from the example above the generated test case covers all the states and transitions of the specification account. It should be noted that other test cases might not cover all the transitions and states of the specification account, as they might include paths from the origin state of an account to its final state without including all the possible transitions. Such an example could be a sequence of events that trigger the following transitions: openAccount, deposit, block, unblock, withdraw, close (i.e. transition interest is not contained).

In addition, to test the behavior of the generated code in conditions that invalid transitions are triggered, we generated test cases which contain a sequence of events that their consecutive execution cannot be successful. To do so we adjust our approach to generate completely random test cases. This leads to the generation of many invalid sequences of events.Two examples of invalid sequence of events are demonstrated in listing 5.4.To illustrate, a transition cannot take place to an account that is in the blocked state, hence an error should be thrown instead (line 6,10 in listing 5.4). Similarly, triggering a withdraw transition for an instance of an account that will result to the negative balance of the account should be interrupted abnormally (i.e. as it will result to a negative balance of an account which is not permitted by the invariants of a specification account, line 27 in listin 5.4).

```
1  /*An invalid sequence of events*/
2  CreateInstance(List(IbanV("NL05INGB8991436658")),
3                 "simple_transaction.Account",
4                 "openAccount",
5                 List(MoneyV(CurrencyV("EUR"),MonetaryAmountV(412.89))))
6  TriggerTransition(List(IbanV("NL05INGB8991436658")),
7                    "simple_transaction.Account",
8                    "block",
```

```
9                 List()),
10 TriggerTransition(List(IbanV("NL05INGB8991436658")),
11                 "simple_transaction.Account",
12                 "withdraw",
13                 List(MoneyV(CurrencyV("EUR"),MonetaryAmountV(262.89)))),
14 TriggerTransition(List(IbanV("NL05INGB8991436658")),
15                 "simple_transaction.Account",
16                 "close",
17                  List())
18
19 /*Another invalid sequence of events*/
20 CreateInstance(List(IbanV("NL85INGB9511369477")),
21              "simple_transaction.Account",
22              "openAccount",
23              List(MoneyV(CurrencyV("EUR"),MonetaryAmountV(214.58))))
24 TriggerTransition(List(IbanV("NL85INGB9511369477")),
25                 "simple_transaction.Account",
26                 "interest",
27                 List(PercentageV(5.0)),
28 TriggerTransition(List(IbanV("NL85INGB9511369477")),
29                 "simple_transaction.Account",
30                 "withdraw",
31                 List(MoneyV(CurrencyV("EUR"),MonetaryAmountV(786.48)))),
32 TriggerTransition(List(IbanV("NL85INGB9511369477")),
33                 "simple_transaction.Account",
34                 "close",
35                 List())
```

Listing 5.4: Two invalid test cases for a specification account

Finally, in order to further evaluate our proposed testing approach for code generators, we introduced manually a bug in the generated code. More specifically, we changed the invariants of a specification account in order to allow a negative balance of an account. Hypothetically, if the code generation tool chain had wrongly translated the invariants of the models to the generated code, then our method should be able to catch this fatal bug of the code generator.

5.2 Comparison between the traces of the generated code and the definitional interpreter

As we pointed out in the previous chapters, the main goal of our approach is to test the code generation tool chain and more specifically to test whether the generated code produced by the code generator preserves the behavior of the model. Accordingly, after we stimulate the generated code and the interpreter with the generated test cases, a comparison between the traces produced by the generated code and the traces produced by the definitional interpreter needs to take place.

To realize that, we compare the two traces after the execution of each event. In more detail, each test case is constituted of a sequence of events, hence after the process of the corresponding event by the definitional interpreter and the generated code a comparison takes place between the instances that have been formed in the current state of the interpreter and the instances that have been formed by the backend application (generated code).

Before we perform the comparison, we convert the instances that have been created by the backend to the representation that is used by our interpreter and described in chapter 3. However, even in the case that the code generator translates correct the model into code, we cannot expect identical behavior in certain type of fields that a state machine instance might contain, such fields are the date/time fields and operations (see an example of a Maverick specification with date/time fields and operations in listing 5.1). Thus, the comparison algorithm in that case should be able to tolerate differences between date/time fields that exist in the interpreter and those that are in the generated code.

As Stürmer et al. argue it is important to use "a notion of sufficiently similar behavior" [49]. Therefore, date/times fields are compared with respect the following definition of similar behavior: Two date/time fields are compatible if their absolute difference is equal or less than the threshold of twenty seconds. Then given this definition of compatibility, we proceed by comparing each state machine instance in the simulation with the state machine in produced by the generated code. The comparison checks whether the fields and state of an instance produced by the generated code conforms to the fields and state of the instance produced by the interpreter.

In case that the comparison fails, then a message is displayed indicating where exactly the comparison failed along with an indication of the value that was expected (produce by our definitional interpreter) and the actual value (produced by the generated code).

5.3 Results

In the current section we present the results of the execution of different test cases. Before we display a result of the conformance testing performed for a particular test case, we first present the test case by mentioning the sequence of transitions that it triggers. It should be mentioned that all the test cases are in the form we presented in section 5.1. Because it is impossible to present the results of all the generated cases for which we tested the code generator, we describe representative examples of test cases along with their results.

We begin with the scenario of introducing a bug in the generated backend by changing the invariant conditions of an account. These invariants state that an account must always have an account balance that is greater or equal to zero euros. In more detail, the invariant block, which contains predicates that must always hold for a specification instance, is translated by the code generator to conditions that are included to both pre- and post- of each transition of a Maverick specification. For instance, the generated code of a withdraw transition of an account specification, as illustrated in listing 5.5, includes in its pre- and post- conditions predicates which assure the amount that will be deducted from the balance of an account will not result to a not negative balance (lines 9,18 in listing 5.5).

```
1
2  case class Withdraw(amount: MMoney) extends AccountCommand {
3      override def preCondition(implicit rc: ReadContext): PreConditionF = {
4        case f =>
5          // Are all variables present? &&
6          condition(f.data.balance.isDefined, """Variable `balance` is not
               defined.""") &&
7          // And the actual preconditions &&
8          condition((amount > net.ing.xlinq.builtin.MMoney.box(EUR(0.00))),
               """amount > EUR 0.00""") &&
9          condition(((f.data.balance.get - amount) >=
               net.ing.xlinq.builtin.MMoney.box(EUR(0.00))), """balance - amount
               >= EUR 0.00""")
10     }
11
12     /*....*/
13
14     // No external invocations
15     override def postCondition(implicit pc: PrevReadContext, rc:
           ReadContext): PostConditionF = {
16       case f =>
17         condition((f.newState.data.balance.get == (f.oldState.data.balance.get
               - amount)), """balance' == balance - amount""") &&
18         condition((f.newState.data.balance.get >=
               net.ing.xlinq.builtin.MMoney.box(EUR(0.00))), """this.balance >=
               EUR 0.00""")
19     }
20
21   }
```

Listing 5.5: The generated code of a withdraw transition

To evaluate our approach we introduce a bug in the aforementioned generated code by putting into comments the lines 9,18 in listing 5.5). This means that we should expect our approach to catch this fatal error of allowing an account to have negative balance after a withdraw transition. This assumption is proved correctly after the run of a few test cases. In particular given the following test case:

1. Open account with IBAN *NL60INGB7304192254* and initial balance of *318.82 EUR*.
2. Withdraw *460.17 EUR* from account with IBAN *NL60INGB7304192254*.
3. Block account with IBAN *NL60INGB7304192254*.
4. Unblock account with IBAN *NL60INGB7304192254*.

After the execution of the second event our interpreter throws an error and does not proceed with the handling of the remaining events. This is illustrated in figure 5.1 where it becomes evident that a *TransitionException* was thrown after the execution of the second

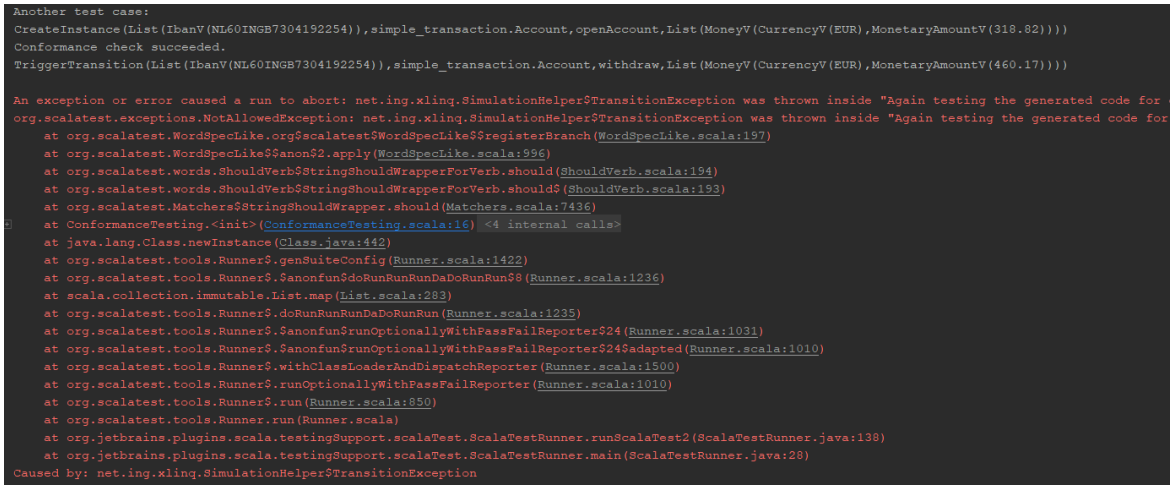


Figure 5.1: An error is thrown from the interpreter after processing the second event.

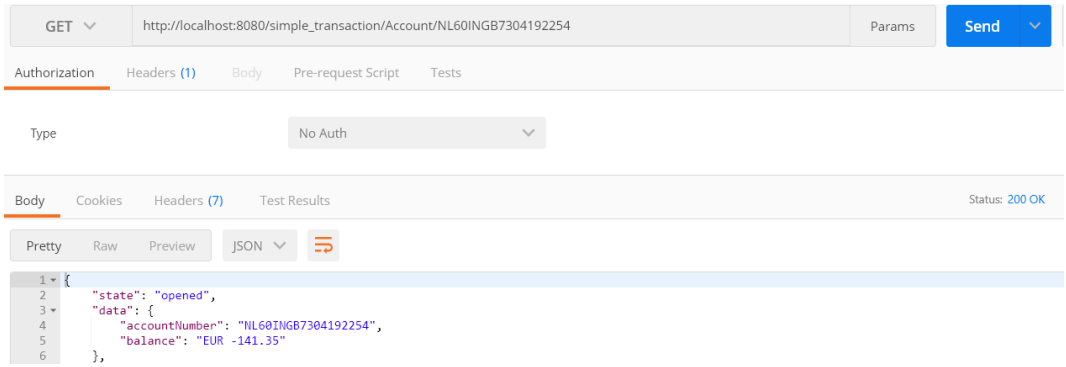


Figure 5.2: Negative balance of an account instance

event (as described above). On the other hand, the generated code process successfully the event. To double check that this is the case we send a *GET* request to retrieve the current state of the aforementioned account (using the Postman tool[1]). Indeed, as shown in figure 5.2 the account is in state *opened* with a negative balance which should have been prevented.

After reverting back the change that we made in the generated code, we proceed with testing thoroughly the backend as generated by the code generator and without any modification from us.

Our approach, after the run of multiple different test cases, revealed an already existing bug in the generated system. More particularly, given the following sequence of events:

1. Open account with IBAN *NL37INGB6374353384* and initial balance of *705.44 EUR*.

2. Withdraw *26.66 EUR* from account with IBAN *NL37INGB6374353384*.

[1]https://www.getpostman.com/

```
Another test case:
CreateInstance(List(IbanV(NL37INGB6374353384)),simple_transaction.Account,openAccount,List(MoneyV(CurrencyV(EUR),MonetaryAmountV(705.44))))
Conformance check succeeded.
TriggerTransition(List(IbanV(NL37INGB6374353384)),simple_transaction.Account,withdraw,List(MoneyV(CurrencyV(EUR),MonetaryAmountV(26.66))))
Conformance check succeeded.
TriggerTransition(List(IbanV(NL37INGB6374353384)),simple_transaction.Account,withdraw,List(MoneyV(CurrencyV(EUR),MonetaryAmountV(7.43))))
Conformance check succeeded.
TriggerTransition(List(IbanV(NL37INGB6374353384)),simple_transaction.Account,interest,List(PercentageV(5.0)))
After the execution of the event: TriggerTransition(List(IbanV(NL37INGB6374353384)),simple_transaction.Account,interest,List(PercentageV(5.0))).
The conformance test failed for the following reasons:
Conformance failed in field with name: balance. Value expected: MoneyV(CurrencyV(EUR),MonetaryAmountV(704.9175)), Actual value: MoneyV(CurrencyV(EUR),MonetaryAmountV(4028.10))
TriggerTransition(List(IbanV(NL37INGB6374353384)),simple_transaction.Account,withdraw,List(MoneyV(CurrencyV(EUR),MonetaryAmountV(704.9175))))
After the execution of the event: TriggerTransition(List(IbanV(NL37INGB6374353384)),simple_transaction.Account,withdraw,List(MoneyV(CurrencyV(EUR),MonetaryAmountV(704.9175)))).
The conformance test failed for the following reasons:
Conformance failed in field with name: balance. Value expected: MoneyV(CurrencyV(EUR),MonetaryAmountV(0.0000)), Actual value: MoneyV(CurrencyV(EUR),MonetaryAmountV(3323.18))
TriggerTransition(List(IbanV(NL37INGB6374353384)),simple_transaction.Account,close,List())
```

Figure 5.3: Sequence of events that indicate a bug after the execution of transition *interest*.

3. Withdraw *7.43 EUR* from account with IBAN *NL37INGB6374353384*. *Perform the interest transition on the same account with an interest rate of %5.*

4. Withdraw *704.9175 EUR* from account with IBAN *NL37INGB6374353384*.

5. Close account with IBAN *NL37INGB6374353384*.

As it can be seen in figure 5.3, the conformance check first fails after the execution of the transition *interest* and continues to fail after the execution of the remaining events. Having a closer look in that case leads us to the observation that the error lies in the multiplication of the percentage with the amount of money. This error cause a difference in the balance of the two account instances (i.e. instance of the account resulted from the definitional interpreter versus the instance resulted from the generated code). Subsequently, this difference in the balance of the account causes the next transitions to also fail in the conformance check.

An interest transition multiplies the balance of an account with a percentage indicating the current interest rate and adds this result to the current balance of an account. Taking into account this and using our approach we can understand that the bug lies in the multiplication of a *percentage* type with a money field. Therefore, the first step in the investigation of the failing test case is to search where the multiplication of a percentage type with a money type is defined in the generated code. This leads us to actually locate the bug in the generated *Mul.scala* file. In particular, the bug is located in one of the overloading methods of the object *Mul* (line 3 in listing 5.6). This is opposed to the correct definition of a multiplication that is given by our definitional interpreter (line 9 in listing 5.6). In particular, the implementation of "multiplication by percentage" is off by a factor of 100.

```
1 //Definition of Money * Percentage in the generated code
2   def apply(percentage: Dimensionless, d: MMoney): MMoney = Mul(d, percentage)
3   def apply(money: MMoney, percentage: Dimensionless): MMoney = money match {
4     case SomeMoney(amount) => box(amount * percentage.toPercent)
5     case ZeroMoney => ZeroMoney
6   }
7
8 //Definition of Money * Percentage given by our definitional interpreter
9 case MoneyV(CurrencyV(lc), MonetaryAmountV(lv)) => interp(rhs, program, env,
      currentState) match {
```

```
scala> val p: Dimensionless = Percent(5)
p: squants.Dimensionless = 5.0 %

scala> val pp: Double = p.toPercent
pp: Double = 5.0
```

Figure 5.4: Using Squants library in Scala REPL

```
10          case RealV(rv) => MoneyV(CurrencyV(lc), MonetaryAmountV(lv *
                BigDecimal(rv)))
11          case IntegerV(rv) => MoneyV(CurrencyV(lc), MonetaryAmountV(lv *
                BigDecimal(rv)))
12          case PercentageV(rv) => MoneyV(CurrencyV(lc), MonetaryAmountV(lv *
                BigDecimal(rv / 100)))
13          case MoneyV(CurrencyV(rc), MonetaryAmountV(rv)) =>
14            if (lc == rc)
15              MoneyV(CurrencyV(lc), MonetaryAmountV(lv * rv))
16            else
17              throw IllegalExpressionException("Illegal expression. You cannot
                    multiply different currencies.")
18          case _ => throw IllegalExpressionException("Illegal expression for
                multiplication")
19        }
```

Listing 5.6: Definition of multiplication between a percentage and a money type

To understand better why this is the case we need to dive into the generated code. More specifically, the generated application uses the library *Squants*[2] to deal with Money and Percentage types. Using the Scala REPL we can understand how exactly the open-source library *Squants* works in line 4 (listing 5.6), this helps us to see immediately why the multiplication is wrong. The figure 5.4 shows that the expression *percentage.toPercent* converts a percent to a double representation, however, without dividing with 100 and this is what makes the "multiplication by percentage" wrong.

Furthermore, we performed several runs of test cases related with the modified versions of the specifications account and transaction. The modified version of the specification transaction helps us to test whether the date/time fields and their operations behave sufficient similar with the definition given by our definitional interpreter (i.e. based on our definition of compatibility). The differences that we observed during the run of the respective test cases were only some milliseconds indicating that the generated system conforms to the expected result. A simple example with the modified version of the transaction specification is presented in figure 5.5 where first two accounts are created and then a transaction is initiated between them.

Similarly, the modified version of the specification account allows us to test the behavior of other data structures that the Maverick language supports such as *set* and *map*. Based on

[2]http://www.squants.com/

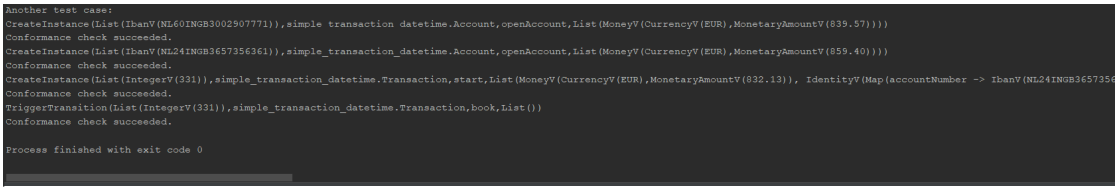


Figure 5.5: Testing the modified version of the specification Transaction (with date and time fields)

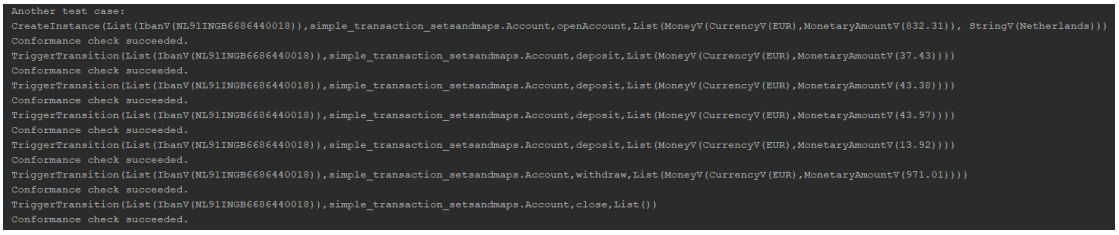


Figure 5.6: Testing the modified version of the specification Account (with date and time fields)

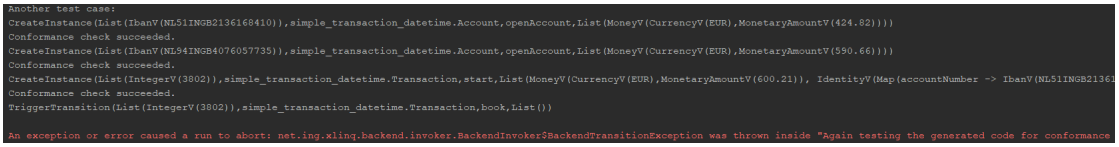


Figure 5.7: Handling an invalid test case

the results of the test cases runs, these data structures also seem to behave as expected (eg. figure 5.6).

Table 5.1: Summary of results per specification for a valid execution of test cases

Specification name	**Test cases generated (#)**	**Conf. succeeded (#)**	**Conf. failed (#)**
Account	20	13	7
Account (sets and maps)	20	11	9
Transaction	1	1	0
Transaction (date/time)	1	1	0

A summary of the results for a valid execution of test cases is given in table 5.1 (containing only valid test cases). The results displayed in that table are given for one run of valid test cases. It should be clarified that the number of the generated test cases is directly related with the number of transitions and states that a Maverick specification contains and the maximum length of a path from an origin to a destination state (the length that is used

in abstract test case generation). For the table presented in 5.1 the test cases were generated by taking into account a maximum length of 8 events/transitions in a sequence. In addition, the test cases that failed to the conformance check had a common cause that lead to that failure. That is the transition *interest*, which as we showed earlier contains a bug related with the "multiplication by percentage". Finally, it should be noted that the number of test cases that succeeded the conformance check might differ in another run of valid test cases. The reason behind this is that in each run different test cases will be generated which might contain less or more event sequences that contain an event related with the faulty *interest* transition.

In addition, we tested several times the code generation tool chain with invalid test cases to see if the generated code handles correctly these cases. An example of such invalid test case we presented earlier where we introduced a bug in the generated code to check if our interpreter will catch the error. If we do not modify the generated code we see that the generated system reacts in the correct way by interrupting the execution of the transition.

Another example can be seen in figure 5.7, where first two accounts are created and then a transaction is initiated between the two accounts. However, the amount that is being transferred from the one account to the other will cause the balance of the "sender" to become negative. Therefore, we should except that the external invocation of the withdraw transition in the "sender" account will fail, causing also the primary transition to fail. This means that the *book* transition should not be completed successfully. Indeed the generated code handles correct also this invalid test case and therefore throws an error. Thus, the atomicity of a transition seems to be preserved in the generated system, meaning that when a secondary transition fails then the primary transition fails too.

Chapter 6

Related Work

In this chapter we discuss relevant research that focuses on testing code generators. In addition, we briefly discuss approaches which try to verify whether a code generator works correctly. We divide these works into two main categories: testing approaches and formal approaches. It should be mentioned that as "formal" we consider approaches that aim to prove properties of a system using some sort of mathematical reasoning and which try to guarantee that these properties hold in all cases. On the other hand, testing approaches try to detect bugs or prove that a property does not hold (or holds) under a sample of all possible cases. However, it could be the case that a testing approach uses as its basis the formal methods of mathematics to test a code generator (eg. formal specification of the semantics).

6.1 Testing approaches for code generators

In the current section we describe approaches that are directly related with our approach as they have the exact same purpose, that is to test code generators. As we explained in our introductory section, code generators are not typical software artifacts as their input and output are objects that also have execution semantics besides just syntactic structure. Our approach focuses on testing generated code's behavior when it is executed, hence, we do not discuss approaches that focus mainly on syntactic constructs of a code generator (eg. grammar-based testing [26, 29]).

Rajeev et al. do not focus only on the syntactical aspects of a code generator but similarly to our approach they also try to test code generators from a semantics perspective [37, 40, 41, 42]. In particular, in the aforementioned papers they propose an approach for generating test models that take into account both syntactic and semantic aspects of the modelling language.

The input to their method is a formal meta-model that contains syntactic and semantic definitions of the modelling language expressed using inference rules along with a test specification that contains coverage criteria regarding which syntactic and semantic rules should be tested. Their method constructs an inference tree representing the semantic scenario to be tested and uses a custom constraint solver to solve the constraints extracted from the inference tree. In that way the method generates syntactically valid test models along with

inputs/outputs that will make the model capable of exhibiting the scenario. These test models are fed to the code generator and the generated program is then executed using as inputs those that were generated along with the test model. Finally, the output of the execution is compared for conformance to the output that was also generated with the test model.

Their method is similar to our approach in terms that both approaches focus on the execution semantics of the modelling language and in the fact that their generated test suite also proceeds in a back-to-back fashion. However, our approach differs in many ways, the two main differences are: that our approach do not take into account syntactic aspects of the modelling language as we do not generate test models, and secondly their approach is based on formal methods (eg. inference rules) whereas our approach is based on a definitional interpreter.

Other works that also employ the idea of back-to-back testing for code generators are those of Sturmer et. al. [46, 47, 49]. The approach of Stürmer et al. mainly focuses on the automatic generation of test cases (i.e. test models, inputs and outputs for the models) using graph transformation rules. In particular, they formalize the transformation rules implemented in the code generation tool chain as graph transformation rules which serve as blueprint for the test model generation. Before they proceed to back-to-back-testing of a model and the corresponding generated code they generate input and expected outputs for the models based on structural test design techniques on both model and code level. During the comparison of two outputs they use a definition of sufficient similar behavior to tolerate the differences in the value and in the time domain. This workflow is also integrated by Conrad into the certification process defined by the safety standard IEC 61508-3 [4, 9].

The aforementioned testing approach has many similarities to our testing approach such us the use of back-to-back-testing and the notion of a sufficient similar behavior in order to tolerate expected differences between the generated code and the model. However, their approach relies on transformation rules implemented in the code generator which means that this requires details of the implementation of the code generation tool chain. In contrast to their approach, our testing approach does not require any specific details of the code generator's source code. In addition to that, we develop a definitional interpreter to get a concise definition of the semantics of the modelling language and its expected behavior.

Another testing approach that tests the transformation performed by a code generator from a semantic perspective is given by Jörges et al.[21]. Similarly with others and to our method they presented a back-to-back-testing approach of the Genesys code generator framework which is based on jABC (i.e. a framework for model driven development)[22].

More specifically, the authors exploit the fact that the models (Service Logic Graphs) are composed by components (Service Independent Building Blocks) that upon their execution leave a unique footprint, the concatenation of these footprints represents a particular trace which is then compared against the traces that the execution of the generated code produced. This testing approach, in contrast to our, can perform testing on multiple meta-levels as the code generators are also created in a model-based manner. In addition, we also leverage the execution of the model, however, we do so by using a definitional interpreter.

Stoel et al. use an interactive simulation of Rebel specifications by translating an event that a user wants to perform into Satisfiability Modulo Theories (SMT) formulas [45]. Subsequently, they use an SMT solver to check whether the event's execution is feasible, if it

is not they inform accordingly the user in order to try a different event. Finally, the traces produced by the simulation are replayed to the generated system which produces an output that is then compared to the simulation's output. Our approach also exploits the simulation of the model by, in contrast to their approach, using a definitional interpreter. Also we do not develop an interactive simulation but on the contrary we generate test cases capable of exhibiting different scenarios on the model level which serve as the input for our simulation.

In another work, Yang et al. are testing C compilers using randomly synthesized programs [57]. In particular, they exploit a similar idea as we do, that is if one has two or more implementations of the same specification, then all implementations must produce the same result from the same valid input otherwise one of the implementation is faulty. However, we do not generate programs and we use a definitional interpreter as another implementation of the modelling language instead of multiple compilers.

6.2 Formal approaches for verifying code generators

In this section we briefly discuss approaches that try to formally prove that the generated code has the same semantics as the model or aim to prove that a property of the model holds in the generated code for every valid input. These approaches differ inherently from our approach us we develop a testing approach that from its nature can only provide partial guarantees for the correctness of the code generation tool chain.

One category of verification approaches is the use of theorem proving techniques [28, 19]. Although these approaches constitute a very strong approach in proving the correctness of code generators, in practice their use in an industrial context is still considered infeasible. This is due to several reasons, some of them including the complexity that entails the construction of such proofs, the effort needed for the use of theorem proving tools and last but not least a model transformation in an industrial context might contain hundreds of rules which besides the increased difficulty, create scalability issues [36].

Model checking is another formal method that is used to prove whether a system is working correctly. Model checkers exploit state-space exploration to specify all the possible paths of a system, having executed all paths can prove if a system is correct [30]. Basically the goal of model checking is to find a reachable state in which a property of the model does not hold (known also as safety property). In case that a state is reached where this property does not hold then a counterexample is returned with the trace that leads to that state. Model checking can be applied either to the generated system or to the source model by translating properties of the model that should be preserved to some formal language. Examples of such approaches constitute the works of Staats et al. [44] and Rahim et al. [1].

Stoel et al. uses bounded model checking, which is a variant of model checking that searches a counterexample in executions whose length is bounded by some integer k [5], to check whether there are states where the invariants modeled with the Rebel specification language do not hold [45]. They do so by mapping Rebel to SMT formulas with which they are trying to verify that the invariants hold for traces up to length k.

However, model checking is an incredibly expensive method. Taking into account all the possible inputs that a system accepts along with the failures which can experience, running

a model checker makes it an extremely expensive procedure in terms of time and resources [30].

Chapter 7

Conclusion and Future Work

In this last chapter we give a summary of this thesis and we discuss the project's contributions. Then, we give some suggestions for further research and improvements of our approach.

7.1 Conclusion

In this thesis we focused on testing generated code's behavior rather than its concrete syntax. In the introduction we already stressed the importance of testing code generators from a semantic perspective and we presented the main research question of the current thesis: *how could Code Generators be tested against a definitional interpreter?*

To answer this question, we described and demonstrated our method that leverages a *definitional interpreter* to test whether the generated backend application is generated properly from ING's Maverick specifications. In chapter 2 we gave a high-level description of our testing approach that employs a similar approach to back-to-back testing based on the execution semantics of the Maverick DSL given by a definitional interpreter. In addition, we provided an overview of ING's code generators and we described the semantics of the Maverick DSL.

Consequently, we described in chapter 3 the implementation of the main building block of our approach, that is the development of the definitional interpeter which has the explicit goal to provide us with a semnatic specification of the Maverick DSL. On top of that we presented our approach for generating test sequences that will be given as input to both the definitional interpreter and generated code. This enable us firstly to proceed in back-to-back fashion and secondly to increase our confidence about the correctness of the generated code as well as the practicability of our approach.

During the evaluation part of our approach (chapter 5), we tested the generated backend application using invalid and valid execution. With the invalid execution we tested the generated system whether it behaves correct in cases that the input triggers a sequence of events that should not be possible according to the specification. With the valid execution we tested whether the traces produced by the execution of the generated code conform to the traces expected from the definitional interpreter.

With our testing approach we found bugs that were manually injected in the generated code but also an already existing bug in the generated system. This proves that the proposed approach already can be efficiently used to detect bugs in the system that was generated by the code generation tool chain. In particular, we demonstrated that our approach is capable of finding faults related with wrongly generated pre/post conditions or invariants of the model. An example of this is given in section 5.3 where we intentionally changed the generated pre and post conditions of a *withdraw* transition to allow an account to have a negative balance. In addition, our approach detected an already existing error in the implementation of the multiplication of a percentage type field with a money type field.

No faults were found when we tested the capability of the specification instances to interact with each other. More specifically, a transition's atomicity seems to be preserved in the generated code, in the sense that when a secondary transition fails the primary transition fails too. However, our approach does not support concurrent interactions between specification instances and therefore we cannot draw conclusions about the behavior of these interactions on concurrent situations.

Moreover, compared to other related works that we described in chapter 6 our method distinguishes itself mainly through the use of the definitional interpreter. As we showed in chapter 3, the development of a definitional interpreter can provide us with a concise definition of the behavior of the modelling language by focusing on the simplicity and clarity of the implementation. In contrast to formal approaches, our approach is capable of addressing complexity and scalability issues. For instance, as modelling languages usually tend to evolve by adding or alternating language constructs, code generators have also to be updated to resemble these changes. In such cases our testing approach could easily reflect this change by possibly just adding a new case in the large match expression that we presented in chapter 3 and proceed with implementing the definition of the newly added constructs.

Finally, based also on our experience within ING, such approach produces a specification of the semantics of the modelling language which can be easily communicated along the developers, especially compared to semantic formalisms that are used in formal approaches. Most of the developers are expert programmers which makes it easier for them to understand a semantic specification given by a definitional interpreter.

7.2 Future work

In this section we suggest some directions for further research and improvements of our approach based on our findings and the limitations of our approach.

As we explained earlier, one of the limitations of our approach is that does not test the generated code in situations that we have concurrent interactions between several specification instances as this is not supported through the current simulation of the model. Because concurrent interactions is mostly the case in production environments, it is very important to focus on modifying the current simulation of the model in order to support concurrent interactions between the specification instances.

Furthermore, our approach is not focusing on developing a novel method for generating

test cases that can be given to both the interpreter and generated system. Our approach is a testing method, which means can only give partial guarantees for the correctness of the generated code. As a result, further research should be conducted in combining our method with a more sophisticated approach in generating test cases that might lead to the exposure of new bugs.

Finally, the Maverick specifications used during the experiments that we conducted, are only simple examples of an account and transaction specification. Although, we were able to detect bugs and prove the value of our approach using these specifications, normally such specifications will be far more complex within a banking environment. Therefore, future work should use more complex specifications along with our approach.

Bibliography

[1] Lukman Ab Rahim and Jon Whittle. Verifying semantic conformance of state machine-to-java code generators. In *International Conference on Model Driven Engineering Languages and Systems*, pages 166–180. Springer, 2010.

[2] Benoit Baudry, Trung Dinh-Trong, Jean-Marie Mottu, Devon Simmonds, Robert France, Sudipto Ghosh, Franck Fleurey, and Yves Le Traon. Model transformation testing challenges. In *ECMDA workshop on Integration of Model Driven Development and Model Driven Testing.*, 2006.

[3] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[4] Ron Bell. Introduction to iec 61508. In *Proceedings of the 10th Australian workshop on Safety critical systems and software-Volume 55*, pages 3–12. Australian Computer Society, Inc., 2006.

[5] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, Yunshan Zhu, et al. Bounded model checking. *Advances in computers*, 58(11):117–148, 2003.

[6] Bettina Buth, Karl-Heinz Buth, Martin Fränzle, Burghard v Karger, Yassine Lakhneche, Hans Langmaack, and Markus Müller-Olm. Provably correct compiler development and implementation. In *International Conference on Compiler Construction*, pages 141–155. Springer, 1992.

[7] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE transactions on software engineering*, (3):178–187, 1978.

[8] David R Christiansen, Klaus Grue, Henning Niss, Peter Sestoft, and Kristján S Sigtryggsson. An actuarial programming language for life insurance and pensions. In *Proceedings of 30th International Congress of Actuaries. Washington DC: International Actuarial Association*, 2013.

[9] Mirko Conrad. Testing-based translation validation of generated code in the context of iec 61508. *Formal Methods in System Design*, 35(3):389–401, 2009.

[10] Maulik A Dave. Compiler verification: a bibliography. *ACM SIGSOFT Software Engineering Notes*, 28(6):2–2, 2003.

[11] Ibrahim K El-Far and James A Whittaker. Model-based software testing. *Encyclopedia of Software Engineering*, 2002.

[12] Arthur D Friedman and Premachandran R Menon. *Fault detection in digital circuits*. Prentice Hall, 1971.

[13] Neelam Gupta, Aditya P Mathur, and Mary Lou Soffa. Automated test data generation using an iterative relaxation method. *ACM SIGSOFT Software Engineering Notes*, 23 (6):231–244, 1998.

[14] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[15] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[16] Gerard J Holzmann and William Slattery Lieberman. *Design and validation of computer protocols*, volume 512. Prentice hall Englewood Cliffs, 1991.

[17] John Hutchinson, Mark Rouncefield, and Jon Whittle. Model-driven engineering practices in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 633–642. ACM, 2011.

[18] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of mde in industry. In *Proceedings of the 33rd international conference on software engineering*, pages 471–480. ACM, 2011.

[19] Nassima Izerrouken, Xavier Thirioux, Marc Pantel, and Martin Strecker. Certifying an automated code generator using formal tools: Preliminary experiments in the geneauto project. In *European Congress on Embedded Real-Time Software (ERTS), Toulouse*, volume 29, pages 2008–01, 2008.

[20] Leslie A Johnson et al. Do-178b, software considerations in airborne systems and equipment certification. *Crosstalk, October*, 199, 1998.

[21] Sven Jörges and Bernhard Steffen. Back-to-back testing of model-based code generators. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 425–444. Springer, 2014.

[22] Sven Jörges, Tiziana Margaria, and Bernhard Steffen. Genesys: service-oriented construction of property conform code generators. *Innovations in Systems and Software Engineering*, 4(4):361–384, 2008.

[23] Sarmen Keshishzadeh and Arjan J Mooij. Formalizing dsl semantics for reasoning and conformance testing. In *International Conference on Software Engineering and Formal Methods*, pages 81–95. Springer, 2014.

[24] Thomas Koshy. *Discrete mathematics with applications*. Elsevier, 2004.

[25] Nikolai Kosmatov, Bruno Legeard, Fabien Peureux, and Mark Utting. Boundary coverage criteria for test generation from formal models. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 139–150. IEEE, 2004.

[26] Ralf Lämmel and Wolfram Schulte. Controllable combinatorial coverage in grammar-based testing. In *IFIP International Conference on Testing of Communicating Systems*, pages 19–38. Springer, 2006.

[27] David Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[28] Dirk Leinenbach, Wolfgang Paul, and Elena Petrova. Towards the formal verification of a c0 compiler: Code generation and implementation correctnes. In *null*, pages 2–12. IEEE, 2005.

[29] Peter M. Maurer. Generating test data with enhanced context-free grammars. *Ieee Software*, 7(4):50–55, 1990.

[30] Caitie McCaffrey. The verification of a distributed system. *Communications of the ACM*, 59(2):52–55, 2016.

[31] Parastoo Mohagheghi and Vegard Dehlen. Where is the proof?-a review of experiences from applying mde in industry. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 432–443. Springer, 2008.

[32] Chamin Nalinda and Chamath Keppitiyagama. Domain specific language for specifying operations of a central counterparty. In *Advances in ICT for Emerging Regions (ICTer), 2017 Seventeenth International Conference on*, pages 1–8. IEEE, 2017.

[33] Jeff Offutt and Aynur Abdurazik. Generating tests from uml specifications. In *International Conference on the Unified Modeling Language*, pages 416–429. Springer, 1999.

[34] David Owen, Dejan Desovski, and Bojan Cukic. Random testing of formal software models and induced coverage. In *Proceedings of the 1st international workshop on Random testing*, pages 20–27. ACM, 2006.

[35] Gordon D Plotkin. A structural approach to operational semantics. 1981.

[36] Lukman Ab Rahim and Jon Whittle. A survey of approaches for verifying model transformations. *Software & Systems Modeling*, 14(2):1003–1028, 2015.

[37] AC Rajeev, Prahladavaradan Sampath, KC Shashidhar, and S Ramesh. Cogente: A tool for code generator testing. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 349–350. ACM, 2010.

[38] John C Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference-Volume 2*, pages 717–740. ACM, 1972.

[39] Krishan Sabnani and Anton Dahbura. A protocol test generation procedure. *Computer Networks and ISDN systems*, 15(4):285–297, 1988.

[40] Prahladavaradan Sampath, AC Rajeev, S Ramesh, and KC Shashidhar. Testing model-processing tools for embedded systems. In *Real Time and Embedded Technology and Applications Symposium, 2007. RTAS'07. 13th IEEE*, pages 203–214. IEEE, 2007.

[41] Prahladavaradan Sampath, AC Rajeev, KC Shashidhar, and S Ramesh. How to test program generators? a case study using flex. In *Software Engineering and Formal Methods, 2007. SEFM 2007. Fifth IEEE International Conference on*, pages 80–92. IEEE, 2007.

[42] Prahladavaradan Sampath, AC Rajeev, S Ramesh, and KC Shashidhar. Behaviour directed testing of auto-code generators. In *2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pages 191–200. IEEE, 2008.

[43] David A Schmidt and Denotational Semantics. A methodology for language development. 1997.

[44] Matthew Staats and Mats PE Heimdahl. Partial translation verification for untrusted code-generators. In *International Conference on Formal Engineering Methods*, pages 226–237. Springer, 2008.

[45] Jouke Stoel, Tijs van der Storm, Jurgen Vinju, and Joost Bosman. Solving the bank with rebel: on the design of the rebel specification language and its application inside a bank. In *Proceedings of the 1st Industry Track on Software Language Engineering*, pages 13–20. ACM, 2016.

[46] Igno Sturmer and Mirko Conrad. Test suite design for code generation tools. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 286–290. IEEE, 2003.

[47] Ingo Stürmer and Mirko Conrad. Code generator testing in practice. In *GI Jahrestagung (2)*, pages 33–37, 2004.

[48] Ingo Stürmer, Daniela Weinberg, and Mirko Conrad. Overview of existing safeguarding techniques for automatically generated code. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–6. ACM, 2005.

[49] Ingo Stürmer, Mirko Conrad, Heiko Doerr, and Peter Pepper. Systematic testing of model-based code generators. *IEEE Transactions on Software Engineering*, 33(9): 622–634, 2007.

[50] Mark Utting and Bruno Legeard. *Practical model-based testing: a tools approach*. Elsevier, 2010.

[51] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[52] Mladen A Vouk. Back-to-back testing. *Information and software technology*, 32(1): 34–45, 1990.

[53] Stephan Weißleder. Simulated satisfaction of coverage criteria on uml state machines. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 117–126. IEEE, 2010.

[54] Stephan Weißleder and Bernd-Holger Schlingloff. Deriving input partitions from uml models for automatic test generation. In *International Conference on Model Driven Engineering Languages and Systems*, pages 151–163. Springer, 2007.

[55] Elaine J. Weyuker and Bingchiang Jeng. Analyzing partition testing strategies. *IEEE Transactions on software Engineering*, (7):703–711, 1991.

[56] James A Whittaker and Michael G Thomason. A markov chain model for statistical software testing. *IEEE Transactions on Software engineering*, 20(10):812–824, 1994.

[57] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *ACM SIGPLAN Notices*, volume 46, pages 283–294. ACM, 2011.

[58] Justyna Zander, Ina Schieferdecker, and Pieter J. Mosterman. *Model-Based Testing for Embedded Systems*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2011. ISBN 1439818452, 9781439818459.