```
?SUCHTHAT(Example, generate_examples(Model),
          heuristic(Example) == good).
```

Heuristics for generating good examples using QuickCheck

Master's thesis in Algorithms, Languages and Logic

SEBASTIAN IVARSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2016

Master's thesis

Heuristics for generating good examples using QuickCheck

SEBASTIAN IVARSSON

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2016

Heuristics for generating good examples using QuickCheck
SEBASTIAN IVARSSON

Supervisor: John Hughes, Department of Computer Science and Engineering, and Quviq AB
Examiner: Patrik Jansson, Department of Computer Science and Engineering

Master's Thesis
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Department of Computer Science and Engineering
Gothenburg, Sweden 2016

Heuristics for generating good examples using QuickCheck
SEBASTIAN IVARSSON
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

As software systems grow more complex, the need for advanced testing grows with them. To thoroughly test modern software, automated tools are often used to relieve developers of the manual labour of writing tests. This thesis seeks to improve one such tool, QuickCheck, more specifically the Erlang version from Quviq AB. In a recent research project an extension to QuickCheck was developed at Quviq and Chalmers, in the form of a prototype tool called FindExamples. The purpose of the tool is to generate interesting examples of the behaviour of a program from a state machine specification using a heuristic. In this thesis, the tool has been refined to be easier to use, and integrated into one of Quviq's products, QuickCheck CI. Furthermore, the heuristic used by the tool to select good examples has been analyzed, and a few new alternatives have been proposed and tested. A combination of these new ideas along with the original heuristic has shown some promise, both in testing during development and in a small experiment involving 22 students using examples to predict the output of a program. In the experiment, the participants using the examples generated by the developed heuristic were able to outperform those given a set of examples generated with full expression coverage.

Keywords: Erlang, QuickCheck, Property-based testing, Feature-based testing, Good examples

Acknowledgements

First, I would like to thank my supervisor, John Hughes, for his help throughout this thesis, and for proposing the initial idea of the topic. Furthermore, the ideas and comments from both Thomas Arts and Alex Gerdes from Quviq and Chalmers were helpful in giving a different viewpoint on some of the issues faced. Also, I am very thankful for the early introduction to functional programming in the undergraduate programme by Koen Claessen, and for the continued inspiration in the follow-up courses in the master's programme by Patrik Jansson, Mary Sheeran and John Hughes.

Sebastian Ivarsson, Gothenburg, June 2016

Contents

List of Figures

List of Tables

List of Code Listings

1 Introduction

To put the work of this thesis into context, this chapter introduces the background and purpose of the project, along with a presentation of some of the work that relates to the concepts handled in the thesis.

In the following chapters, the theory behind QuickCheck and the related tools is presented in Chapter 2, the work carried out is described in Chapter 3, and the results obtained are presented in Chapter 4. Finally, the conclusions drawn throughout the work are described in Chapter 5.

1.1 Background

As software systems grow larger, the need for quick, automatic and systematic testing grows with them. The larger the system, the more complex it gets, making it impossible for humans to keep track of its behaviour and whether the system is following its specification or not. Testing is necessary to try to enforce the specification on the implementation, but a large amount of manual work is required to create test suites that properly test a system.

Writing and maintaining such test suites can be a daunting task, in some cases as demanding as maintaining the program itself. It is therefore helpful to use automated tools to aid in testing, to avoid the burden of specifying each test case manually. One such tool that has been very successful is QuickCheck [1].

QuickCheck was originally developed by Claessen and Hughes for Haskell in 1999, but has been ported to several other languages since then. The version that is relevant to this thesis is the one written in Erlang created by Quviq AB, founded in 2006 by John Hughes and Thomas Arts[1]. While the original Haskell version focuses primarily on pure functional code, Erlang QuickCheck (EQC) has been extended with the ability to also test stateful code, by providing an interface for defining state machines modeling the state of the system under test [2]. Another extension is the ability to also test C-code, which opens up for a large number of possible code bases to use QuickCheck on.

[1]`http://www.quviq.com`

1.1.1 Model based testing

When testing stateful code, it is not only the commands and the input to the program that are interesting, but also the order in which they are given. In order to test such code, one must specify the commands to be tested, and what the expected output of each command will be. For large systems, this quickly becomes infeasible, and another approach is needed. Model based testing solves this issue by introducing a simplified model of the system, known to be correct, which is used to generate test cases. After constructing such a model, it can be used to obtain traces, or execution paths that can be used as tests for the real system. These traces can either be generated by hand, or automatically through randomization. This allows the tests to capture unforeseen behaviour, that the developer and tester may have overlooked.

1.1.2 Random testing

In order to completely check the correctness of a system by testing, one would have to write tests for every possible execution path. Of course, doing this in the general case is impossible, as there may be an infinite number of paths. The purpose of testing is to find a good subset of scenarios to use as tests. By letting a developer choose these subsets, it is very likely that some interesting test cases are overlooked. Furthermore, it is very hard to maintain manually written test suites, as a single change in the API may force changes in several tests. To avoid this problem, one can use tests that are randomly generated. Such tests are much less predictable, and may find hard-to-think-of errors, that a human would perhaps never even consider. Random tests also have the advantage that their generators are much more concise, and are less likely to contain duplicated code, making them easier to maintain.

1.1.3 QuickCheck

QuickCheck is a property-based, randomized testing tool that has been successfully used for many years. A few examples include using the Haskell version to test large applications such as the window manager XMonad [3], and the version control system Darcs [4]. For the Erlang version, bugs have been found in real systems both within the telecom- [2] and automotive [5] industry. Another interesting use is within the medical field, where QuickCheck was used to test an organ position tracker [6].

MoreBugs and FindExamples

To further develop QuickCheck, two prototype tools building on it and its accompanying library were developed by Quviq and Chalmers during the EU project Prowess [7]. The first of these, called MoreBugs, is a prototype of how QuickCheck testing could be made more efficient. It tries to avoid discovering the same bug multiple times, by keeping track of what kind of tests have been run previously, and then avoids generating these again in order to find new bugs during the same run.

Also developed by Quviq and Chalmers during the Prowess project is the prototype tool FindExamples, that builds on MoreBugs to provide examples of the correct specified behaviour of the system. The purpose of these examples is to serve as an illustration of what the abstract QuickCheck properties test, as the abstract properties are easier to understand when complemented with good examples. Furthermore, the examples can work as unit tests, and since they are designed to be easily understood, they can give an indication to the user of what sort of behaviour is actually tested.

In general, examples are used in most fields to simplify the understanding of abstract properties or concepts, from mathematics to presenting papers [8]. For QuickCheck, this is particularly true when it comes to state machine testing, where an entire module makes up the model used to specify the intended behaviour of the system under test, and thus concrete examples can aid greatly in understanding what is actually tested with the model.

QuickCheck CI

As a way of providing open source developers with powerful testing tools for Continuous Integration, Quviq provides the web-service QuickCheck CI[2]. It allows the maintainer of an open-source GitHub repository to connect their repository and have their Erlang or C-code tested with the full Erlang QuickCheck library. The results from running the tests are then published on the web page, including the output from QuickCheck, but also coverage statistics and a history of all the builds run.

1.2 Purpose

The purpose of this thesis was divided into two parts, where the aim was:

- To improve QuickCheck CI by integrating the FindExamples tool into the web application.
- To improve the FindExamples tool itself, by investigating and developing the heuristic used to find the examples, both by modifying the original heuristic and trying new ideas. This also includes evaluating the improved heuristic.

[2]`http://www.quickcheck-ci.com`

1.3 Limitations

Given the limited amount of time available, some restrictions had to be made on what was expected to be achieved.

- The focus for the integration work is on functionality and not aesthetics, and to keep the changes as small as possible, preferring to reuse existing infrastructure rather than rebuilding the system.
- For the development of heuristics, the focus is on exploration and prototyping, rather than producing a finished tool, and so the evaluation process is only indicative of performance. The performance in terms of running time or memory usage is not a priority in this work.

1.4 Related work

A lot of research is done on testing techniques, as is to be expected given the amount of time and money spent on testing in industry. As the time spent on testing can be seen as an investment that lowers the cost of maintenance and support later on in the project, the effort spent testing must be weighed against the possible profit of releasing the product earlier.

For the concept of generating good examples to aid in the understanding of a system, no previous work has been found apart from FindExamples itself, however there is a lot of work related to automated and random testing that is of interest. Particularly interesting is work related to generating tests for some special purpose, and there are several articles written on the subject of generating test cases with varying degrees of randomness involved [9]–[12].

1.4.1 Software testing in general

There are a few different high-level approaches to verification of software, ranging from unit testing to provable correctness. With unit testing, testing is done on the lowest possible level, where test-cases are run on the smallest components – methods or functions. This approach favours modular design, where units are decoupled from each other as much as possible. The use of unit tests is taken to the extreme in a development methodology called Extreme Programming [13], where test-driven development is used. There, a developer must write tests before the real code.

On the other side of the spectrum, there are provably correct programs where the correctness is mathematically proven. Even though this is the most safe way of assuring correctness, it is also the hardest and most resource demanding. Furthermore, many programs cannot be proven correct due to non-deterministic behaviour and the fact that the general problem of automatically proving program correctness is undecidable, a well known example of this being the halting problem [14]. In addition, proofs of program correctness require a specification of what is to be proven, which pushes the issue of bugs from the code to the specification.

There are also several other approaches in between. For this thesis, the concept of clarifying complex properties and specifications is of interest. There is another extension to QuickCheck, which was developed during Prowess that determines if unit tests are covered by properties, connecting the simple test cases to the abstract properties [15]. Another interesting idea is the one found in the Ruby tool Cucumber [16], where tests are specified in natural language, and then translated into code – allowing the tests to work as the specification – while still keeping it readable to anyone, regardless of programming proficiency.

1.4.2 Property-based and random testing

Apart from QuickCheck, there are several other tools that use property-based and random testing. Both property-based testing [17] and random testing [18] have been used for many years. This section contains a few tools that employ interesting techniques related to these testing methods.

Randoop

Sharing some ideas with MoreBugs and FindExamples, Randoop is a feedback-oriented testing tool for Java and .NET [19]. Much like QuickCheck it generates sequences of method calls to test stateful code, but unlike QuickCheck, Randoop does not wait until having generated the entire sequence before executing methods. Instead, this is done as they are generated (the reason QuickCheck does not do this is to facilitate shrinking). The result of executing the methods is used as feedback. If a call is illegal, the sequence is terminated, and if an error is found the sequence is used as a counterexample. It is only for successful method calls that the sequence is kept and further built upon.

The tool uses some interesting techniques, such as combining successful sequences to construct new ones, and the output of the tool is a ready-to-run test suite – just like in QuickCheck. This suite contains both a set of failing test sequences, useful when looking for bugs, and a set of well-behaved tests that can be used for regression testing.

Randoop has a set of standard contracts that it checks, corresponding to properties in QuickCheck. These include common object-oriented paradigm laws, such as the reflexivity of an objects `equals()` method, as well as the fact that the `hashcode()` and `toString()` methods should not throw exceptions. Furthermore, uncaught exceptions in general are considered faults, and are only filtered manually if they are documented properly. In this aspect, QuickCheck provides more power for the user to decide on what is to be tested by allowing them to create the properties.

DART

DART (Directed Automated Random Testing) is a tool used to automatically check C-code for errors [20]. It uses a combination of static and dynamic analysis of the code to detect crashes and failed assertions. One of the most important advantages is that the tool can be used without any extra code. This is accomplished by statically extracting the interface of the program under test, and then executing it several times with different inputs to exercise different paths in the code. The way these paths are chosen is a combination of random concrete, and directed symbolic testing.

First, a completely random input is generated and executed, where at each branching point, a new set of inputs is constructed to be run after the current run. This set of inputs is chosen to make the execution follow a different path than the one that was currently taken. To do this, DART uses symbolic constraints, that when solved, result in a set of inputs capable of reaching new paths in the code. After having constructed a complete set of inputs, the process is repeated until all possible paths have been tested.

BLAST

An interesting tool that checks user-specified predicates in C-code is BLAST [21]. It has been extended with the ability to generate test suites containing both positive examples proving that the predicate holds in certain locations of the code, as well as counterexamples of where it does not [22]. The predicates can be specified by the user, and thus offer a powerful way of guaranteeing properties. One way this can be used is to check the use of elevated privileges in code, making sure dangerous calls such as `exec` and `system` are never called while the program has root access.

Another useful application of BLAST is finding dead code, that is, code which is never executed. Such code blocks can be interesting as unreachable code is often due to an error made by the programmer. Finding these code blocks is done simply by setting the predicate to always be true, thus any code that never fulfills the predicate was never executed and is hence considered dead.

2

Theory

This thesis deals mainly with the Erlang version of QuickCheck [2] and two prototype tools developed as extensions to the library [23], [24]. Since the purpose has been the integration and improvement of one of these tools, this chapter gives a broad introduction to the QuickCheck library in general, and the state of these tools in particular. Here, the tools are described as they were at the start of thesis.

The main use of QuickCheck has been the part of the library that deals with stateful testing, the `eqc_statem` and `eqc_fsm` modules [1]. The difference between the two is that `eqc_fsm` deals with *finite* state machines with a finite number of named states, whereas `eqc_statem` does not have this restriction. There are some advantages with using `eqc_fsm` when possible though, for instance that the named states can implicitly handle many preconditions, and that diagrams can be automatically generated from the specification.

Like regular QuickCheck, the testing is done by constructing properties, and in the case of stateful testing, these properties are much larger than when testing stateless code, as they also involve creating a model of the stateful program.

2.1 Stateless testing

In order to understand the stateful testing, the basic concepts of stateless QuickCheck must be understood. A QuickCheck property is essentially a boolean condition that is supposed to hold for all values of some kind (the values need not be of the same data type, as the dynamic typing of Erlang allows for functions with multiple input types). The way these values are chosen is specified by a generator, a function that randomly produces values of a certain kind, and QuickCheck makes sure the condition holds for each of them.

2.1.1 Generators

For most simple data types, there are already generators defined in the QuickCheck library (specifically the `eqc_gen` module), which can be combined to create gener-

[1]The documentation for all the QuickCheck modules is available at: `http://quviq.com/documentation/eqc/index.html`

Listing 2.1: A QuickCheck generator for lists of pairs of integers and strings.

```erlang
-import(eqc_gen, [list/1,int/0,char/0]).
sample_generator() ->
  list({int(), list(char())}).
```

ators for more complicated types. Listing 2.1 shows an example of a generator that builds on the standard generators using tuples and lists.

By using the basic generators as building blocks, the user can construct most Erlang data structures. However, sometimes the standard generators are not enough, and then the `eqc_gen` module also provides functions for creating generators from concrete values. Say for example that an application takes a special argument that can only be one of a few possible atoms. One must then generate an atom only from the set of valid atoms, which can be done by using the `eqc_gen:elements/1` function which takes a list of possible values and generates a randomly chosen element from the list.

2.1.2 Properties

Once the input has been generated, the actual test should also be run. The test is run by executing the property written by the user. A property can be seen as a special function that takes no arguments, but may use generators to generate data, and returns a boolean which indicates the result of the test. Listing 2.2 shows what a simple property used for testing an implementation of a sorting algorithm could look like.

Listing 2.2: QuickCheck property specifying that the user's `my_quicksort/1` function should behave like the standard Erlang `lists:sort/1` function.

```erlang
-include("eqc/include/eqc.hrl").
prop_sort() ->
  ?FORALL(List, sample_generator(),
    lists:sort(List) == my_quicksort(List)).
```

Here, `?FORALL` is a macro defined in the `eqc` module, that takes a variable, a generator and an expression returning a property (booleans are one type of property). It works basically like a regular $\forall x \in \mathcal{X} : P(x)$ universal quantification for a property $P(\cdot)$, but instead of trying all values, QuickCheck generates one hundred values from the generator, and for each it binds the value to the variable, evaluating the expression into a property. If any of these fail (return false), the test is considered a failure, and the generated value that caused the error is taken as a counterexample.

Apart from the `?FORALL` macro, there are several others that allow more control of the properties created, and properties can be extended by nesting macros and function calls to construct a resulting property with the desired behaviour. These functions often have type signatures similar to:

```
eqc:numtests(N::nat(), P::prop()) -> prop()
```

That is, they wrap the property with some extra functionality or change some internal setting used by QuickCheck when it is executed. In this case, the natural number `N` specifies how many tests that should be run (instead of the standard 100). Since the `eqc:prop()` type is opaque, its internal data structure is hidden, and thus wrapping is the only way to change the behaviour of a property after it has been constructed.

2.1.3 Shrinking

When a counterexample is found for a property it is not likely to be a minimal counterexample of the given bug. Since the input data to tests is generated at random, the actual one found to fail is often not very helpful to present to the user. One of the main selling points of QuickCheck is its ability to shrink failed test cases into minimal ones that still exhibit the fault. This is done by repeatedly trying to simplify the counterexample and seeing if it still causes a failure. For regular values, shrinking is done by trying “smaller” values until the test no longer fails, for instance shorter lists, smaller numbers and so on.

2.2 Stateful testing and state machines

In order to test stateful code with QuickCheck, a model of the intended functionality of the program is needed. This is implemented as a state machine, where each function in the API of the program under test is extended into a family of functions in the model. This model is written by the user, and serves as the template of correct behaviour when executing tests. The model contains its own state, often a simplified version of what is really happening in the program under test. For each of the available commands in the program, the model contains a transition function that modifies the model state in the expected manner. Optionally, the user may specify a pre- and a post-condition for each command. The precondition tells if the command is valid to generate given the current state, and the postcondition checks if the program behaved as expected when the command was executed. Listing 2.3 shows the transition function (`create_account_next/3`) along with the actual program call (`create_account/2`) and the pre- and postconditions.

Properties are still used for stateful testing, but their structure differs slightly from the stateless properties, and they often follow a similar pattern. An example of a typical property can be seen in Listing 2.4. This pattern can be summarized as follows. First, the commands are generated by the built-in command generator `eqc_statem:commands/1`. Second, some setup is performed to prepare for the test, after which the commands are executed. Last, cleanup is performed in order to allow for continued testing, and the result of the execution is checked. In Listing 2.4, the `eqc_statem` module is used, however, the overall procedure is similar for `eqc_fsm`.

Listing 2.3: The state machine specification for the `create_account` command in `bank_eqc.erl`.

```
%%%%% Create account
create_account_args(S) ->
  [account(S), name(S)].

create_account(AccountName, Name) ->
  bank:create_account(AccountName, Name).

create_account_next(S, _R, [AName, UName]) ->
  case create_account_ok(S, {AName, UName}) of
    true  -> S#state{accounts = [{AName, UName, 0} |
                                 S#state.accounts]};
    false -> S
  end.

create_account_pre(S) ->
  S#state.open.

create_account_post(S, [AName, UName], R) ->
  Account = {AName, UName},
  case create_account_ok(S, Account) of
    true -> R == Account;
    false -> R == false
  end.

create_account_ok(S, {AName, Name}) ->
  logged_in(Name, S) andalso
    lists:filter(fun({AN, UN, _B}) ->
                     AN == AName andalso UN == Name
                 end, S#state.accounts) == [].
```

In the example, the property `Res == ok` is instrumented by using `eqc:aggregate/2` in conjunction with `eqc_statem:command_names/1` to print a list of the frequency of which the different commands occurred. The result of running the generated commands is then what determines whether the test passes or not.

To generate the commands, the `eqc_statem:commands/1` function collects all the valid commands by looking at the functions exported by the module containing the state machine model. It also gathers the precondition functions this way, in order to only generate valid sequences of commands.

The `eqc_statem:run_commands/2` function takes the module that contains the state machine specification, and the generated list of commands as input. It needs the module name in order to automatically find the postcondition functions. For each command that is executed, the model state is updated and the post condition is checked. If it is false, execution is stopped, as a bug has been found, and the list of commands up until that point are a counterexample to the property under test.

Listing 2.4: An eqc_statem property that generates commands randomly from the module containing the state machine model, executes them, and checks the result.

```
prop_state() ->
  ?FORALL(Cmds, eqc_statem:commands(?MODULE),
    begin
      run_setup(),
      {_, _, Res} = eqc:statem:run_commands(?MODULE, Cmds),
      cleanup()
      eqc:aggregate(eqc_statem:command_names(Cmds),
                    Res == ok)
    end).
```

2.2.1 Shrinking command sequences

Just as in the case of stateless testing, QuickCheck is able to shrink the test cases used in stateful testing. Here, the test case is a sequence of commands to be executed, and thus, a minimal example should contain only the commands necessary to provoke the bug. QuickCheck does this both by shrinking the arguments of commands, and by repeatedly deleting commands from the sequence, provided the test case still fails. For example, if there is a fault in the function `d/1` on some specific input `V1`, an original test case:

```
a() -> ok
V1 = b()
c(V1) -> ok
d(V1) -> error
```

might be shrunk to the simpler sequence:

```
V1 = b()
d(V1) -> error
```

The notation used here works as follows. Executing a command (for instance `a()`) results in the value to the right of the arrow (`ok`). If however, the actual value is not interesting in itself (such as a pointer when testing C-code), “`V1 =`” is used, where a variable (`V1`) replaces the actual value and is used in the rest of the example. This helps clarifying where the same pointer is used, as a variable `V1` is easier to keep track of than a pointer value such as `18759424`.

In the shrunk example, it is clear that the error is either in the output produced by `b/0`, or in the way `d/1` handles the input `V1`. Therefore, there is no longer a need to consider `a/0` or `c/1`, and the process of debugging this error is simplified.

2.3 FindExamples

The idea of the FindExamples tool is to make the abstract QuickCheck properties easier to understand. For larger programs, especially those requiring setup before running, the required properties and state machine models can become quite complex and hard to understand. It also becomes increasingly difficult to know what has actually been tested, as samples of test cases are generally large and unstructured due to their randomness.

To simplify the understanding of difficult concepts, concrete examples are often used. This is true for properties and models as well, and by giving concrete examples of the system behaviour, the FindExamples tool can make the understanding of QuickCheck properties and models easier. The tool works on state machine model specifications, generating what is thought to be interesting examples of when the tests pass. A test case, that is, a sequence of commands run on the API of the program under test, may either pass or fail due to a number of reasons. These include the program crashing, or that the program behaviour deviates from the model.

Of course, simply generating random tests that pass and presenting them to the user is not likely to be helpful. The examples must somehow be filtered to obtain only the interesting ones, which let the user understand the behaviour of the system from just looking at the sequence of commands and their output.

This is where the shrinking capabilities of QuickCheck come in. By using a heuristic to determine whether or not a test case is interesting, and then shrinking it while it is still interesting, the examples that are generated are both interesting (from the heuristic's point of view) and minimal. Such examples are thus suitable to show to the user. To allow the generation of several different examples at once, the concept of features is used to avoid generating examples that are too similar.

2.3.1 Feature based testing

QuickCheck is not only able to generate and run individual tests, it is also able to generate suites of tests using the `eqc_suite` module. These can be used as static regression tests, where you can be sure the same things are tested each time the suite is run. The way QuickCheck generates such a suite is by using the concept of *features*. A feature is an attribute of a specific test case, and could be for instance a list of the names of the commands executed, or the length of the list of commands. By randomly generating tests, inspecting what features they exhibit, and then keeping track of the features already found, a suite that covers as many different features as possible (within a given time limit) is generated.

The features of a test are specified by the user when writing the property by wrapping it in a call to `eqc:features/2`. A feature can be any Erlang data structure, and each test case can have a list of any number of them. By assigning tests features in clever ways, one can generate test suites for many different purposes, and this approach is used to great effect in FindExamples.

A special and useful feature that `eqc_suite` provides built-in support for is code coverage, that is, information about what parts of the code are executed by the test case, and QuickCheck provides functions that let users create test suites that cover as much of the code as possible. There are two different levels of coverage that can be used, line coverage from the `cover` module, or the more fine grained expression coverage provided by the `eqc_cover` module, included in QuickCheck.

2.3.2 MoreBugs

Another technique used by FindExamples is the prototype tool MoreBugs, an extension to QuickCheck. The purpose of MoreBugs is to find several bugs in a single run, by keeping track of what bugs have been found so far. For instance, say there is a bug in a function `f` that can be provoked by calling it directly without any setup. If there is a bug in another function `g` that requires other commands to run before it can be executed, it is very likely that QuickCheck only finds the bug in `f`, as that will be generated much more often.

In order to keep track of previously found bugs, MoreBugs generalizes bugs into patterns, and makes sure no tests that match already found patterns are generated as testing continues. This is made possible by the syntactic generalization used – bugs are characterized by the commands and arguments that are used to provoke them, something that can be compared during test case generation. This allows for the avoidance of already found bugs altogether – there is no need to fully generate a random test only to discover it finds an old bug.

The tool is also able to detect when bugs subsume each other, that is, when a counterexample is a more general form of the same bug as another counterexample. For instance, if there is a bug in the function `f/2`, that is provoked only when it is called twice with the same first argument, one counterexample for it could be:

```
f(a, b) -> ok
f(a, b) -> {'EXIT', ...}
```

However, the following counterexample is more general, and arguably more clear, as it shows that it is in fact the first argument that matters:

```
f(a, b) -> ok
f(a, c) -> {'EXIT', ...}
```

For these two counterexamples, the second would subsume the first by being more general. In FindExamples, the parts of MoreBugs that perform the comparisons on bugs are used when abstractly comparing examples by their features, to avoid the rediscovery of similar examples.

2.3.3 The original heuristic: delete

To determine whether an example is interesting or not, the FindExamples tool uses a simple heuristic. If the deletion of a single command makes the test fail, it is considered an interesting example. The example is then presented as the original list of commands that pass the test, and which command to delete in order for the test to fail. The failing and deleted command also make up the feature of the test case, used by `eqc_suite`, where the commands are abstracted using code from

MoreBugs to avoid finding multiple copies of the same example.

In this context, failing does not mean that the commands remaining after deletion are executed and checked for failure. Instead, the hypothesis that the deleted command does not affect the behaviour of the commands following it is tested until it fails, in which case a good example has been found. The testing is done by only running the model, as the results of running the commands are already available, so there is no need to re-execute them. If the pre- or postcondition for any command fails during this testing, the deleted command must have affected it in some way and the example is thus considered good.

To illustrate this, consider an example using the Erlang registry. Registering the same process twice is not allowed, and thus, the program should return some negative result, a caught exception in this case. By using the delete-heuristic, FindExamples may come up with the following good example:

```
  1. V1 = reg_eqc:spawn()
  2. reg_eqc:register(a, V1) -> true
  3. reg_eqc:register(b, V1) -> {'EXIT', {badarg, ...}}
Deleting command 2 changes the behaviour of command 3:
  1. V1 = reg_eqc:spawn()
  3. reg_eqc:register(a, V1) -> should not fail
```

Running the resulting shorter sequence of commands, one would expect to get:

```
1. V1 = reg_eqc:spawn()
3. reg_eqc:register(b, V1) -> true
```

This is a valid use of the registry, and should pass if executed as a test. However, as FindExamples checks the postconditions using the values from the original sequence of commands (`{'EXIT', {badarg, ...}}` for command 3 in this case), the postcondition of command 3 will fail, as it gets `true` instead.

Thus, the heuristic has found a good example, and one could argue that this actually is a good example; it illustrates the fact that an Erlang process is only allowed to be registered to one name. In general, this heuristic is good at finding examples that showcase the interplay between two commands, where the deletion of one modifies the behaviour of the other.

2.3.4 Drawbacks with the delete-heuristic

There is however still room for improvement in the prototype tool. One weakness that was found during the development of the original tool was its inability to find interesting behaviour that is not clearly visible after only one command deletion. A concrete example of this is when generating examples for a model of the `dets` module. `dets` provides permanent file storage and works like a key-value store. In order to use a `dets` table, it must first be opened, and after using it, it should be closed, much like operating on a regular file. One very important property of a `dets` table is whether or not the contents are kept after closing it. Optimally, this should be illustrated by the examples generated by FindExamples, by showing an example where data is accessed both before and after closing the table.

This is not the case however, as the tool seems unable to generate such an example. The reason for this is that deleting only one command will provoke a simpler good example, as is illustrated by the following example:

```
1.    open_file(dets_table, [{type, bag}]) -> dets_table
2.    insert_new(dets_table, {2, 0}) -> true
3.    close(dets_table) -> ok
4.    open_file(dets_table, [{type, bag}]) -> dets_table
5.    lookup(dets_table, 2) -> [{2, 0}]
```

This sequence shows the fact that closing and opening the `dets` table keeps its contents. Now, if the delete-heuristic were to delete the `close` command, the second `open` would fail (opening twice is not allowed), and a change in behaviour would be found. However, to illustrate that behaviour, the other commands are not necessary, and the original example would be shrunk to just:

```
1.    open_file(dets_table, [{type, bag}]) -> dets_table
2.    close(dets_table) -> ok
3.    open_file(dets_table, [{type, bag}]) -> dets_table
```

which does not show the fact the contents are kept. This issue is similar for many instances of examples where the behaviour is unaffected by the deletion of a sequence of commands.

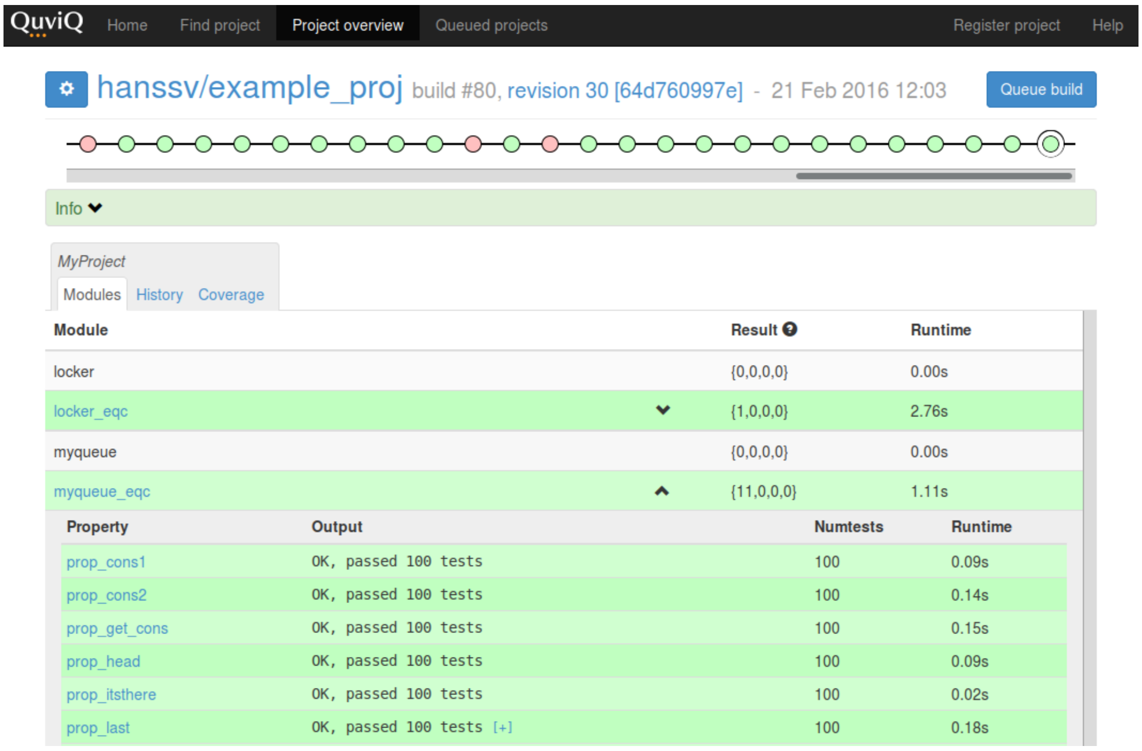
2.3.5 Parameterized modules

A special technique sometimes used in Erlang is *parameterized modules* [25]. This language feature was only introduced as an experimental part of Erlang, and was later dropped, but it is still possible to use it through a *parse transform*. A parse transform can change or add functionality in a module by modifying its syntax tree during compilation, allowing for some patterns to be abstracted away and hidden from the programmer. One of these patterns is the parameterization of modules, provided by the `pmod_pt` transform, allowing for extra arguments to be passed to a module, making them implicit arguments to all functions in the module.

The reason for parameterized modules being used both in QuickCheck and this thesis, is that they allow writing modules that "wrap around" other modules, to extend their functionality. For example, one could imagine a performance critical algorithm that requires memoization to be efficient. Then, one could implement the algorithm without memoization in a module `A`, not involving the complexity of the state associated with memoization, and a separate module `M` that can perform memoization on any module using a certain interface. By using the module `M:new(A)` to call the functions of `A`, the memoization will be used without any extra effort. The concept is somewhat similar to inheritance in object oriented languages. In QuickCheck, `eqc_fsm`s are wrapped in a parameterized callback module that has the same external interface as an `eqc_statem` module, using the internal `eqc_fsm` interface to compute the results. Thus, the code used to handle both kinds of state machines can be the same, avoiding unnecessary code duplication.

2.4 QuickCheck CI

To simplify development of open source projects, the web service QuickCheck CI provides full access to the Erlang QuickCheck library through its build server. Users can connect their GitHub repositories and have the QuickCheck CI server run QuickCheck on any properties in the code. The properties are grouped by modules, and each property may also hold a set of counterexamples. Figure 2.1 shows the view of an example project with several properties. A history of the previously performed builds, and the results of the testing for them is also available.



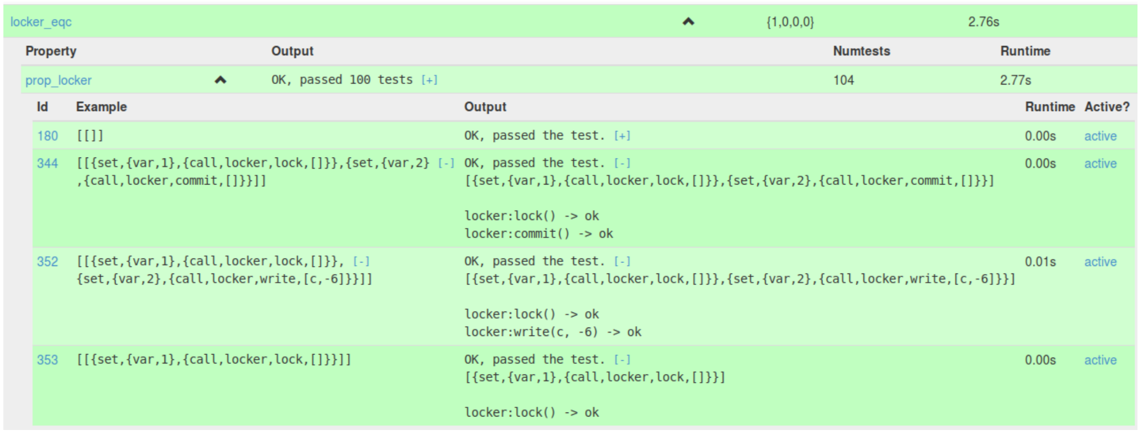
hanssv/example_proj build #80, revision 30 [64d760997e] - 21 Feb 2016 12:03

Module		Result	Runtime
locker		{0,0,0,0}	0.00s
locker_eqc	⌄	{1,0,0,0}	2.76s
myqueue		{0,0,0,0}	0.00s
myqueue_eqc	⌃	{11,0,0,0}	1.11s

Property	Output	Numtests	Runtime
prop_cons1	`OK, passed 100 tests`	100	0.09s
prop_cons2	`OK, passed 100 tests`	100	0.14s
prop_get_cons	`OK, passed 100 tests`	100	0.15s
prop_head	`OK, passed 100 tests`	100	0.09s
prop_itsthere	`OK, passed 100 tests`	100	0.02s
prop_last	`OK, passed 100 tests [+]`	100	0.18s

Figure 2.1: The project view of the QuickCheck CI web page, showing that the two modules `locker_eqc` and `myqueue_eqc` contain QuickCheck properties. The `myqueue_eqc` module is expanded to show the properties that have been tested from it. At the top, the red and green circles represent previously failed and passed builds, respectively.

Each time a property fails when tested, the counterexample that falsified the property is saved and then retested each coming build. The examples thus work as a form of regression test suite, preventing old bugs from reappearing unnoticed. The counterexamples are displayed as can be seen in Figure 2.2, along with a button to deactivate each example, "active". By deactivating an example, it is no longer run for coming builds, which can be useful for examples that are no longer relevant. Reasons for this can be a change in the specification or the API of the program.



locker_eqc {1,0,0,0} 2.76s

Property	Output	Numtests	Runtime
prop_locker	OK, passed 100 tests [+]	104	2.77s

Id	Example	Output	Runtime	Active?
180	`[[]]`	OK, passed the test. [+]	0.00s	active

344	`[[{set,{var,1},{call,locker,lock,[]}},{set,{var,2} ,{call,locker,commit,[]}}]]` [-]	OK, passed the test. [-]
		`[{set,{var,1},{call,locker,lock,[]}},{set,{var,2},{call,locker,commit,[]}}]`
		
		`locker:lock() -> ok`
		`locker:commit() -> ok`
			0.00s	active

352	`[[{set,{var,1},{call,locker,lock,[]}}, {set,{var,2},{call,locker,write,[c,-6]}}]]` [-]	OK, passed the test. [-]
		`[{set,{var,1},{call,locker,lock,[]}},{set,{var,2},{call,locker,write,[c,-6]}}]`
		
		`locker:lock() -> ok`
		`locker:write(c, -6) -> ok`
			0.01s	active

353	`[[{set,{var,1},{call,locker,lock,[]}}]]`	OK, passed the test. [-]
		`[{set,{var,1},{call,locker,lock,[]}}]`
		
		`locker:lock() -> ok`
			0.00s	active

Figure 2.2: An expanded view of the property `prop_locker` that has a few old examples saved. These are old counterexamples that have failed at some time during the development of the program, and are now used as regression tests. The "Output" column displays the test in a human readable form, whereas the actual data structure of the example is shown to the left, useful for debugging.

3
Methods and implementation

The work done in this thesis can be divided into two main parts: the integration of the FindExamples tool into the QuickCheck CI web service, and the study and development of new heuristics for finding good examples. Due to the unfamiliarity with both Erlang QuickCheck in general, and FindExamples and QuickCheck CI in particular at the start of the project, the integration part of the work doubled as an introduction to the systems.

3.1 Study of the systems

The first goal was to become familiar with the QuickCheck state machines used to test stateful code. Bundled with the FindExamples code provided by Quviq were several example modules used to test the tool, and these served as a template for developing a new example from scratch, a simple model of a bank. The choice of example was made based on the desire to have several actions depending on the state of the program. This is certainly true for a bank, where the state must be kept correct to avoid angry customers.

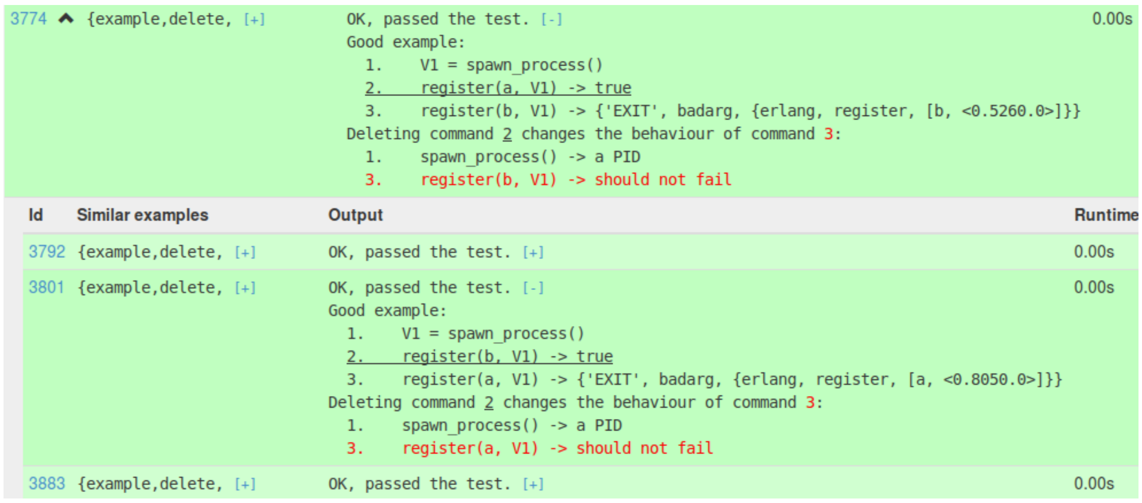
Keeping the example quite simple, the bank module performs only basic operations, such as creating users, accounts, and depositing or withdrawing money. In order to introduce more state, some operations require the user to be logged in, and all of them also require the bank to be open. Of course, the user and account must also exist for a transaction to be successful. A typical example of the use of the bank API is the following sequence of commands, where a user deposits some money. In the banking examples to follow, `u1, u2, ...` are users, and `p` and `a` stand for password and account respectively:

```
1.    open() -> ok
2.    create_user(u1, p3) -> {u1, p3}
3.    login(u1, p3) -> ok
4.    create_account(a2, u1) -> {a2, u1}
5.    deposit(u1, p3, a2, savings, 10) -> 10
```

In parallel with the development of the bank example, a QuickCheck specification was developed, using the callback-behaviour defined in `eqc_statem`. This was a helpful exercise in using the "new" parts of QuickCheck. Furthermore, the model was used to continuously test the implementation during development. The most important parts of the source code for this example can be found in Appendix A.

3.2 Integration

Due to the fact that QuickCheck CI already handled examples before the introduction of the FindExamples tool, the integration did not require modifications to the foundation of the web server, and also, the database could be kept intact. When a regular property fails, the counterexample is stored in the database as an example that will be run for each successive build of the project. The example is also shown within the property in the web interface, along with the result of running it and the corresponding output. Thus, the examples found by the tool did not have to be modified to be stored in the database, however, the handling of examples had to change significantly in order for them to display correctly – and for the new examples from FindExamples – to even be generated in the first place. There were a few other issues to solve, one of them being the way examples are grouped.



```
3774 ^ {example,delete, [+]        OK, passed the test. [-]                                                        0.00s
                                   Good example:
                                     1.    V1 = spawn_process()
                                     2.    register(a, V1) -> true
                                     3.    register(b, V1) -> {'EXIT', badarg, {erlang, register, [b, <0.5260.0>]}}
                                   Deleting command 2 changes the behaviour of command 3:
                                     1.    spawn_process() -> a PID
                                     3.    register(b, V1) -> should not fail
```

Id	Similar examples	Output	Runtime
3792	{example,delete, [+]	OK, passed the test. [+]	0.00s
3801	{example,delete, [+]	OK, passed the test. [-] Good example: 1. V1 = spawn_process() 2. register(b, V1) -> true 3. register(a, V1) -> {'EXIT', badarg, {erlang, register, [a, <0.8050.0>]}} Deleting command 2 changes the behaviour of command 3: 1. spawn_process() -> a PID 3. register(a, V1) -> should not fail	0.00s
3883	{example,delete, [+]	OK, passed the test. [+]	0.00s

Figure 3.1: Similar examples from the registry model grouped together in QuickCheck CI, differing only in the names used for registering, `a` and `b`.

Grouping of examples is done to avoid several examples showing the same bug or behaviour from being displayed, by comparing the examples abstractly to ignore insignificant differences such as differing concrete values of arguments. For instance, if a process identifier (pid) used in two otherwise equal examples of the registry model differ, the examples should still be grouped together. Figure 3.1 shows how this is displayed in the web interface. Since the new type of examples contain more data than just the commands that are run (for instance the index of the failing command), they might not compare equal when they should be considered equivalent. This issue was solved by comparing such examples only by their abstracted feature and not the entire data structure.

3.2.1 Modifications to the API of FindExamples

The original version of the FindExamples tool was an early prototype, and as such, there were plenty of things that needed to be cleaned up and simplified before using it further. The first example of this was the way the tool was used. In order to generate examples, one would have to write a special property that would not be possible to test with regular QuickCheck. Also, an internal and undocumented function had to be used within this special property. The way this was remedied was by changing the API of the tool completely, introducing a new function `find_examples:generate_examples/5` that acts as a wrapper around a regular property, allowing it to still be used with regular QuickCheck without modification. See Listing 3.1 for an example of how this function is used to construct a property.

Listing 3.1: A property generating examples for a model of the Erlang registry.

```
-include_lib("eqc/include/eqc.hrl").
-include_lib("eqc/include/eqc_statem.hrl").
prop_registry() ->
  ?FORALL(Cmds,commands(?MODULE),
          begin
            % Setting up for a new test
            [catch erlang:unregister(N) || N <- ?NAMES],
            {Hist, _State, Res} = run_commands(?MODULE, Cmds),
            find_examples:generate_examples(?MODULE, Cmds, Hist,
                                            Res, Res == ok))
          end).
```

The `find_examples:generate_examples/5` function takes as arguments the module and regular QuickCheck property in question, as well as the commands run, the history (containing the output from running them), and the actual result of the test. Given these, it checks whether or not the list of commands can be used to find a good example. If the property is being run by regular `eqc:quickcheck/1`, the wrapper will behave exactly as the inner property. It will only generate the examples when run by the special `find_examples:find_examples/1` function, that takes a property as its only argument. It runs the example generation, returning the resulting list of examples.

Choosing how the property behaves depending on what context it is being called from was implemented in a slightly tricky way. The `generate_examples/5` function must realize that it should generate examples, and thus `find_examples/1` must somehow notify the property of this. To do this, the fact that QuickCheck properties are actually generators is used, along with the `eqc_gen:with_parameter/3` function, that passes an arbitrary argument to the property. This parameter is then read from within `generate_examples/5`, and the appropriate action is taken based on its value.

Another important aspect when integrating the FindExamples tool into QuickCheck

CI was the fact that the web server must be able to tell what properties are capable of generating examples, that is, contain a call to `generate_examples/5`. Since a property is basically a generator for a zero-argument function, it is not possible to see what it contains from the outside; the `eqc:prop()` type is opaque. One way of solving this would have been to use global state, for instance in the form of the Erlang registry, and register a process from within the property if it is able to generate examples. However, a cleaner approach was to instead make the programmer annotate their properties with a module attribute `-generate_examples(Prop_name)`, which can be read by the web server before deciding to generate examples or not. This saves it from running the example generation unnecessarily, as though regular properties will not make `find_examples/1` fail, running them is a waste of time.

3.2.2 Finding bugs

During development, with planted bugs in the test-programs, it became clear that some bugs that QuickCheck was very unlikely to find in just 100 test runs were actually more likely to appear when running the feature-based generation of examples. One very clear example of this was in the banking example, where the following bug was planted: The "+"-sign in deposit was replaced with a "-"-sign. See Appendix A.1 for the source code of the banking program without the bug. This is a very serious bug, but hard to find in a relatively small amount of random tests, due to the depth at which it occurs; after a user is created, logged in and an account is setup; the user must be successfully used to first deposit and then try to withdraw some money. To help the user find such bugs, should FindExamples stumble upon them, the tool was extended to not only return the good examples found by the heuristic, but also a list of bugs found. This way, the effort of finding such bugs is not wasted. In QuickCheck CI, these bugs are handled just like counterexamples for failing properties.

3.2.3 Pretty printing

When displaying examples in the terminal, a textual representation is used, generated by the FindExamples tool. The way this was done prior to the thesis was mainly meant for debugging, and was not easily comprehensible for first time users. To simplify the output generated, but also make future extensions to the printing capabilities possible, the printing part of the tool was largely rewritten. Internally, the printing code uses the standard Erlang library `prettypr`. The library is based on a similar Haskell library [26], and provides combinators for constructing nicely formatted documents. The `prettypr` module provides a function for converting such a document into plaintext that can be printed directly to the terminal. The prototype tool contained code for generating such a document and then converting it to plaintext, but also for generating LaTeX-code directly, without using pretty printing.

Web

Instead of adding a third heap of printing code to specialize the printing for web display, the existing code was modified to use an internal data type to encode the data needed to print examples and commands, somewhat emulating the data structure of the examples themselves, but also containing the contextual information needed to distinguish different scenarios when printing. For example, if a piece of text needs to be printed in a special way, or in some way be marked, that part of the data structure is tagged with a special value for that kind of marking, like deletion or failure. To do this, the data structure had to be built from the ground up, with new types for print-examples, commands, and calls. This meant that `prettypr` library could not be used directly as it is, but instead the resulting data structure could later be translated to a `prettypr` document containing either plaintext for terminal output or HTML code for web output.

Latex

Apart from the plaintext and HTML formats that use `prettypr`, support for generating LaTeX-code from the internal data type was also implemented. Since the contextual information is included in the data structure, the amount of code needed for adding a new printing format was reduced. With working LaTeX-generation, printable documents could easily be produced automatically from sets of examples.

Compact

As the development of heuristics went on in the second part of the work, where different heuristics required different printing, the printing code needed to be extended to handle the new ones as well. Furthermore, a way to compactly present a collection of examples from different heuristics was needed. For instance, an example from the original delete-heuristic consists of two command sequences, one with the deleted command still left and one without. Such an example could actually be seen as two examples consisting just of their respective list of commands, meaning they have nothing to do with the heuristic they were found by, and thus examples of different kinds can be printed in a unified format alongside each other. Another advantage of this approach is that some examples, from possibly different heuristics, can be merged, as they may use the exact same sequence of commands. This was very useful for the evaluation, where examples from different heuristics needed to be presented in a concise form on a piece of paper to people that had little experience with QuickCheck (details on this evaluation can be found in Section 3.4.1).

3.3 Improvement of heuristics

As the integration was becoming more and more finalized, the work started shifting towards the spawning of ideas for new heuristics and possible improvements to the old one. This section explains the work carried out consisting of studying available literature and experimenting with ideas for new and improved heuristics.

3.3.1 Literature study

Despite the lack of literature relating directly to the problem of generating representative examples from properties, there is much interesting literature regarding the generation of tests in general. To get some inspiration and ideas of how the heuristics could be improved, and also to learn about what had been done previously, this area of literature was studied. Some of the most relevant literature is cited in Section 1.4.

3.3.2 Ideas for heuristics

The integration work with the original FindExamples tool was not only good in that it improved the functionality of QuickCheck CI, it was also a good introduction to QuickCheck in general, and the concept of generating examples in particular. During this work, a few ideas were forming, and a lot was learned about the more advanced uses of the QuickCheck library. Quite early on, a discussion with John Hughes led to some good starting points in the search for new heuristics. Two ideas that arose during this discussion were the use of coverage based generation, and the idea of reusing some of the ideas behind Randoop when generating commands [27]. During the implementation of these ideas, a third potential idea formed – a slight modification of the original delete-heuristic into a swap-heuristic.

Later on in the project, ideas for not only inventing new heuristics, but also tweak the original ones were forming. These include the deletion of several commands and the use of contextual information to determine whether test cases are interesting.

3.3.3 New heuristic: coverage

Often when developing software, the quality of test suites is measured in the percentage of coverage they obtain, that is, how large a part of the executable expressions are actually executed during a run of the test suite [28]. Different coverage tools have different granularities when it comes to the level of detail they can track, for example, the Erlang `cover` module tracks only line coverage, whereas the `eqc_cover` module is able to track individual expressions, and is thus able to provide better statistics. Furthermore, the `eqc_suite` module contains functions for generating test suites that cover as much of a module as possible.

It was thus natural to use coverage as a first attempt of an alternative heuristic, given that there already exist functions handling coverage in the QuickCheck library, as well as the fact that it is such a simple and quantifiable metric to measure the supposed quality of tests. Since coverage tools were already available, the task of implementing this heuristic was not too hard, and a working implementation was easily produced and tested on a few of the example programs available.

3.3.4 Modified command generation

In an attempt to improve the performance of the coverage based generation, a Randoop-styled command generation technique was also implemented. This generator did eventually make its way to other heuristics too, as it was implemented in a very general way, and could in principle be used by most other conceivable heuristics. However, due to an issue that was found late in its development (explained in Section 4.2.2) this generator was not used in the version that was eventually evaluated.

The idea is to try and start from previously successful command sequences when generating new ones, to reuse some of the good sequences found earlier. One example of this would be the following. Suppose you wanted to generate a valid `withdraw` command in the bank example, but there was already a sequence with a valid deposit command:

```
1. open() -> ok
2. create_user(u1, p1) -> {u1, p1}
3. login(u1, p1) -> ok
4. create_account(a1, u1) -> {a1, u1}
5. deposit(u1, p1, a1, 42) -> 42
```

Simply adding the command `withdraw(u1,p1,a1,5)` to this sequence of commands would showcase a valid withdraw, which is much more likely to generate than generating the entire sequence of the 6 commands from scratch randomly.

To do this, the generator that generates the commands run by `run_commands/2` had to be modified, to not only generate the commands completely randomly, but also maintain a set of candidate sequences to build on top of. This being Erlang, the implementation was done as a server process, getting command sequences from the property when they pass, and sending them to the command generator when it produces the next sequence. To ensure some new commands are still generated, the generator uses a weighted probability when it chooses between generating new commands, and reusing and extending old ones. By keeping the server internal to the example generation module, this command generator is as easy to use for the user as the regular built in generator.

The first version of this generator suffered from a big problem, namely shrinking. With the server generator, QuickCheck was not able to shrink command sequences, and thus, the resulting examples were very long (up towards one hundred commands) and not very useful. To remedy this, a suggestion from John Hughes was to use a parameterized module to wrap around the state machine module, and maintain the command candidates there. This way, the command generation can be done one command at a time, as this allows QuickCheck's shrinking to work as usual.

Parameterized modules are explained in Section 2.3.5, and by using them, the command generator can be easily applied to any heuristic. The interface is simple, the command generator is replaced by a heuristic-specific generator that delegates the call to the command server through the parameterized module, and once a good example has been found, it is sent to the server from within the property that searches for examples. The server is able to store only the fully shrunk examples by caching the last received command sequence until a new test is begun, avoiding the unnecessary storing of excessively long examples.

3.3.5 New heuristic: swap

A natural modification to the original heuristic of deleting commands and observing any differing behaviour, is to instead modify the arguments to the commands and observe the resulting new behaviour. An example of this could be the following, where the fact that two users can be created in the system is illustrated:

```
1. open() -> ok
2. create_user(u1, p1) -> {u1, p1}
3. create_user(u2, p2) -> {u2, p2}
```

If the first argument of command 2 is changed to `u2`, command 3 should change behaviour to return `false` instead, showing that a user cannot be created twice. Both these scenarios are interesting, and indicate that the idea of changing arguments can find good examples.

To implement this heuristic, one has to be able to randomly generate new arguments. Using the QuickCheck generators available in the model, this is easily done, however, once again shrinking was affected by using this naïve approach. By generating the arguments separately from the commands, the result of trying out a set of commands may differ from run to run, since the arguments are generated non-deterministically. This showed up as poorly shrunk examples littered with unnecessary commands.

To avoid this issue, a suggestion from John Hughes was used; generating an alternative command directly. By doing this, both the commands and their arguments will shrink together, moving the randomness to a single point, allowing shrinking to work as it should. Much like the command server generator explained earlier, the generator had to be extended, and again, a parameterized module was used (since nested parameterized modules are allowed, it is actually possible to use the command server for the swap-heuristic too).

When generating a command, the parameterized module uses the underlying command generator in the state machine model, and generates a pair of valid commands given the current state of the model. The command name is the same, but the arguments may differ and thus be the source of a good example. Now, since QuickCheck executes the model to check preconditions during the command generation, the calls returned must be valid calls, they cannot simply be a tuple of calls as that cannot be executed. Still, both the calls must be available in the data structure that is passed on, as that is all that will be available to the heuristic. Instead, a small trick is used, where the `erlang:apply/3` function is used in the following data structure:

```
{call, erlang, apply, [erlang, element, [1, {Cmd1, Cmd2}]]}
```

When passed to QuickCheck, it will behave in exactly the same way as `Cmd1`, but both commands are still accessible to the property when finding the examples, and so the alternative can be tried out in place of the original, essentially a modification of the arguments to the command.

Apart from the command generation, the overall structure of the implementation of this heuristic was quite similar to that of the original delete-heuristic; it uses `eqc_suite:feature_based/1` and a property determining whether or not a test case is interesting, and many of the other ideas from the original heuristic. These include the avoidance of re-adding already found examples by checking an `ets`-table before adding them, and pruning duplicate examples when done. Also, the abstraction techniques from MoreBugs were used to abstract the features of examples when comparing them. The feature used in this heuristic consisted of the reason for failure, along with the abstract representation of three commands: the one that changed behaviour, and the original and modified version of the command that changed arguments.

3.3.6 Extensions of the delete-heuristic

One of the weaknesses of the original delete-heuristic was its inability to find examples showcasing the state-preserving behaviour of a program, for instance the fact that a `dets` table maintains its contents after closing and re-opening it. To try and find such examples, a modification to the original delete-heuristic was made, namely that the deletion is not limited to a single command, instead, a whole sequence of commands can be deleted, provided that the resulting program behaves in the same way as the old one. Thus, the close and open calls in `dets` would be candidates for deletion of this heuristic, as the behaviour should be the same before and after deleting them.

To get this working, one must be careful to choose when, and what, commands are allowed to be deleted, as well as choosing a suitable feature to not have an infinite stream of uninteresting examples unwillingly generated. The way this is done is to restrict the deleted commands to be in immediate succession, and that they start with a failing precondition (removing a call to close just before an open will make the precondition of open fail, but then deleting open will make the test ok again).

Another detail needed in the implementation is an idea of “negative” and “positive” commands. To avoid generating examples where the unaffected behaviour is not interesting in the first place, a filtering is made on the commands, determining if they were successful or not. Without this, a resulting example may look as follows, where no interesting behaviour is shown, regardless of the deletion of commands 2 and 3:

```
1. open() -> ok
2. close() -> ok
3. open() -> ok
4. withdraw(u1, p1, a1, 10) -> false
```

As the heuristic knows nothing about the program, the user must supply the information of what commands are negative, in the form of a callback function `negative/1`. For the banking example, this would mean a function that returns true only when given the value `false`, possibly also if given an exception. If the user has not implemented the `negative` function, all return values are assumed positive, allowing the heuristic to continue working, albeit with lower quality examples as a result. In many of the examples tried, it is worth the small effort of defining this function, as it is often the case that a program uses a few values to indicate failure, and by using a function for this purpose, the user can use all the power of Erlang, including pattern matching, to define the set of negative return values, for example, any type of exception or error, with a single line of code.

3.3.7 Combination of heuristics

In an effort to combine the strengths of two previous heuristics, the modified delete and swap, a new method was tried out and implemented. It uses a combination of deleting and modifying commands while trying to find interesting examples. Thanks to the two heuristics being rather similar, quite small modifications were needed to make them work together. The idea is that some behaviour is better illustrated by removing the command entirely rather than modifying the arguments, as that often results in the command becoming useless – the new arguments are invalid and thus don't change the state of the system – making it equivalent to deleting the command altogether. The reason for wanting to delete the command instead is that it is easier to read when the command is removed, and not just invalidated. Furthermore, given the inability of the swap-heuristic to modify zero-arity commands, the delete-heuristic was deemed a good complement.

When combining the two heuristics, the resulting examples will have features of differing types, making them hard to compare, meaning there might be similar examples from the two heuristics that show the same thing. To avoid redundancy, some way of combining results of different types was required; especially when wanting to present the examples in a concise and simple way. Because of this, a compact version of examples was introduced, more like a regular QuickCheck counterexample, namely a list of executable commands. The difference here is that also included are the results and an abstract representation of the commands. This way, the compact examples can be printed with their result easily, and similar examples can be pruned away.

Each delete or swap example is turned into two compact examples, the original list of commands and their recorded results are trivially turned into one, while the second requires some more work. From the example data structure, the alternative list of commands can be constructed, by either deleting or swapping the relevant command. The resulting list of commands is then executed to obtain the actual results (not only from the model) of running it, and the second compact example can be constructed. The compact examples can now be mixed from both heuristics, and any duplicates can be removed, further decreasing the space needed to display them.

3.4 Evaluation of heuristics

During early development, the main evaluation of how a heuristic performed was done by comparison with the existing delete-heuristic. For a new heuristic to even be considered for more thorough evaluation, it must at least perform on par with the existing delete-heuristic. At an early stage in continuous development, the comparison is made on the fly, as the examples generated by the delete-heuristic are fairly constant and easy to understand, so it is quickly evident if a new candidate is consistently missing some examples previously found. Conversely, given the many runs of the delete-heuristic during the web-integration, any new example found by a candidate heuristic would likely be noticed.

In order to get more formal results of the usefulness of the new heuristic, an experiment similar to one conducted at the end of the FindExamples prototype project was carried out.

3.4.1 Evaluation experiment

During the Prowess research project, an experiment was carried out to test the effectiveness of the original delete-heuristic. It was carried out by letting test subjects try to predict the outcome of a program given examples from the original FindExamples tool. This experiment was rather small in size, and was was focused on testing only the heuristic, with no control group.

To get stronger results, a larger experiment was carried out as part of this thesis, and a few modifications to the design of the experiment were made. As John Hughes was lecturing a course on parallel functional programming involving Erlang during the second half of the project, the students of the course were used as test subjects during the break of a lecture. Each student was given two sheets of paper, one containing representative examples of the banking program, and another containing command sequences from the banking program without their results, for them to fill in. Each such sequence makes up one task, and for each command, one point could be scored.

The goal was to show that the final combined heuristic was really a helpful tool for developers and to do so, it was compared with a reference set of examples. Thus, half of the students were given the good examples from the developed heuristic, whereas the other half were used as a reference group. Deciding which students were given what sets of examples was done completely randomly, and the papers had no connection to the students, leaving no way of identifying who had answered what. Since the experiment was carried out in a lecture room, there was no need to handle any personal information, and the anonymity of the participants was fully maintained.

There were a few candidates for the choice of the reference examples. The easiest way would be to generate random examples of the same size as those of the heuristic, but this would likely not be very helpful, and the fact that the heuristic performs better than random is not a very strong statement. Instead, the coverage based heuristic was used, as that is the simplest and most obvious way of measuring test quality, and it is an approach widely used in industry. Showing that the final heuristic outperforms a suite of tests with maximum coverage would be a stronger statement.

After getting the papers back from the participants, the results were collected and the total score of each student was calculated. The mean and standard deviation of the score of the two different groups were calculated. This was then used as the measurement of performance for the two methods.

3.4.2 Statistical significance

When performing an evaluation with randomly chosen test subjects, the results must be critically studied before drawing any conclusions from them [29]. In this case, the sample of test-subjects is representative of the intended audience for Erlang QuickCheck: functional programmers. The goal was to find a statistically significant indication of the usefulness of examples. Thus, the null hypothesis for the experiment was the following:

- The performance of the students with the good examples does not differ from the performance of the students with cover-examples

To test this hypothesis, a two sample t-test with differing variances (also called Welch's t-test) [30] was used to compare the means of the two groups of students. The reason for this choice is that although the underlying distribution of scores is unknown, the distribution of the mean scores is likely to be normal, along with the fact that the score is rather tightly bounded (0 is the absolute minimum, and the total number of commands in the tasks is a relatively small integer). As for the variance, it is also unknown, but since Welch's t-test works for differing variances, it is not an issue.

With this statistical test, the p-value threshold is set to 0.05. The p-value measures the probability of making a Type I error, that is, rejecting the null hypothesis when it is actually true. Thus, as long as the p-value is lower than the threshold, the null hypothesis can be rejected with reasonable confidence.

Let the cover-group be group 1, and the group given the good examples be group 2. Next, let $\bar{x}_1, \bar{x}_2$ denote the sample means of the two groups, and s_1, s_2 denote the sample standard deviation, as defined by:

$$s_i = \sqrt{\frac{1}{N_i - 1} \sum_{j=1}^{N_i} (x_{i,j} - \bar{x}_i)^2}$$

Where $x_{i,j}$ is the score of student j in group i.

The test statistic t can now be computed as follows:

$$t = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\frac{s_1^2}{N_1} + \frac{s_2^2}{N_2}}}$$

To be able to test the hypothesis and perform a two tailed test, the degree of freedom needs to be approximated using:

$$v \approx \frac{(\frac{s_1^2}{N_1} + \frac{s_2^2}{N_2})^2}{\frac{s_1^4}{N_1^2(N_1-1)} + \frac{s_2^4}{N_2^2(N_2-1)}}$$

Now, given t and v, the p-value can be computed using the t-distribution, either by a table or more exactly by using statistical software, for instance `scipy` for Python[1], to determine if the experiment showed a significant difference; that is, if $p < 0.05$. A two tailed test is used, as no assumption on the performance of the two methods can be made.

Another important statistical measure to consider is *effect size*, also mentioned in [30], which gives an indication of how much of a difference there are between the means of two populations [31]. The effect size is calculated by using the means and the pooled standard deviation:

$$s_{pooled} = \sqrt{\frac{(n_1 - 1) * s_1^2 + (n_2 - 1) * s_2^2}{n_1 + n_2 - 2}}$$

The effect size d or Cohen's d is then calculated as:

$$d = \frac{\bar{x}_2 - \bar{x}_1}{s_{pooled}}$$

According to general practice, there are three "guideline" levels of d-values, where $0.2, 0.5, 0.8$ correspond to *small, medium* and *large* effects, respectively [31].

3.4.3 Selection of examples

To conduct the experiment, two sets of examples had to be created. For the cover-heuristic, there was a theoretical maximum of examples that could be generated: once the set of examples obtains full coverage, there will be no more examples generated. The generation was thus set to run for long enough to obtain full coverage.

For the combined heuristic though, it was harder to select a representative set, as 1) the suite of examples generated each time differed slightly, 2) it is much harder, if not impossible, to know when all possible examples have been found. Thus, a few

[1]`http://www.scipy.org`

tests were carried out. Letting the generation run for a long time (several minutes) 50 times, and comparing the resulting sets of examples, making sure there were no significant differences.

3.4.4 Selection of tasks

In order to properly test the heuristic, a good set of tasks for the students had to be selected. The tasks had to be hard enough to not be trivial even without examples for help, but at the same time be possible to finish in around 15 minutes. Ideally, one would want the tasks to be generated randomly somehow, to avoid any human bias in them. However, after experimenting with generating tasks using QuickCheck's generators, it quickly became evident that doing so was in effect redoing the thesis all over again, as it would require a new sort of heuristic. This approach was thus discarded, and the manual method was used instead. The randomly generated tasks did serve as an inspiration-source for the constructed tasks though.

4 Results and discussion

In this chapter, the results found during the thesis are presented. First, the successful integration into QuickCheck CI is described, showing some examples of how the final version looks. Second, the findings from the work of developing and improving the heuristics are presented, along with the results of the experiment carried out to test the usefulness of the generated examples. Finally, some general observations made so far are also presented, related to the experiences of working with QuickCheck testing.

4.1 Integration

In order to not affect the running web server, the integration work was carried out on a local copy of the server, running the same code on a virtual machine. The work went by without any major difficulties, and the resulting changes did not require any deep changes in the structure of the server or database, as most of the new functionality could be added using the existing infrastructure.

4.1.1 Refactoring the FindExamples API

It was not only the web application that needed to be updated, the FindExamples tool also needed some modification to work properly in QuickCheck CI. The most challenging part was to come up with a way for the web server to derive the purpose of an opaque property. The final solution, making the user add a module attribute for each property, was the best compromise attainable. Even though it requires some extra work and code from the user, it saves the server from needlessly executing properties to find out whether or not they are able to generate examples, as that was one of the original ideas to solve the problem. A further advantage of the chosen approach is the simplicity of it, there is no extra code needed in the FindExamples tool, and the check added to the web server is very straightforward.

4.1.2 The web interface

By upgrading the printing code within the FindExamples tool, the addition of the new examples to the web interface was made much simpler. The web server uses

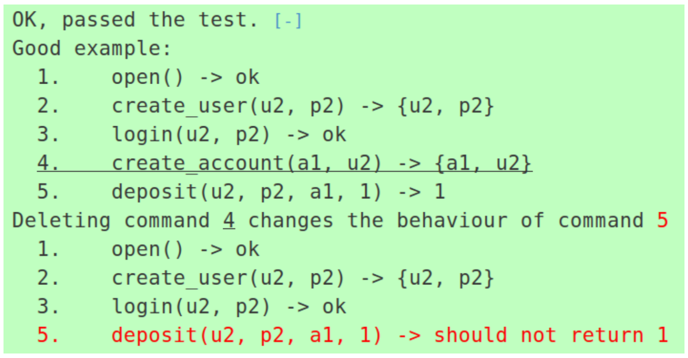

```
OK, passed the test. [-]
Good example:
  1.    open() -> ok
  2.    create_user(u2, p2) -> {u2, p2}
  3.    login(u2, p2) -> ok
  4.    create_account(a1, u2) -> {a1, u2}
  5.    deposit(u2, p2, a1, 1) -> 1
Deleting command 4 changes the behaviour of command 5
  1.    open() -> ok
  2.    create_user(u2, p2) -> {u2, p2}
  3.    login(u2, p2) -> ok
  5.    deposit(u2, p2, a1, 1) -> should not return 1
```

Figure 4.1: The output from one of the examples from the banking program. It shows the feature of the test by underlining and colouring the relevant commands. This particular example illustrates the fact that the account must be created before any money can be deposited to it.

the printing code from the tool to produce the output shown to the user. Figure 4.1 shows how a good example is displayed to the user, clearly showing why the sequence of commands is an interesting example.

The web page keeps track and re-tests all examples found for each new build the user makes, and one of the additions is the ability to recognize changed behaviour in examples. A change in behaviour can be that the bad part of an example suddenly passes, or that the feature of the test case changes from what it used to be. These events are reported as failures to the user, and are thus clearly visible. Figure 4.2 shows what a changed example looks like after fixing a bug.

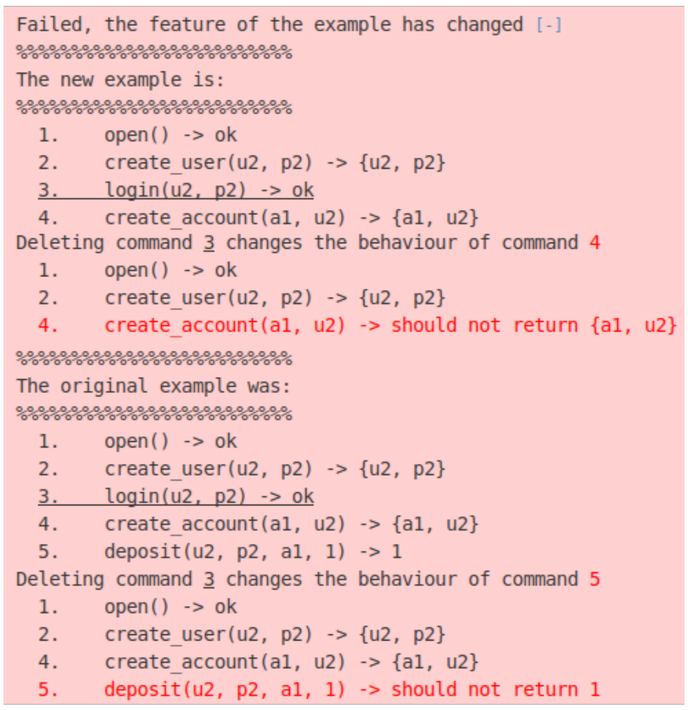
When the number of examples for a property grows, it becomes hard to keep track of new ones in the overview (see Figure 4.3). To clearly mark when new examples are found, the build in which they were first detected is tracked, and those examples are shown in blue.

4.1.3 Integration of new heuristics

As the project focus was shifted more towards the development of heuristics, new types of examples were created and required a few changes to the previous integration to work. In parallel with this development, the QuickCheck CI code was updated to work with all of the new heuristic techniques, with the exception of the cover-heuristic, as that uses the same coverage-tool used by QuickCheck CI for measuring coverage of tests, and nesting of coverage collection is not supported.

4.2 Improvement of heuristics

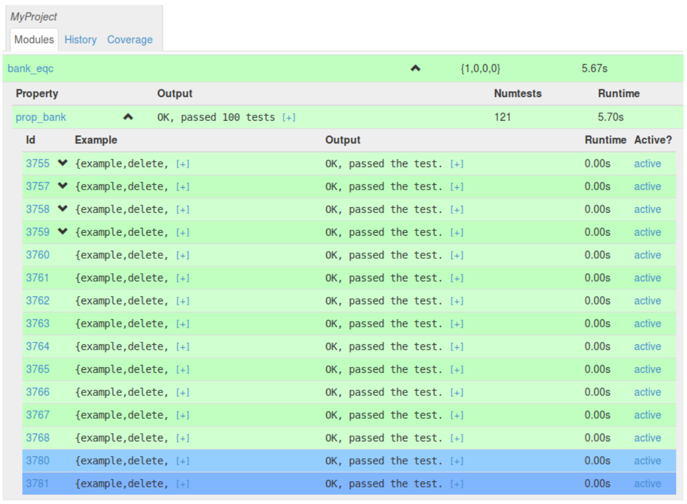
Throughout the work of developing improvements to the heuristic of FindExamples, many ideas were tested with varying amounts of success. The original delete-heuristic continuously served as the basic benchmark for all new ideas. For them



```
Failed, the feature of the example has changed [-]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
The new example is:
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  1.    open() -> ok
  2.    create_user(u2, p2) -> {u2, p2}
  3.    login(u2, p2) -> ok
  4.    create_account(a1, u2) -> {a1, u2}
Deleting command 3 changes the behaviour of command 4
  1.    open() -> ok
  2.    create_user(u2, p2) -> {u2, p2}
  4.    create_account(a1, u2) -> should not return {a1, u2}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
The original example was:
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  1.    open() -> ok
  2.    create_user(u2, p2) -> {u2, p2}
  3.    login(u2, p2) -> ok
  4.    create_account(a1, u2) -> {a1, u2}
  5.    deposit(u2, p2, a1, 1) -> 1
Deleting command 3 changes the behaviour of command 5
  1.    open() -> ok
  2.    create_user(u2, p2) -> {u2, p2}
  4.    create_account(a1, u2) -> {a1, u2}
  5.    deposit(u2, p2, a1, 1) -> should not return 1
```

Figure 4.2: An example that has changed its feature after a bug in the model was fixed from an earlier build. The reason for the bug was that the model did not check that the user was logged in when creating an account.



bank_eqc		{1,0,0,0}	5.67s

Property	Output	Numtests	Runtime
prop_bank	OK, passed 100 tests [+]	121	5.70s

Id	Example	Output	Runtime	Active?
3755	{example,delete, [+]	OK, passed the test. [+]	0.00s	active
3757	{example,delete, [+]	OK, passed the test. [+]	0.00s	active
3758	{example,delete, [+]	OK, passed the test. [+]	0.00s	active
3759	{example,delete, [+]	OK, passed the test. [+]	0.00s	active
3760	{example,delete, [+]	OK, passed the test. [+]	0.00s	active
3761	{example,delete, [+]	OK, passed the test. [+]	0.00s	active
3762	{example,delete, [+]	OK, passed the test. [+]	0.00s	active
3763	{example,delete, [+]	OK, passed the test. [+]	0.00s	active
3764	{example,delete, [+]	OK, passed the test. [+]	0.00s	active
3765	{example,delete, [+]	OK, passed the test. [+]	0.00s	active
3766	{example,delete, [+]	OK, passed the test. [+]	0.00s	active
3767	{example,delete, [+]	OK, passed the test. [+]	0.00s	active
3768	{example,delete, [+]	OK, passed the test. [+]	0.00s	active
3780	{example,delete, [+]	OK, passed the test. [+]	0.00s	active
3781	{example,delete, [+]	OK, passed the test. [+]	0.00s	active

Figure 4.3: An overview of the examples for a property in QuickCheck CI. Green means that the example is passing, whereas blue is a passing test that was found in the current build.

to be considered promising, they either had to perform at least on par with the original heuristic, or in some way produce new kinds of examples that were not previously found. This section presents the results found when trying out the heuristics introduced in the previous chapter, discussing some of their strengths and weaknesses.

4.2.1 The cover-heuristic

The simple idea of generating tests that together cover as much of the model code as possible sounds like a good idea; if all code of the model is covered – all the interesting behaviour must surely be covered? Sadly, this is not true, and beyond being slower than the original heuristic, the examples generated are not any better in the case of the current example programs, even in the runs where all testable code is covered.

One example that shows this is a case of the bank program, where a user successfully deposits money to the same account twice. This would be a good example to show, as the expected behaviour would be that the resulting balance is the sum of the two amounts deposited. However, the second call to deposit will execute the exact same code as the first (the code is available in Appendix A.1), and any example with two calls to `deposit` could be shrunk to one with a single call, as the coverage will be equal.

Another type of example that is not generated by the cover-heuristic, is an example where the bank is closed and reopened, after which a valid transaction is performed, showing that the bank keeps its state after closing (it would be very unfortunate otherwise). Such an example would be impossible for the cover-heuristic to find, as the state of the system is exactly the same before and after the close/open sequence, so any test case that contains such a sequence will be shrunk to one without it, still covering the same expressions.

These results are consistent with those found in several studies where the use of coverage as a target for test suites is dismissed [32]–[34]. In [33], the authors recommend the usage of coverage only as a measure of adequacy, that is, code coverage can be useful in pointing out parts of the code that are not sufficiently tested, but it cannot guarantee the quality of tests.

Coverage based test generation as a heuristic on its own is thus discarded, based on the fact that it is not able to find some of the important examples the original heuristic is able to find, despite it reaching its theoretical maximum and achieving an effective cover of 100%. However, a more advanced heuristic using coverage as a component, for instance while also using the model state, could still be a successful strategy, and would be an interesting approach to investigate further.

Despite not using coverage successfully as a way of finding examples, it still serves as a useful benchmark of what other heuristics can achieve, for instance in the evaluation experiment, as coverage is the standard approach to measuring test quality in industry.

4.2.2 Modified command generation

The idea of Randoop-style command generation originally seemed like a good idea, and even though it was successfully implemented and used along (almost) all of the heuristics tried, there were a few issues with it. These issues mostly related to shrinking, and were thus very hard to get to the bottom with. Debugging errors in shrinking is rather difficult, especially when the generator depends on an internal state. Even though most of the issues were cleared, there is some unexpected behaviour when shrinking commands generated for programs with re-used variables, in particular the registry example (also mentioned in Section 2.3.3), where pids are used multiple times as variables. Here, when shrinking, the index of the variable (specifying which value to use in a later command) sometimes gets offset, and thus may cause a faulty value to be used, resulting in a crash. For instance, a call to `register` may get a boolean value instead of a pid, which causes an exception.

Because of this, it was not deemed safe to use the command server extension for the heuristic being evaluated, but it may become useful in the future if the bug is fixed. Unfortunately, the underlying reason for it has not been found, although it has been determined that it is only provoked during shrinking, as there are no issues when the shrinking step is skipped altogether.

4.2.3 The swap-heuristic

For the swapping heuristic, the results are somewhat mixed but generally positive. While it is able to find some very good examples that the delete-heuristic is not able to find, one could argue that the examples produced are a bit more convoluted and hard to make sense of compared to the original delete-heuristic. One example of this is that some examples are quite similar to those of the delete-heuristic, but instead of deleting a command, the swap-heuristic simply invalidates it (not as in a failing pre-condition, but as in an unsuccessful command that passes but has no effect), making the resulting example harder to read than the corresponding delete example. This is illustrated in the following example:

```
  1. open() -> ok
  2. create_user(u1, p1) -> {u1, p1}
~ 3. login(u1, p1) -> ok
  4. create_account(a1, u1) -> {a1, p1}
Command 4 changes behaviour if command 3 is changed to:
  1. open() -> ok
  2. create_user(u1, p1) -> {u1, p1}
~ 3. login(u3, p1) -> ...
! 4. create_account(a1, u1) -> should not return {a1, u1}
```

Where the login in the second part will have no effect, and could just as well have been deleted.

On the other hand, the swap-heuristic introduces a family of new examples that the delete-heuristic does not find. This includes examples showing whether or not something can be done twice for different values. For instance, it is not possible to create two users with the same name, but two users may have the same password. By being able to modify a command down to a single argument, the swap-heuristic is able to better show how each argument affects the behaviour of a program. For instance, the sequence of commands:

```
1. open() -> ok
2. create_user(u1, p1) -> {u1, p1}
3. create_user(u2, p1) -> {u2, p1}
```

would not be generated by the delete-heuristic, as deletion will not change the behaviour, whereas changing the first argument of command 2 to `u2` does.

Another type of example that is found is where a command that sets up the system contains arguments that affect the behaviour of the rest of the program. An example of this is the following example from a model of a simple FIFO queue, where the argument to `new/1` determines the size of the queue:

```
~ 1.    V1 = q:new(2)
  2.    q:put(V1, 0) -> ok
  3.    q:put(V1, 0) -> ok
Command 3 changes behaviour if command 1 is changed to:
~ 1.    V1 = q:new(1)
  2.    q:put(V1, 0) -> ok
! 3.    q:put(V1, 0) -> call not allowed
```

This example illustrates how the size of the queue affects the amount of data that can be put into it, and would not be possible to show using the delete-heuristic.

There are however some disadvantages to this heuristic, one of them being the amount of examples it generates. This is not only an issue regarding running times, but also the fact that too many examples may overwhelm a human user. For the banking program, giving it enough time, the swap-heuristic can find several hundred examples, compared to around 20 for the delete-heuristic. Even after pruning away more general examples (using code from MoreBugs), there is an unreasonable amount of examples.

This among other things stops the swap-heuristic from being as useful on its own as the delete-heuristic. Apart from the sheer amount of examples it generates, there is also the obvious limitation of it not being able to modify zero-arity commands. For instance, every time the original heuristic is run on the banking example, a set of `open() -> ok, x -> false` examples are generated, where `x` is any command that has `open` as a precondition, and thus deleting it will make them fail. These are not possible to find with the swap-heuristic.

4.2.4 Extensions of the delete-heuristic

As described in Section 3.3.6, a few modifications were made to the original heuristic too. Overall, these proved to be quite successful, as the extended version was able to find a few examples that the original could not, mainly those showing how state is maintained. This is evident in both the `bank` and `dets` example programs, for instance the previously mentioned close-open example for `dets`:

```
  1.    open_file(dets_table, [{type, bag}]) -> dets_table
  2.    insert_new(dets_table, {2, 0}) -> true
  3.    close(dets_table) -> ok
  4.    open_file(dets_table, [{type, bag}]) -> dets_table
  5.    lookup(dets_table, 2) -> [{2, 0}]
Deleting the commands [3,4] does not change the behaviour of the program:
  1.    open_file(dets_table, [{type, bag}]) -> dets_table
  2.    insert_new(dets_table, {2, 0}) -> true
  5.    lookup(dets_table, 2) -> [{2, 0}]
```

This example shows that the table maintains its contents after closing and re-opening. The feature of this example consists of the first deleted command (`close`), and the first successful command afterwards (`lookup`). The concept of successful here comes from another extension, the idea of negative return values. By restricting the feature to only contain positive commands, the quality of the generated examples is increased. Without the restriction, examples on the form similar to the one above could be generated, but with an unsuccessful `lookup` instead, and thus not showing anything of interest – there is no indication of whether or not the table maintains its contents. Thus, the addition of negative return values is necessary for the multi-deletion not to produce too many examples, some of which will not be interesting. The usefulness of this extension thus heavily depends on whether or not the user has implemented the necessary `negative/1` callback function, as without it, such bad examples may be generated.

4.2.5 The final heuristic

After much internal testing of these heuristics, it seemed the best results (in view of the example programs described throughout this thesis) were obtained using the combined heuristic, that is, both the extended delete- and swap-heuristic. This heuristic seems to benefit from the strengths of both its components, and by merging the resulting examples as described in Section 3.3.7, the amount of examples is greatly reduced from the basic swap-heuristic.

The reason for this seems to be that the swap-heuristic generates many combinations of the same components. Say a sequence of commands `A` can be modified into either `B` or `C` by a single modification to a command. Then, the swap-heuristic could generate the following examples: `A->B,A->C,B->A,C->A,B->C,C->B`, where `->` indicates the change of an argument into another sequence. When compacted, this set of up to 6 examples could be compacted down to the three components `A,B,C`, greatly reducing the number of examples. Furthermore, since there may be many ways to modify each command (several arguments, multiple values), the total number of possible examples is even larger, and compacting them becomes important.

Another issue that is solved by using the compact representation of examples is the avoidance of swap-examples that could have been shown by deletion instead (as described in Section 4.2.3). This is done by simply ignoring the part of a swap-example that contains the negative version of the modified command, as that is likely to be covered by a corresponding delete example where the unsuccessful command is deleted completely instead.

By using the extended version of the delete-heuristic, the combined heuristic is able to find examples of how state is handled that the original heuristic would not be able to find; most importantly the state-preserving multi-deletions described earlier. The swap-heuristic adds many new examples that are not possible to get with the delete-heuristic, as described in Section 4.2.3.

Using the compact representation of examples, all the examples from the combined method can be compared, and since the sets of examples from the different heuristics may overlap in their components (the shorter version of a delete-example may be part of a swap-example for instance), duplicates can be filtered out. This also simplifies the printing of these examples, as there is less to print, and all are printed in the same simple way that looks more like regular QuickCheck counterexamples. To a user, the list of commands is interesting in itself, not the reason for why the heuristic considers them interesting.

4.3 Evaluation results

As the last part of the work, the experiment to test the performance of the chosen combined heuristic against the cover-heuristic was conducted. The cover-heuristic can be seen as the naïve baseline implementation for generating representative examples.

4.3.1 Selection of examples

For the cover-heuristic, it was enough to set the number of tests high enough (over $10,000$) to obtain full effective cover (some parts of the code would never be covered, for instance generators). The resulting test suite contained 22 examples.

For the combined heuristic, some more work was needed. First, the examples generated by the heuristic had to be deemed stable enough, by performing a test where 50 long runs were made. In total, 66 different examples were generated in the 50 runs, with the mean number of examples generated in each run being 53.2 (with a standard deviation of 1.4). Of these 66 examples, 49 were generated in every run, but there were a few outliers. Inspecting these manually, it was clear that they were indeed quite similar, most of them being multiple sequences of `deposit/withdraw`, of which there are many swap-examples that differ only very slightly in their reuse of integers. To a human reader, it is obvious these are duplicated examples, but since the heuristic cannot assume anything about the banking program, these cannot be avoided, as the fact that the values differ may be significant in other programs. It does seem though, that these sort of examples can take a while to find, explaining why they do not turn up in each run like most of the other examples. The full distribution of the appearance frequency of examples is shown in Table 4.1.

Table 4.1: Results of testing the frequency at which examples appear when running the combined heuristic 20 times, generating a total of 66 unique examples.

Appearance frequency	Number of examples	Percentage
$n < 10\%$	5	7.6%
$10\% \leq n < 25\%$	5	7.6%
$25\% \leq n < 50\%$	5	7.6%
$50\% \leq n < 75\%$	2	3.0%
$75\% \leq n < 90\%$	0	0%
$90\% \leq n < 100\%$	0	0%
$n = 100\%$	49	74.2%

The results of this test agree with the general impressions of the examples during development. Most examples remain the same across runs, with a few rare outliers. In the sample test, and also often during development, the most rare ones were the examples displaying "unchanged" behaviour, found by the extended delete-heuristic.

With these results suggesting the combined heuristic was generating a rather stable set of examples (apart from the variance in the humanly identifiable redundant tests), the approach for selecting the set used for evaluation was fairly simple: running the generation for long enough, specifically with `eqc:numtests` set to $10,000$.

The final sets of examples from both the cover- and combined heuristic can be found in Appendix B.1 and B.2. There are 22 examples in the cover set, and 55 examples from the combined heuristic.

4.3.2 Selection of tasks

As described in Section 3.4.4, the tasks were constructed manually, using randomly generated sequences as a starting point, also incorporating failing preconditions (as the QuickCheck generators do not generate such sequences). When constructing the tasks, it was made sure that a few important scenarios were included:

- Failing preconditions
- Persisting state after `close-open`
- Users being logged out after closing the bank
- Several successful `deposit` and `withdraw` operations with a changing balance
- Using several users and accounts in the same task

These scenarios were chosen as particularly interesting, as they require a deeper understanding of the banking program. The final set of 7 tasks can be found in Appendix B.3, with command sequence length varying from 4 to 21.

4.3.3 Experiment results

The result of the experiment was collected as a list of test scores for the two methods. The raw data collected is displayed in Table 4.2. Information on the collected data can be found in Table 4.3. The experiment was carried out with 22 participants, corresponding to 11 results from each of the two groups. Overall, a majority of the students finished all the tasks, and the number of answered tasks from each group was not very different. Since there were that many more examples to look through for the participants using the examples from the combined heuristic, one could have thought they would struggle more to finish in the limited time available (15 minutes), but this seemed to not be an issue.

Table 4.2: The number of points scored by each student in the evaluation experiment. The maximum number of points possible to score was 69.

Coverage	53	54	30	32	22	39	32	16	58	34	59
The combined heuristic	58	58	67	56	48	52	46	45	36	39	53

Table 4.3: Statistics of the data collected during the evaluation experiment.

	Coverage	**The combined heuristic**
Number of samples	11	11
Mean	39	50.727
Standard deviation	14.846	9.067
	Computed statistics	
***t*-statistic**	2.236	
Degree of freedom	18.054	
Pooled standard dev.	12.301	
Cohen's *d*	0.953	

Given the data collected, the t-statistic was computed to be 2.236, and the degree of freedom 18.054. Calculating the p-value for the t-distribution using `scipy` with these values, the resulting p-value is $0.0395 < 0.05$. Thus, the Welch t-test suggest that there is a significant difference in the performance of the two groups, indicating that the good examples from the heuristic perform better than those generated with full coverage. Furthermore, the d-value of 0.953 suggest that the effect size of the test is rather large. However, the sample size is still quite small, and the results should therefore be seen purely as an indication, and larger sample sizes would be needed to draw stronger conclusions from the results. Unfortunately, this was not possible in the limited time frame of the thesis.

One initial thought that came up after reviewing the results of the experiment, along with a few comments from the participants, was the fact that the students that received the cover-examples had a hard time grasping the concept of calls not being allowed due to preconditions. To test this, the scores were recalculated, disregarding any points gained from commands with failing preconditions. The resulting statistics did reduce the advantage of the combined heuristic, the mean score decreased to

46.273 and 36 respectively. The resulting t-statistic was 2.067 with 17.895 degrees of freedom, corresponding to a p-value of $0.0549 \not< 0.05$, and the d-value 0.881, suggesting that the power of the test decreases in this case. However, considering that the lack of pre-conditions essentially is a weakness in the cover-heuristic, and a strength for the combined heuristic, these results aren't necessarily fair either, and not that much worse than the original results.

4.3.4 Threats to validity

Even though the experiment indicates that the combined heuristic performs better than the basic cover-heuristic, one must consider a few important factors. The first, and most important, is the small sample size, which weakens the strength of the result. Another issue is the fact that the example program used is based on a real-world application, and thus the command names are suggestive of their behaviour. Furthermore, one could argue that the heuristic may be over-fitted to the banking example, as it has been used as part of the continuous evaluation of the heuristics during development. However, care has been taken to keep the heuristics useful in as general a setting as possible. Finally, one could argue that the return values and data used only involve very simple data-types, and more complex ones would be needed for a more realistic evaluation. In the event of an experiment with a longer time available for the students in the future, this should be mitigated by using a more complex example that is not so relatable, and this example should not be used in the development of the heuristic. Of course, if possible, the experiment should be carried out with a larger sample size.

4.3.5 Observations during the experiment

During the evaluation experiment, one participant commented on the fact that it was not clear from the examples that two users could be logged in at the same time. This also showed that it was not entirely clear how the commands were run. Should one consider them as run from a specific user, in which case it makes no sense to login two users at the same time, or from the bank's point of view, where it would make sense to have multiple users logging in independently? Now, this also led to some thoughts on whether the heuristic could actually find an example of two users logging in, and it seems to not be very likely. The following is an example that would show this behaviour:

```
1. open() -> ok
2. create_user(u1, p1) -> {u1, p1}
3. create_user(u2, p2) -> {u2, p2}
4. login(u1, p1) -> ok
5. login(u2, p2) -> ok
```

First of all, the delete-heuristic would not be able to find this example, as deleting either of the `login`-commands would not affect the other. Furthermore, deleting one of the `create_user` commands would affect the corresponding `login`, but the commands related to the other user would be shrunk away.

For the swap-heuristic, changing both the arguments of command 4 to `{u2, p2}` would affect command 5, and thus generate the example, but the problem is that the following example would get the same feature:

```
1. open() -> ok
2. create_user(u1, p1) -> {u1, p1}
3. login(u2, p2) -> false
4. login(u1, p1) -> ok
```

Changing the arguments of command 3 to `{u1,p1}` changes the behaviour of command 4, and the current abstraction code could not tell these two apart. Of course, this suggests the need for further tuning of the swap-heuristic, and is definitely a case to consider when developing the heuristic further.

4.4 General thoughts

This section presents some general interesting results found and insights gained during the thesis, both related to the concept of good examples in particular and property-based testing in general.

4.4.1 Features

When developing or extending heuristics, one of the most important design choices is what to include in the feature data structure. As the feature is what controls the way QuickCheck generates its feature-based test suite, getting it right is what determines the examples that are found. It is very hard to get right though, as most of the time, you will get precisely what you asked for, even though that is not what was intended. For instance, if the feature of the multi-delete would be the entire sequence of deleted commands, (abstracted to not account for specific variable values) the result would be an infinite stream of examples, as new commands could be added arbitrarily to obtain new examples with unique features. A better approach is to just include the first command, which is how it was implemented in the final version.

4.4.2 The duality of properties

When writing properties to be used during development and integration of Find-Examples, it quickly became evident that there is no one way of doing it correctly. Quite on the contrary, one must thoroughly consider the purpose of the testing before writing a state machine specification. The reason for this is that there are often several ways, all valid, to write each pre- and post-condition, but also the input generators. One example of this is the account-creation operation in the bank-example. One way to write it is to be very forgiving, and let commands be generated whenever the bank is open. This is good for finding odd errors, for example if a non-existing user could create an account. However, when good examples are our goal, we are not interested as much with the defensive cases, and would rather have the account be created for a user that has been active before, and preferably also using the correct password, as an account is useful to move forward in further tests. To do this, we would want the command to only be generated when at least one user is logged in, and thus we need another precondition.

This might seem like a good idea for getting more concise and focused testing, but it turns out it can lead to some very confusing good examples found by the original delete-heuristic. Consider the following example:

```
1. open() -> ok
2. create_user(u1, p1) -> {u1, p1}
3. login(u1, p1) -> ok
4. create_account(a2, u2) -> false
```

This is a valid example, it should not crash, even though the fourth command was unsuccessful. This makes it a valid candidate for being a good example. Consider now if we were to delete the third command. Intuitively, nothing should change, the last call should still be unsuccessful, but it should not fail. But remember that we needed a new precondition for the `create_account/2` command, namely that there must be a logged in user! So now, FindExamples is very likely to produce the "good" example:

```
Deleting command 3 changes the behaviour of 4:
...
  4. create_account(a2, u2) -> call not allowed
```

For a user, this makes no sense, as the intuitive behaviour would be that the call to `create_account` would still return false. Therefore, one must be very careful to craft the state machine specification to be reasonably forgiving, while still focusing it enough to get decent data within the first 100 generated command sequences. It may even be worth considering writing different modules for a single program. Using undirected testing, unexpected behaviour can be tested, which also lets the heuristic of FindExamples work freely to find good examples, for instance of how bad input is handled. On the other hand more directed testing, with stricter preconditions and generators, is more likely to find "deeper" bugs that require certain setup before being triggered. Such testing also enables the finding of good examples involving more hard-to-reach behaviour of the system.

4.4.3 Limitations of fully automated testing

Fully automated testing is very convenient for developers as it requires no extra effort. During this thesis, it has become clear though, that the fact that the examples must be generated completely automatically limits the quality of the examples found, as the heuristics cannot know anything that is not explicitly stated through pre- or post-conditions. That is, the heuristics cannot know how to interpret the return values of commands. An attempt to increase the amount of information available was made through the extension of the delete-heuristic, by having the user provide a function defining what values are deemed negative. Without this information, the heuristic is very likely to generate examples where the “interesting” part is actually a command that is unsuccessful.

An example of this is the standard behaviour of the banking example, where `false` is returned for valid but unsuccessful commands. The reason for this is that we want to make sure the program can handle such input without crashing, and not testing it would mean we could miss such bugs when running the property for tests. Even though the user understands that sequences of commands that all return false are not very interesting, the heuristic cannot know that without the extra `negative` function, and would thus consider those “good”. For instance, a sequence ending with a successful withdraw operation is very likely to display some interesting behaviour, whereas an unsuccessful one may contain no interesting information at all. Thus, knowing which is which helps greatly in finding examples, especially in finding unchanged behaviour after deleting a set of commands from a sequence.

However, for some programs, it may not be possible to statically define a function that can decide whether a return value is positive or negative, as it may depend on the context in which it was returned. In such cases, some extra annotation may be necessary, where the user could define functions for deciding whether the running of a command was positive or negative also given the context the command was executed. Such an extension would likely require change to QuickCheck itself, as it would have to be part of the state machine specification, involving passing the state before each call to this function. For this reason, it is outside the scope of this thesis.

5 Conclusions

This chapter includes the most important results found and conclusions drawn when working with this project. Furthermore, it contains a section on possible ideas for future work in this area that will not be investigated within the scope of this thesis.

5.1 Integration

The prototype tool FindExamples has been successfully integrated into the QuickCheck CI web service. The programmer's interface to the tool has been greatly simplified, making the tool much easier to use. Using the example generation is now as easy as wrapping your property with another function call. Furthermore, the integration makes the storing of and access to the examples generated much easier, especially as a program is developed over time.

For QuickCheck CI, the integration is quite simple, it requires no new dependencies (except for MoreBugs, which is likely to become part of regular QuickCheck anyway). Also, since there were already examples present, albeit only counterexamples directly from QuickCheck, the database does not need to be modified, thus greatly reducing the effort needed to integrate it into the production system in the future.

Furthermore, the FindExamples tool was extended to also report bugs that it finds. The previous version would not do anything about failing command sequences, but as is explained in Section 3.2.2, if there are bugs that require specific precomputation, FindExamples may stumble upon such bugs as QuickCheck is unlikely to find them given only the standard 100 runs.

5.2 Improvement of heuristics

Several new ideas for heuristics have been proposed, and a combination of them has been selected as the most successful one. It comprises of an improved version of the original delete-heuristic along with the new idea of a swap-heuristic. The improvements include the ability to not only delete single commands, but also sequences that do not affect the outcome of the rest of the program. The swap-heuristic finds examples by changing the arguments of commands instead of deleting them altogether. When combined, these two heuristics seem to perform rather well, and a few new kinds of examples that had not been found with the original heuristic have been found for the example programs used during development. This includes the family of examples that show state preserving behaviour, for instance the fact that a `dets` table retains its contents after closing and reopening it.

5.2.1 Evaluation

An experiment was carried out to verify the usefulness of the heuristic against the most straightforward and obvious approach: coverage. By testing the very purpose of the examples – conveying the behaviour of a program in a simple way – this experiment indicates that the good examples chosen by the combined heuristic outperform those chosen by generating a test suite with full expression coverage of both the underlying code and the QuickCheck model. The conducted experiment was quite small in size (22 participants), and should thus be seen only as an indicative result.

5.3 Future work

While working on this project, many good ideas have come up that have been impossible to realize due to the limited time available. This section presents some of these ideas that would be good to follow up further and more thoroughly.

5.3.1 Presenting examples

Working with the examples in QuickCheck CI, it became clear that there could be much work done in simply finding a better way of printing the examples. Throughout this work, most of the printing has been done directly to the terminal, limiting the flexibility of how examples could be printed. In a web application though, one could imagine endless ways of presenting examples, or even groups of examples, as it is very often the case that examples show some common prefix, and could thus be presented together in some way. One example of an idea on how to print examples is for the swap examples, where one could present them in a tree-like structure;

```
                              open() -> ok
                              create_user(u1, p1) -> ok
                               /             \
login(u1, p1) -> ok                           login(u1, p2) -> false
create_account(a1, u1) -> ok                  create_account(a1, u1) -> false
```

where the resulting behaviour of changing the password for user `u1` is shown clearly.

Another idea that was originally proposed by Thomas Arts, is to extend QuickCheck CI to be able to present examples generated by FindExamples in more than one way. The more verbose type printing used now, with the example containing the reason for it being interesting from the heuristic's point of view, may not be of interest to a user, instead they would rather only see a set of regular examples (command sequences) in the common case, and only see the longer representation on demand, and also be able to find the corresponding full example containing the reason for it being considered interesting. This could include some way of linking together different examples that are similar, enabling the user to browse through the examples. As this would require deeper changes to the way examples are handled in QuickCheck CI, it is beyond the scope of this thesis, but it would be a nice way of showing the examples.

5.3.2 Heuristics

There have been many ideas for new or extended heuristics throughout this thesis, and unfortunately there has not been enough time to further investigate all of them.

One example of this is a possible extension to the idea of combining heuristics, by instead of running the heuristics basically as they are (in that as soon as a new feature is found, it is returned), one could imagine using an evolutionary algorithm, more specifically a genetic algorithm, that could evolve sequences of commands to find good examples. Genetic algorithms are usually used in optimization problems where the input space is too large for analytical optimization. However, it has also been used as a method for random testing [35], [36]. The regular operations of genetic algorithms are easily applicable to command sequences, where each sequence can be seen as a chromosome, and each command a gene. It would thus be fairly straightforward to implement crossover and mutation, and the initialization is already fully implemented in QuickCheck as command generators. The only problem, and it is the big one, is to develop a fitness function for a chromosome, that is, a command sequence. This problem is similar to the very problem this thesis seeks to solve, but this approach would make the combination of individual heuristics simpler, as one could quantify and weigh the results of different heuristics together instead of the more binary approach used today.

Another idea that has been discussed is the use of the model state to find examples. Often, the fact that the state is changing as the result of a command indicates that the command was in some way interesting. However, it is not trivial to devise a strategy for deciding what changes to the state are actually interesting, and the fact that the state changes might say nothing at all. One could imagine programs where every call updates the state in some way, for example by incrementing a counter, in which case only looking at whether or not the state is changing will not work. It would however be interesting to look at combining this idea with some other heuristic (for instance cover) and see what kind of examples are found.

References

[1] K. Claessen and J. Hughes, “QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs”, in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, New York, NY, USA, 2000, pp. 268–279.

[2] T. Arts, J. Hughes, J. Johansson and U. Wiger, “Testing telecoms software with Quviq QuickCheck”, in *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, ACM, 2006, pp. 2–10.

[3] D. Stewart and S. Janssen, “XMonad: A Tiling Window Manager”, in *Proceedings of the ACM SIGPLAN workshop on Haskell*, 2007.

[4] D. Roundy, “Darcs: Distributed Version Management in Haskell”, *Haskell ’05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pp. 1–4, 2005.

[5] T. Arts, J. Hughes, U. Norell and H. Svensson, “Testing AUTOSAR software with QuickCheck”, in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, IEEE, 2015, pp. 1–4.

[6] A. F. Yamashita, A. Bergqvist and T. Arts, “Experiences from testing a radiotherapy support system with QuickCheck”, in *Tests and Proofs : Papers Presented at the Second International Conference TAP 2008 Bernhard Beckert Reports of the Faculty of Informatics*, 5, 2008.

[7] *Prowess: property based testing for web-services.* [Online]. Available: `http://www.prowessproject.eu` (visited on 16/03/2016).

[8] S. L. Peyton Jones, J. Hughes and J. Launchbury, “How to give a good research talk”, *ACM SIGPLAN Notices*, vol. 28, no. 11, pp. 9–12, 1993.

[9] T. Y. Chen, H. Leung and I. K. Mak, “Adaptive Random Testing”, *Advances in Computer Science - ASIAN 2004*, pp. 3156–3157, 2005.

[10] P. Godefroid, M. Levin and D. Molnar, “Automated whitebox fuzz testing”, *NDSS*, vol. 8, pp. 151–166, 2008.

[11] C. Cadar, P. Godefroid, S. Khurshid, C. S. Pasareanu, K. Sen, N. Tillmann and W. Visser, “Symbolic execution for software testing in practice: preliminary assessment”, *2011 33rd International Conference on Software Engineering (ICSE)*, pp. 1066–1071, 2011.

[12] K. Sen, D. Marinov and G. Agha, “CUTE : A Concolic Unit Testing Engine for C”, *Program*, vol. 30, pp. 263–272, 2005.

[13] K. Beck, *Extreme programming explained: embrace change.* Addison-Wesley, 2000.

[14] A. M. Turing, “On computable numbers, with an application to the entsheidungsproblem”, *Journal of Math*, vol. 58, no. 1936, pp. 345–363, 1936.

[15] A. Gerdes, J. Hughes, N. Smallbone and M. Wang, “Linking unit tests and properties”, in *Proceedings of the 14th ACM SIGPLAN Workshop on Erlang*, ACM, 2015, pp. 19–26.

[16] M. Wynne and A. Hellesoy, *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf, 2012.

[17] G. Fink and M. Bishop, “Property-Based Testing ; A New Approach to Testing for Assurance”, *Software Engineering Notes*, vol. 22, no. 4, p. 74, 1997.

[18] J. W. Duran and S. C. Ntafos, “An evaluation of random testing”, *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 438–444, 1984.

[19] C. Pacheco, S. K. Lahiri, M. D. Ernst and T. Ball, “Feedback-directed random test generation”, *Proceedings - International Conference on Software Engineering*, pp. 75–84, 2007.

[20] P. Godefroid, N. Klarlund and K. Sen, “DART: Directed Automated Random Testing”, in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA*, 2005, pp. 213–223.

[21] T. a. Henzinger, R. Jhala, R. Majumdar and G. Sutre, “Software verification with blast”, in *Model Checking Software*, 2003, pp. 235–239.

[22] D. Beyer, A. J. Chlipala, T. a. Henzinger, R. Jhala and R. Majumdar, “Generating tests from counterexamples”, *Proceedings - International Conference on Software Engineering*, vol. 26, pp. 326–335, 2004.

[23] T. Arts, N. Smallbone, R. Taylor and S. Thompson, “D3.1 Interface compliance tools and techniques”, *Prowess: Property Based Testing of Web services: EU-ICT Specific targeted research project (STREP) ICT-2011-317820*, pp. 1–39, 2011. [Online]. Available: `http://www.prowessproject.eu/wp-content/uploads/2012/10/Prowess_D3-1.pdf`.

[24] N. Smallbone and M. Wang, “D5.4 Linking unit tests and properties”, *Prowess: Property Based Testing of Web services: EU-ICT Specific targeted research project (STREP) ICT-2011-317820*, no. March, 2014. [Online]. Available: `http://www.prowessproject.eu/wp-content/uploads/2012/10/Prowess_D5-4.pdf`.

[25] R. Carlsson, “Parameterized modules in Erlang”, *Proceedings of the 2003 ACM SIGPLAN workshop on Erlang*, pp. 29–35, 2003.

[26] J. Hughes, “The design of a pretty-printing library”, in *Advanced Functional Programming*, J. Jeuring and E. Meijer, Eds., Springer-Verlag, 1995, pp. 53–96.

[27] C. Pacheco and M. D. Ernst, “Randoop: Feedback-Directed Random Testing for Java”, *Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion - OOPSLA ’07*, vol. 5, p. 815, 2007.

[28] Q. Yang, J. J. Li and D. M. Weiss, “A survey of coverage-based testing tools”, *Computer Journal*, vol. 52, no. 5, pp. 589–597, 2009.

[29] A. Arcuri and L. Briand, “A Hitchhiker’s guide to statistical tests for assessing randomized algorithms in software engineering”, *Software Testing Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.

[30] G. D. Ruxton, “The unequal variance t-test is an underused alternative to Student’s t-test and the Mann-Whitney U test”, *Behavioral Ecology*, vol. 17, no. 4, pp. 688–690, 2006.

[31] S. Nakagawa and I. C. Cuthill, “Effect size, confidence interval and statistical significance: A practical guide for biologists”, *Biological Reviews*, vol. 82, no. 4, pp. 591–605, 2007.

[32] H. Zhu, P. A. V. Hall and J. H. R. May, “Software unit test coverage and adequacy”, *ACM Computing Surveys (CSUR)*, vol. 29, no. 4, pp. 366–427, 1997.

[33] G. Gay, M. Staats, M. Whalen and M. P. E. Heimdahl, “The risks of coverage-directed test case generation”, *IEEE Transactions on Software Engineering*, vol. 41, no. 8, pp. 803–819, 2015.

[34] L. Inozemtseva and R. Holmes, “Coverage is not strongly correlated with test suite effectiveness”, *Proceedings of the 36th International Conference on Software Engineering*, pp. 435–445, 2014.

[35] J. H. Andrews, T. Menzies and F. C. H. Li, “Genetic algorithms for randomized unit testing”, *IEEE Transactions on Software Engineering*, vol. 37, no. 1, pp. 80–94, 2011.

[36] L. Baresi, P. L. Lanzi and M. Miraz, “TestFul: An evolutionary test approach for Java”, *ICST 2010 - 3rd International Conference on Software Testing, Verification and Validation*, pp. 185–194, 2010.

A

Appendix A

This appendix contains the source code for the banking example. The first module contains the actual program code, whereas the second contains the QuickCheck state machine specification used to test the program. Both of these modules are available in full from: `https://github.com/sebiva/quickcheck-ci-test`.

A.1 bank.erl

The following is a part of the implementation of a simple banking program used for testing. The actual implementation is a `gen_server`, which is needed to maintain the state, but as that is not of interest to the examples, that part of the code has been omitted here.

```
-module(bank).
-record(state, {open = false, accounts = [], customers = [], logged_in = []}).
-compile(export_all).
-behaviour(gen_server).

init(_Args) ->
  {ok, #state{}}.

open(State) when not State#state.open ->
  {State#state{open=true}, ok};
open(State) ->
  {State, false}.

close(State) when State#state.open ->
  {State#state{open=false, logged_in=[]}, ok};
close(State) ->
  {State, false}.

create_user(Name, Pwd, State) ->
  User = {Name, Pwd},
  case user_exists(Name, State) of
    false ->
      {State#state{customers = [User | State#state.customers]}, User};
    _     ->
```

```
      {State, false}
  end.

user_exists(Name, State) ->
  case lists:keyfind(Name, 1, State#state.customers) of
    false -> false;
    _     -> true
  end.

create_account(AccountName, UserName, State) ->
  Account = {AccountName, UserName},
  case logged_in(UserName, State) of
    true ->
      Accounts = State#state.accounts,
      case lists:filter(fun({{A, U}, _B})
                           -> {A, U} == Account end, Accounts) of
        [] -> {State#state{accounts = [{Account, 0} | Accounts]}, Account};
        _  -> {State, false}
      end;
    false -> {State, false}
  end.

logged_in(Name, State) ->
  lists:member(Name, State#state.logged_in).

login(Name, Pwd, State) ->
  case lists:member({Name, Pwd}, State#state.customers) of
    false -> {State, false};
    true  ->
      case logged_in(Name, State) of
        true -> {State, false};
        false -> {State#state{logged_in=[Name|State#state.logged_in]}, ok}
      end
  end.

logout(Name, State) ->
  case lists:member(Name, State#state.logged_in) of
    false -> {State, false};
    _     -> {State#state{logged_in=State#state.logged_in -- [Name]}, ok}
  end.

pwd_ok(User = {Name, _Pwd}, State) ->
  lists:member(User, State#state.customers) andalso logged_in(Name, State).

deposit(Name, Pwd, Account, Amount, State) ->
  case pwd_ok({Name, Pwd}, State) of
    true ->
      case lists:keyfind({Account, Name}, 1, State#state.accounts) of
        OldAccount = {N, Balance} ->
```

```
          NewBalance = Balance + Amount,
          {State#state{accounts=(State#state.accounts -- [OldAccount]) ++
                       [{N, NewBalance}]}, NewBalance};
        _                           -> {State, false}
      end;
    false -> {State, false}
  end.

withdraw(Name, Pwd, Account, Amount, State) ->
  case pwd_ok({Name, Pwd}, State) of
    true ->
      case lists:keyfind({Account, Name}, 1, State#state.accounts) of
        OldAccount = {N, Balance} when Balance >= Amount ->
          NewBalance = Balance - Amount,
          {State#state{accounts=(State#state.accounts -- [OldAccount]) ++
                       [{N, NewBalance}]}, NewBalance};
        _ -> {State, false}
      end;
    false -> {State, false}
  end.
```

A.2 bank_eqc.erl

To test the banking code, the following QuickCheck model was used. It contains all the code needed to model the behaviour of the bank, and the property used both to run QuickCheck and generate examples with FindExamples. Note that the API functions of the `bank` module do not match those from Section A.1, as they are calls to `gen_server` wrappers that handle the extra `State` argument.

This listing includes the properties for both the combined, cover- and swap-heuristic.

```
-module(bank_eqc).
-compile(export_all).
-compile({parse_transform, eqc_cover}).
-include_lib("eqc/include/eqc.hrl").
-include_lib("eqc/include/eqc_statem.hrl").
-record(state, {open = false :: boolean(),
                users = [] :: [{atom(), atom()}],
                accounts = [] :: [{atom(), atom(), integer()}],
                logged_in = [] :: [atom()] }).

%%%%% Generators

-define(LOW, 1).
-define(HIGH, 5).

-define(NAMES, [{?LOW, u1}, {?LOW, u2}, {?LOW, u3}]).
-define(PWDS, [{?LOW, p1}, {?LOW, p2}, {?LOW, p3}]).
```

```
-define(ACCOUNTS, [{?LOW, a1}, {?LOW, a2}, {?LOW, a3}]).

negative(false) -> true;
negative({'EXIT', _}) -> true;
negative(_) -> false.

name() -> frequency(?NAMES).
name(S) ->
  InState = [{?HIGH, Name} || {Name, _} <- S#state.users],
  frequency(InState ++ ?NAMES).
pwd() -> frequency(?PWDS).
pwd(S) ->
  InState = [{?HIGH, Pwd} || {_, Pwd} <- S#state.users],
  frequency(InState ++ ?PWDS).
account() -> frequency(?ACCOUNTS).
account(S) ->
  InState = [{?HIGH, Account} || {Account, _, _} <- S#state.accounts],
  frequency(InState ++ ?ACCOUNTS).

%%%%% State machine specification

initial_state() ->
  #state{}.

%%%%% Open
open_args(_S) -> [].
open() -> bank:open().
open_pre(S) -> not S#state.open.
open_next(S, _R, []) -> S#state{open = true}.
open_post(_S, [], R) -> R == ok.

%%%%% Close
close_args(_S) -> [].
close() -> bank:close().
close_pre(S) -> S#state.open.
close_next(S, _R, []) -> S#state{open = false, logged_in = []}.
close_post(_S, [], R) -> R == ok.

%%%%% Create user
create_user_args(_S) ->
  [name(), pwd()].

create_user(Name, Pwd) ->
  bank:create_user(Name, Pwd).

create_user_next(S, _R, [Name, Pwd]) ->
  case create_user_ok(S, Name) of
    false -> S;
    _     -> S#state{users = [{Name, Pwd} | S#state.users]}
  end.

create_user_pre(S) ->
  S#state.open.

create_user_post(S, [Name, Pwd], R) ->
  case create_user_ok(S, Name) of
    true -> R == {Name, Pwd};
    _ ->    R == false
  end.

create_user_ok(S, Name) ->
  case lists:keyfind(Name, 1, S#state.users) of
    false -> true;
    _     -> false
  end.

%%%%% Create account
create_account_args(S) ->
  [account(S), name(S)].

create_account(AccountName, Name) ->
  bank:create_account(AccountName, Name).

create_account_next(S, _R, [AName, UName]) ->
  case create_account_ok(S, {AName, UName}) of
    true  -> S#state{accounts = [{AName, UName, 0} |
                                 S#state.accounts]};
    false -> S
  end.

create_account_pre(S) ->
  S#state.open.

create_account_post(S, [AName, UName], R) ->
  Account = {AName, UName},
  case create_account_ok(S, Account) of
    true -> R == Account;
    false -> R == false
  end.

create_account_ok(S, {AName, Name}) ->
  logged_in(Name, S) andalso
    lists:filter(fun({AN, UN, _B}) ->
                     AN == AName andalso UN == Name
                 end, S#state.accounts) == [].

%%%%% Login
login_args(S) ->
```

```
  [name(S), pwd(S)].

login(Name, Pwd) ->
  bank:login(Name, Pwd).

login_pre(S) ->
  S#state.open.

login_next(S, _R, [Name, Pwd]) ->
  case pwd_ok(Name, Pwd, S) of
    false -> S;
    true -> case logged_in(Name, S) of
              true -> S;
              false -> S#state{logged_in = [Name | S#state.logged_in] }
            end
  end.

login_post(S, [Name, Pwd], R) ->
  case R of
    false -> logged_in(Name, S) orelse
               not exists(Name, S) orelse
                 not pwd_ok(Name, Pwd, S);
    ok    -> not logged_in(Name, S) andalso
               pwd_ok(Name, Pwd, S)
  end.

%%%%% Logout
logout_args(S) ->
  [name(S)].

logout(Name) ->
  bank:logout(Name).

logout_pre(S) ->
  S#state.open.

logout_next(S, _R, [Name]) ->
  case logout_ok(S, Name) of
    true  -> S#state{logged_in = S#state.logged_in -- [Name]};
    false -> S
  end.

logout_post(S, [Name], R) ->
  case logout_ok(S, Name) of
    true  -> R == ok;
    false -> R == false
  end.

logout_ok(S, Name) ->
```

```
  lists:member(Name, S#state.logged_in).

%%%%% Deposit
deposit_args(S) ->
  [name(S), pwd(S), account(S), choose(1, 20)].

deposit(Name, Pwd, Account, Amount) ->
  bank:deposit(Name, Pwd, Account, Amount).

deposit_pre(S) ->
  S#state.open.

deposit_next(S, _R, [Name, Pwd, Account, Amount]) ->
  case deposit_ok(S, Name, Pwd, Account) of
    OldAcc = {AN, UN, Bal} ->
      NewBal = Bal + Amount,
      S#state{accounts=(S#state.accounts--[OldAcc])++[{AN, UN, NewBal}]};
    false -> S
  end.

deposit_post(S, [Name, Pwd, Account, Amount], R) ->
  case deposit_ok(S, Name, Pwd, Account) of
    false -> R == false;
    {_, _, Bal} -> R == (Bal + Amount)
  end.

deposit_ok(S, Name, Pwd, Account) ->
  case logged_in(Name, S) andalso pwd_ok(Name, Pwd, S) of
    true -> case lists:filter(fun({AN, UN, _B}) ->
                                  AN == Account andalso UN == Name end,
                              S#state.accounts) of
              [OldAcc] -> OldAcc;
              _        -> false
            end;
    false -> false
  end.

%%%%% Withdraw
withdraw_args(S) ->
  [name(S), pwd(S), account(S), choose(1, 20)].

withdraw(Name, Pwd, Account, Amount) ->
  bank:withdraw(Name, Pwd, Account, Amount).

withdraw_pre(S) ->
  S#state.open.

withdraw_next(S, _R, [Name, Pwd, Account, Amount]) ->
  case withdraw_ok(S, Name, Pwd, Account, Amount) of
```

```
    OldAcc = {AN, UN, Bal} ->
      NewBal = Bal - Amount,
      S#state{accounts=(S#state.accounts--[OldAcc])++[{AN, UN, NewBal}]};
    false -> S
  end.

withdraw_post(S, [Name, Pwd, Account, Amount], R) ->
  case withdraw_ok(S, Name, Pwd, Account, Amount) of
    false -> R == false;
    {_, _, Bal} -> R == (Bal - Amount)
  end.

withdraw_ok(S, Name, Pwd, Account, Amount) ->
  case logged_in(Name, S) andalso pwd_ok(Name, Pwd, S) of
    true -> case lists:filter(fun({AN, UN, _B}) ->
                                  AN == Account andalso UN == Name end,
                              S#state.accounts) of
              [OldAcc = {_AN, _UN, Bal}] when Bal >= Amount -> OldAcc;
              _         -> false
            end;
    false -> false
  end.

logged_in(Name, S) ->
  lists:member(Name, S#state.logged_in).

exists(Name, S) ->
  lists:member(Name, lists:map(fun({N,_P}) -> N end, S#state.users)).

pwd_ok(Name, Pwd, S) ->
  lists:member({Name, Pwd}, S#state.users).

%%%%% Properties

prop_bank() ->
  ?FORALL(SwapCmds, ex_swap:gen_swapcommands(?MODULE),
          begin
            Commands = ex_swap:get_commands(SwapCmds),
            gen_server:start({global, bank}, bank, [], []),
            {H, S, Res} = run_commands(?MODULE, Commands),
            catch gen_server:stop({global, bank}),
            find_examples:generate_examples(?MODULE, SwapCmds, H, Res,
                pretty_commands(?MODULE, Commands, {H, S, Res},
                                aggregate(command_names(Commands),
                                      Res == ok)))
          end).

prop_cover() ->
  ?FORALL(Commands, commands(?MODULE),
          begin
            gen_server:start({global, bank}, bank, [], []),
            Prop = fun(_H, _S, Res) ->
                         catch gen_server:stop({global, bank}),
                         Res == ok
                     end,
            ex_cover:ex_coverage(?MODULE, Commands, Prop)
          end).

prop_swap() ->
  ?FORALL(SwapCmds, ex_swap:gen_swapcommands(?MODULE),
          begin
            Cmds = ex_swap:get_commands(SwapCmds),
            gen_server:start({global, bank}, bank, [], []),
            {H, _S, Res} = run_commands(?MODULE, Cmds),
            catch gen_server:stop({global, bank}),
            ex_swap:interesting(?MODULE, SwapCmds, H, Res)
          end).
```

B

Appendix B

In this appendix, the examples and tasks handed out to the students during the evaluation experiment are presented in full. The following sections explain their content, while the actual examples and tasks can be found on subsequent pages, as they are shown as they were given to the students, in full page format (apart from the headings that have been changed to show which is which).

B.1 Cover examples

This set of examples was generated by the cover-heuristic, running it long enough to obtain full coverage of both the bank code itself and the model.

B.2 Examples from the combined heuristic

The second set of examples covers two pages, and is the compacted form of the examples obtained by running the final combined heuristic with `eqc_numtests` set to $10,000$ (the feature based testing works by generating new examples until it cannot find any new features in the given limit, so the actual number of tests is likely to be several orders of magnitude larger).

B.3 Tasks for the students to fill in

This is the set of tasks that the students were given to solve. It includes an example of how the answers are supposed to be entered, as well as a short introduction along with some instructions. Furthermore, a few pointers were given before starting:

- The examples are sorted by the name of the last command in them, to enable quicker searching.
- The arguments to the commands in the tasks all have the correct types, so there is no need to check for the ordering of commands (`u1` is always a user and so on).
- A second reminder for them to rather put a question mark than guess.

Cover examples

bank_eqc:close/0

```
1  open()    →  ok
2  close()   →  ok
```

bank_eqc:create_account/2

```
1  open()                   →  ok
2  create_account(a3, u1)   →  false
```

```
1  open()                   →  ok
2  create_user(u2, p2)      →  {u2, p2}
3  login(u2, p2)            →  ok
4  create_account(a2, u2)   →  {a2, u2}
```

```
1  open()                   →  ok
2  create_user(u2, p1)      →  {u2, p1}
3  login(u2, p1)            →  ok
4  create_account(a2, u2)   →  {a2, u2}
5  create_account(a2, u2)   →  false
```

bank_eqc:create_user/2

```
1  open()                →  ok
2  create_user(u1, p2)   →  {u1, p2}
```

```
1  open()                →  ok
2  create_user(u1, p1)   →  {u1, p1}
3  create_user(u1, p2)   →  false
```

bank_eqc:deposit/4

```
1  open()                   →  ok
2  deposit(u1, p2, a3, 1)   →  false
```

```
1  open()                   →  ok
2  create_user(u3, p2)      →  {u3, p2}
3  deposit(u3, p2, a2, 1)   →  false
```

```
1  open()                   →  ok
2  create_user(u2, p3)      →  {u2, p3}
3  login(u2, p3)            →  ok
4  deposit(u2, p3, a1, 1)   →  false
```

```
1  open()                   →  ok
2  create_user(u1, p3)      →  {u1, p3}
3  login(u1, p3)            →  ok
4  deposit(u1, p1, a1, 1)   →  false
```

```
1  open()                   →  ok
2  create_user(u1, p2)      →  {u1, p2}
3  login(u1, p2)            →  ok
4  create_account(a2, u1)   →  {a2, u1}
5  deposit(u1, p2, a2, 1)   →  1
```

bank_eqc:login/2

```
1  open()          →  ok
2  login(u2, p3)   →  false
```

```
1  open()                →  ok
2  create_user(u2, p1)   →  {u2, p1}
3  login(u3, p3)         →  false
```

```
1  open()                →  ok
2  create_user(u2, p3)   →  {u2, p3}
3  login(u2, p3)         →  ok
```

```
1  open()                →  ok
2  create_user(u2, p2)   →  {u2, p2}
3  login(u2, p2)         →  ok
4  login(u2, p2)         →  false
```

bank_eqc:logout/1

```
1  open()       →  ok
2  logout(u2)   →  false
```

```
1  open()                →  ok
2  create_user(u2, p3)   →  {u2, p3}
3  login(u2, p3)         →  ok
4  logout(u2)            →  ok
```

bank_eqc:open/0

```
1  open()   →  ok
```

bank_eqc:withdraw/4

```
1  open()                    →  ok
2  withdraw(u3, p2, a3, 1)   →  false
```

```
1  open()                    →  ok
2  create_user(u1, p3)       →  {u1, p3}
3  login(u1, p3)             →  ok
4  withdraw(u1, p3, a2, 1)   →  false
```

```
1  open()                    →  ok
2  create_user(u2, p3)       →  {u2, p3}
3  login(u2, p3)             →  ok
4  create_account(a3, u2)    →  {a3, u2}
5  withdraw(u2, p3, a3, 1)   →  false
```

```
1  open()                    →  ok
2  create_user(u1, p2)       →  {u1, p2}
3  login(u1, p2)             →  ok
4  create_account(a1, u1)    →  {a1, u1}
5  deposit(u1, p2, a1, 1)    →  1
6  withdraw(u1, p2, a1, 1)   →  0
```

Good examples

bank_eqc:close/0

```
1   close()   →   call not allowed
```

```
1   open()    →   ok
2   close()   →   ok
```

```
1   open()    →   ok
2   close()   →   ok
3   open()    →   ok
4   close()   →   ok
```

bank_eqc:create_account/2

```
1   create_account(a2, u2)   →   call not allowed
```

```
1   open()                   →   ok
2   create_account(a2, u2)   →   false
```

```
1   open()                   →   ok
2   create_user(u2, p2)      →   {u2, p2}
3   create_account(a1, u2)   →   false
```

```
1   open()                   →   ok
2   create_user(u3, p2)      →   {u3, p2}
3   login(u3, p2)            →   ok
4   create_account(a3, u3)   →   {a3, u3}
```

```
1   open()                   →   ok
2   create_user(u3, p2)      →   {u3, p2}
3   login(u3, p2)            →   ok
4   create_account(a3, u3)   →   {a3, u3}
5   create_account(a2, u3)   →   {a2, u3}
```

```
1   open()                   →   ok
2   create_user(u1, p2)      →   {u1, p2}
3   login(u1, p2)            →   ok
4   create_account(a2, u1)   →   {a2, u1}
5   create_account(a2, u1)   →   false
```

```
1   open()                   →   ok
2   create_user(u3, p1)      →   {u3, p1}
3   login(u3, p1)            →   ok
4   logout(u3)               →   ok
5   create_account(a3, u3)   →   false
```

bank_eqc:create_user/2

```
1   create_user(u1, p2)   →   call not allowed
```

```
1   open()                →   ok
2   create_user(u1, p2)   →   {u1, p2}
```

```
1   open()                →   ok
2   create_user(u2, p2)   →   {u2, p2}
3   create_user(u3, p2)   →   {u3, p2}
```

```
1   open()                →   ok
2   create_user(u3, p2)   →   {u3, p2}
3   create_user(u2, p1)   →   {u2, p1}
```

```
1   open()                →   ok
2   create_user(u2, p2)   →   {u2, p2}
3   create_user(u2, p1)   →   false
```

```
1   open()                →   ok
2   close()               →   ok
3   open()                →   ok
4   create_user(u1, p2)   →   {u1, p2}
```

bank_eqc:deposit/4

```
1   deposit(u1, p1, a2, 1)   →   call not allowed
```

```
1   open()                   →   ok
2   deposit(u1, p1, a2, 1)   →   false
```

```
1   open()                   →   ok
2   create_user(u2, p2)      →   {u2, p2}
3   login(u2, p2)            →   ok
4   deposit(u2, p2, a1, 1)   →   false
```

```
1   open()                   →   ok
2   create_user(u2, p2)      →   {u2, p2}
3   login(u2, p2)            →   ok
4   create_account(a2, u2)   →   {a2, u2}
5   deposit(u2, p2, a1, 1)   →   false
```

```
1   open()                   →   ok
2   create_user(u3, p1)      →   {u3, p1}
3   login(u3, p1)            →   ok
4   create_account(a3, u3)   →   {a3, u3}
5   deposit(u3, p1, a3, 1)   →   1
```

```
1   open()                   →   ok
2   create_user(u2, p2)      →   {u2, p2}
3   login(u2, p2)            →   ok
4   create_account(a1, u2)   →   {a1, u2}
5   deposit(u2, p2, a1, 1)   →   1
6   deposit(u2, p2, a1, 2)   →   3
```

```
1   open()                   →   ok
2   create_user(u1, p1)      →   {u1, p1}
3   login(u1, p1)            →   ok
4   create_account(a2, u1)   →   {a2, u1}
5   logout(u1)               →   ok
6   deposit(u1, p1, a2, 1)   →   false
```

```
1   open()                   →   ok
2   create_user(u1, p1)      →   {u1, p1}
3   login(u1, p1)            →   ok
4   create_account(a2, u1)   →   {a2, u1}
5   close()                  →   ok
6   open()                   →   ok
7   deposit(u1, p1, a2, 1)   →   false
```

```
1   open()                    →   ok
2   create_user(u3, p2)       →   {u3, p2}
3   login(u3, p2)             →   ok
4   create_account(a3, u3)    →   {a3, u3}
5   deposit(u3, p2, a3, 2)    →   2
6   withdraw(u3, p2, a3, 2)   →   0
7   deposit(u3, p2, a3, 2)    →   2
```

```
1   open()                    →   ok
2   create_user(u2, p2)       →   {u2, p2}
3   login(u2, p2)             →   ok
4   create_account(a1, u2)    →   {a1, u2}
5   deposit(u2, p2, a1, 2)    →   2
6   withdraw(u2, p2, a1, 2)   →   0
7   deposit(u2, p2, a1, 1)    →   1
```

```
1   open()                   →   ok
2   create_user(u3, p1)      →   {u3, p1}
3   login(u3, p1)            →   ok
4   create_account(a3, u3)   →   {a3, u3}
5   logout(u3)               →   ok
6   login(u3, p1)            →   ok
7   deposit(u3, p1, a3, 1)   →   1
```

```
1   open()                   →   ok
2   create_user(u1, p1)      →   {u1, p1}
3   login(u1, p1)            →   ok
4   create_account(a2, u1)   →   {a2, u1}
5   close()                  →   ok
6   open()                   →   ok
7   login(u1, p1)            →   ok
8   deposit(u1, p1, a2, 1)   →   1
```

(Continued on next page)

bank_eqc:login/2

```
1  login(u2, p2)   →  call not allowed

1  open()          →  ok
2  login(u2, p2)   →  false

1  open()               →  ok
2  create_user(u2, p2)  →  {u2, p2}
3  login(u1, p1)        →  false

1  open()               →  ok
2  create_user(u2, p2)  →  {u2, p2}
3  login(u2, p2)        →  ok

1  open()               →  ok
2  create_user(u3, p2)  →  {u3, p2}
3  login(u3, p2)        →  ok
4  login(u3, p2)        →  false

1  open()               →  ok
2  create_user(u2, p2)  →  {u2, p2}
3  close()              →  ok
4  open()               →  ok
5  login(u2, p2)        →  ok

1  open()               →  ok
2  create_user(u3, p2)  →  {u3, p2}
3  login(u3, p2)        →  ok
4  logout(u3)           →  ok
5  login(u3, p2)        →  ok
```

bank_eqc:logout/1

```
1  logout(u1)   →  call not allowed

1  open()       →  ok
2  logout(u1)   →  false

1  open()               →  ok
2  create_user(u2, p2)  →  {u2, p2}
3  logout(u2)           →  false

1  open()               →  ok
2  create_user(u2, p1)  →  {u2, p1}
3  login(u2, p1)        →  ok
4  logout(u2)           →  ok

1  open()               →  ok
2  create_user(u3, p1)  →  {u3, p1}
3  login(u3, p1)        →  ok
4  logout(u3)           →  ok
5  logout(u3)           →  false

1  open()               →  ok
2  create_user(u2, p2)  →  {u2, p2}
3  login(u2, p2)        →  ok
4  close()              →  ok
5  open()               →  ok
6  login(u2, p2)        →  ok
7  logout(u2)           →  ok
```

bank_eqc:open/0

```
1  open()  →  ok

1  open()  →  ok
2  open()  →  call not allowed

1  open()   →  ok
2  close()  →  ok
3  open()   →  ok
```

bank_eqc:withdraw/4

```
1  withdraw(u3, p1, a2, 1)  →  call not allowed

1  open()                   →  ok
2  withdraw(u3, p1, a2, 1)  →  false

1  open()                   →  ok
2  create_user(u2, p2)      →  {u2, p2}
3  login(u2, p2)            →  ok
4  create_account(a1, u2)   →  {a1, u2}
5  withdraw(u2, p2, a1, 1)  →  false

1  open()                   →  ok
2  create_user(u3, p2)      →  {u3, p2}
3  login(u3, p2)            →  ok
4  create_account(a2, u3)   →  {a2, u3}
5  deposit(u3, p2, a2, 3)   →  3
6  withdraw(u3, p2, a2, 2)  →  1

1  open()                   →  ok
2  create_user(u2, p2)      →  {u2, p2}
3  login(u2, p2)            →  ok
4  create_account(a2, u2)   →  {a2, u2}
5  deposit(u2, p2, a2, 2)   →  2
6  withdraw(u2, p2, a2, 1)  →  1

1  open()                   →  ok
2  create_user(u1, p2)      →  {u1, p2}
3  login(u1, p2)            →  ok
4  create_account(a1, u1)   →  {a1, u1}
5  deposit(u1, p2, a1, 1)   →  1
6  deposit(u1, p2, a1, 2)   →  3
7  withdraw(u1, p2, a1, 3)  →  0

1  open()                   →  ok
2  create_user(u1, p2)      →  {u1, p2}
3  login(u1, p2)            →  ok
4  create_account(a2, u1)   →  {a2, u1}
5  deposit(u1, p2, a2, 1)   →  1
6  logout(u1)               →  ok
7  withdraw(u1, p2, a2, 1)  →  false

1  open()                   →  ok
2  create_user(u3, p2)      →  {u3, p2}
3  login(u3, p2)            →  ok
4  create_account(a3, u3)   →  {a3, u3}
5  deposit(u3, p2, a3, 3)   →  3
6  withdraw(u3, p2, a3, 2)  →  1
7  withdraw(u3, p2, a3, 3)  →  false

1  open()                   →  ok
2  create_user(u3, p2)      →  {u3, p2}
3  login(u3, p2)            →  ok
4  create_account(a3, u3)   →  {a3, u3}
5  deposit(u3, p2, a3, 2)   →  2
6  withdraw(u3, p2, a3, 2)  →  0
7  withdraw(u3, p2, a3, 1)  →  false

1  open()                   →  ok
2  create_user(u3, p2)      →  {u3, p2}
3  login(u3, p2)            →  ok
4  create_account(a3, u3)   →  {a3, u3}
5  deposit(u3, p2, a3, 2)   →  2
6  withdraw(u3, p2, a3, 1)  →  1
7  withdraw(u3, p2, a3, 2)  →  false

1  open()                   →  ok
2  create_user(u1, p2)      →  {u1, p2}
3  login(u1, p2)            →  ok
4  create_account(a3, u1)   →  {a3, u1}
5  deposit(u1, p2, a3, 1)   →  1
6  logout(u1)               →  ok
7  login(u1, p2)            →  ok
8  withdraw(u1, p2, a3, 1)  →  0
```

Tasks for filling in

Given the examples on the attached sheet of paper, your task is to fill in the correct return values for all the calls in the following sequences. A few pointers:

- All examples start over from the same initial state of the bank server
- If the use of a command is not allowed, "call not allowed" is put as the return value
- Mark any fields were you do not know the return value with a '?' rather than taking a wild guess.

Thank you very much for helping me by participating!

The first task is already filled in, and serves as an example of how the tasks should be solved:

1	`create_user(u1, p1)`	→	call not allowed
2	`open()`	→	ok
3	`create_user(u1, p1)`	→	{u1, p1}
4	`cmd_you_are_unsure_about()`	→	?
5	`withdraw(u1, p1, a1, 42)`	→	false

1	`open()`	→	
2	`create_user(u2, p3)`	→	
3	`login(u2, p3)`	→	
4	`deposit(u2, p3, a1, 3)`	→	
5	`logout(u3)`	→	

1	`close()`	→	
2	`open()`	→	
3	`open()`	→	
4	`close()`	→	

1	`open()`	→	
2	`create_user(u1, p3)`	→	
3	`create_user(u2, p3)`	→	
4	`create_account(a2, u1)`	→	
5	`close()`	→	
6	`deposit(u1, p3, a2, 5)`	→	

1	`open()`	→	
2	`create_account(a1, u2)`	→	
3	`create_user(u3, p2)`	→	
4	`login(u3, p2)`	→	
5	`create_account(a1, u3)`	→	
6	`create_user(u2, p1)`	→	
7	`deposit(u3, p2, a1, 4)`	→	
8	`withdraw(u3, p2, a1, 3)`	→	
9	`withdraw(u3, p2, a1, 2)`	→	

1	`open()`	→	
2	`create_user(u1, p1)`	→	
3	`create_user(u2, p2)`	→	
4	`login(u1, p1)`	→	
5	`create_account(a2, u1)`	→	
6	`close()`	→	
7	`deposit(u1, p1, a2, 7)`	→	
8	`open()`	→	
9	`deposit(u1, p1, a2, 9)`	→	
10	`withdraw(u1, p1, a2, 2)`	→	

1	`open()`	→	
2	`create_user(u1, p1)`	→	
3	`login(u1, p1)`	→	
4	`create_user(u2, p1)`	→	
5	`logout(u2)`	→	
6	`create_account(a1, u1)`	→	
7	`login(u2, p1)`	→	
8	`deposit(u1, p1, a1, 8)`	→	
9	`create_account(a2, u2)`	→	
10	`deposit(u1, p1, a2, 1)`	→	
11	`deposit(u1, p1, a1, 2)`	→	
12	`create_user(u1, p3)`	→	
13	`withdraw(u1, p1, a1, 4)`	→	
14	`deposit(u2, p1, a2, 16)`	→	

1	`open()`	→	
2	`create_user(u1, p1)`	→	
3	`create_user(u2, p1)`	→	
4	`close()`	→	
5	`create_account(a1, u1)`	→	
6	`open()`	→	
7	`create_user(u2, p1)`	→	
8	`login(u2, p1)`	→	
9	`login(u1, p1)`	→	
10	`create_account(a1, u2)`	→	
11	`deposit(u2, p1, a1, 3)`	→	
12	`deposit(u2, p1, a1, 6)`	→	
13	`deposit(u1, p1, a1, 7)`	→	
14	`withdraw(u2, p1, a1, 1)`	→	
15	`deposit(u2, p1, a1, 3)`	→	
16	`withdraw(u2, p1, a1, 7)`	→	
17	`withdraw(u2, p1, a1, 5)`	→	
18	`close()`	→	
19	`withdraw(u2, p1, a1, 1)`	→	
20	`open()`	→	
21	`withdraw(u2, p1, a1, 1)`	→	