

MODEL CONSTRUCTION, EVOLUTION, AND USE
IN TESTING OF SOFTWARE SYSTEMS

A THESIS SUBMITTED TO
THE UNIVERSITY OF KENT
IN THE SUBJECT OF COMPUTER SCIENCE
FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY.

By
Pablo Lamela Seijas
December 2017

Abstract

The ubiquity of software places emphasis on the need for techniques that allow us to ensure that software behaves as we expect it to behave. The most widely-used approach to ensuring software quality is unit testing, but this is arguably not a very efficient solution, since each test only checks that the software behaves as expected in one single scenario.

There exist more advanced techniques, like property-based testing, model-checking, and formal verification, but they usually rely on properties, models, and specifications. One source of friction faced by testers that want to use these advanced techniques is that they require the use of abstraction and, as humans, we tend to find it more difficult to think of abstract specifications than to think of concrete examples.

In this thesis, we study how to make it easier to create models that can be used for testing software. In particular, we research the creation of reusable models, ways of automating the generalisation of code and models, and ways of automating the generation of models from legacy unit tests and execution traces.

As a result, we provide techniques for generating tests from state machine models, techniques for inferring parametrised state machines from code, and refactorings that automate the introduction of abstraction for property-based testing models and code in general. All these techniques are illustrated with concrete examples and with open-source implementations that are publicly available.

Acknowledgements

First and foremost, I would like to thank my supervisor Prof. Simon J. Thompson, who always knows what to do, for giving me this opportunity, for sharing his wisdom with me, and for being so patient during all these years (which I am sure has been necessary many times).

Secondly, I would like to thank my parents, for their support, for keeping my feet on the ground, and for being a constant reference in this roller-coaster.

Thirdly, I would also like to thank participants in the Prowess project, for sharing their knowledge with me; meetings were very enlightening and it was a privilege to participate in such a diverse and knowledgeable community.

Specially, I would like to thank Dr. Miguel Ángel Francisco Fernández and Dr. Laura M. Castro Souto, in addition to having contributed in many ways (some of which are mentioned throughout this thesis), our discussions over video-conference were the foundation of James and, by extension, of the parametrised automaton.

I would also like to give a special mention to our colleagues from the University of Sheffield: Dr. Kirill Bogdanov, Dr. Ramsay Taylor, and Prof. John Derrick. It is thanks to them that I know most of what I know about state machine inference, and their ideas have probably influenced this thesis more than I am aware of.

Thanks to my examiners: Dr. Dominic Orchard, Dr. Neil Walkinshaw, and Prof. Joe Armstrong; and to my yearly review examiners: Dr. Stefan Kahrs and Dr. Eerke Boiten, for the interesting discussions, and for making this thesis better.

I also want to thank Dr. Huiqing Li, for her help and advice, for being a role model, and because her work has proven a solid foundation and inspiration.

I would like to thank my house-mates Dr. Olaf Chitil and Dr. Leishi Zhang, for welcoming me in their lovely house during all this time.

Finally, I would like to thank all my friends and family for their support; and I apologise in advance to everybody that I may have forgotten to mention.

Contents

Abstract	ii
Acknowledgements	iii
Contents	iv
List of Tables	xi
List of Figures	xii
1 Introduction	1
1.1 Testing	1
1.2 Models	2
1.3 Simplifying model creation	3
1.4 System under test	4
1.5 Property-based Testing	5
1.5.1 Models and property-based testing	6
1.6 Control and data flow	7
1.7 Summary	8
1.8 Contributions	9
1.8.1 Software contributions	10
2 Background	12
2.1 Erlang	12
2.1.1 Variables and atoms	13
2.1.2 Block expressions	13
2.1.3 Preprocessor directives and macros	13

2.1.4	Meta-programming	14
2.1.5	Frequency server	15
2.2	Finite State Machine	16
2.3	State merging FSM inference	17
2.3.1	Blue-Fringe algorithm	18
2.3.2	K-tails algorithm	20
2.4	QuickCheck	20
2.4.1	QuickCheck <code>eqc_statem</code>	21
2.4.2	QuickCheck <code>eqc_fsm</code>	23
2.5	Graphviz and <code>dot</code>	24
2.6	Design patterns	24
2.6.1	Template pattern	25
2.6.2	Façade pattern	25
3	Modelling components	27
3.1	Contributions	28
3.1.1	Software contributions	28
3.2	Background	29
3.2.1	Representational State Transfer (REST)	29
3.2.1.1	REST vs CRUD	29
3.2.2	JSON	30
3.2.3	XML	31
3.2.4	JVMTI	31
3.3	Universal interface	32
3.3.1	REST assumptions	33
3.3.2	Practical difficulties of REST in HTTP	34
3.4	Systems under test	35
3.4.1	Storage Room	36
3.4.2	Google Tasks	36
3.5	Collection model	36
3.5.1	Locating resources	37
3.5.2	Functions of the model	39
3.5.3	Finite-state machine	44
3.5.4	Generators and JSON	46

3.6	Using the model in practice	49
3.6.1	Discovering by testing	50
3.6.1.1	The existence of a trash	50
3.6.1.2	The representation of PATCH	51
3.6.1.3	Interpretation of mandatory fields	51
3.6.2	The façade pattern	52
3.7	Chapter conclusion	52
4	Differences and generalisation	54
4.1	Contributions	55
4.1.1	Software contributions	56
4.2	Background	56
4.2.1	Erlang behaviours	56
4.2.2	<code>ets</code> and <code>dets</code>	58
4.2.3	Wrangler	59
4.2.3.1	Built-in refactorings	59
4.2.3.2	Inspection tools	59
4.2.3.3	Extension mechanisms	59
4.2.4	Statechum	60
4.2.4.1	PLTSDiff	60
4.2.4.2	Synapse	61
4.3	Source parametrisation	61
4.3.1	Interactive refactoring	62
4.3.1.1	Basic refactorings and transformations	63
4.3.1.2	Behaviour extraction	65
4.3.1.3	Behaviour inlining	66
4.3.1.4	Example	67
4.3.2	Integrated refactoring	72
4.3.2.1	Tree mapping	74
4.3.2.2	Cluster construction	77
4.3.2.3	Cluster linking	80
4.3.2.4	Extra considerations	83
4.3.3	Wrangler usage	90
4.3.4	Case study	91

4.3.4.1	Frequency server	92
4.3.4.2	<code>ets</code> and <code>dets</code> tables	93
4.3.5	Integrated approach as aid in unification	98
4.3.6	Discussion	102
4.4	Model parametrisation	103
4.4.1	Variants and configurations	104
4.4.2	Experiment structure	105
4.4.3	Deallocation behaviour	105
4.4.4	Allocation behaviour	108
4.4.5	Number of initial frequencies	108
4.5	Chapter conclusions	112
5	Combining control and data	114
5.1	Contributions	116
5.1.1	Software contributions	118
5.2	Instrumentation	118
5.2.1	Static and dynamic approaches	119
5.2.1.1	Static approaches	119
5.2.1.2	Dynamic approaches	119
5.2.1.3	Our approach	120
5.2.2	Data and control flow	120
5.2.2.1	Data flow	120
5.2.2.2	Control flow	121
5.3	Architecture of the approach	122
5.3.1	JUnit and JVM	122
5.3.2	JVMTI agent and JNI	122
5.3.3	Erlang server	123
5.3.4	GraphViz and QuickCheck models	123
5.3.4.1	GraphViz	124
5.3.4.2	QuickCheck FSM	124
5.3.5	Feedback	124
5.4	Model construction	124
5.4.1	Common flow graph	125
5.4.2	Merging process	127

5.5	Example	136
5.5.1	Interpreting the model	136
5.5.2	The model is more general	139
5.6	Test generation	139
5.6.1	Building a <code>eqc_fsm</code> model	139
5.6.2	Generation of tests	144
5.7	Pilot study	147
5.7.1	Results of the pilot study	148
5.7.1.1	Number of tests required as input	149
5.7.1.2	Additional effort required to obtain useful models	149
5.7.1.3	Number of new tests generated	150
5.7.1.4	Number of bugs revealed	150
5.7.1.5	Time and computational resources required . . .	151
5.7.1.6	Developer evaluation	151
5.7.2	Second run	151
5.8	Limitations	154
5.8.1	Technical limitations	155
5.8.2	Conceptual limitations	155
5.8.3	Control tracking workaround	156
5.9	Lessons learned	157
5.9.1	Implicit relationships	157
5.9.2	Classification of traces	158
5.9.3	State inference for objects	159
5.10	Chapter conclusions	160
6	Recovering soundness	161
6.1	Contributions	163
6.1.1	Software contributions	164
6.2	Parametrised automaton formal definition	164
6.2.1	Input format	164
6.2.1.1	Alphabet	164
6.2.1.2	Symbol	165
6.2.1.3	Parametrised trace	165
6.2.1.4	Example	166

6.2.2	Parametrised automata	166
6.2.2.1	Possible sources	166
6.2.2.2	Possible source combinations	167
6.2.2.3	Parametrised automata	168
6.2.2.4	Example	168
6.2.3	Run	170
6.2.3.1	Execution state	170
6.2.3.2	Execution state transition algorithm	173
6.2.3.3	Example	175
6.3	Inference algorithm	176
6.3.1	Dependency rewriting	177
6.3.1.1	Pointing to last usage	177
6.3.2	APTA generation	178
6.3.3	State merging	180
6.3.3.1	Difference from Blue-Fringe	180
6.3.3.2	Merging procedure	182
6.3.3.3	Solving non-determinism	182
6.3.3.4	Example	183
6.3.4	Transition merging	184
6.3.4.1	Semantics and restrictions	186
6.3.4.2	Transition non-determinism	187
6.3.4.3	General approach	189
6.3.4.4	Updating the formal model	189
6.3.4.5	Example	189
6.3.4.6	Frequency server	189
6.4	Soundness	193
6.5	Chapter conclusions	196
7	Related Work	198
7.1	Testing web services	198
7.2	Modelling differences	199
7.2.1	Source parametrisation	199
7.2.1.1	IntelliJ IDEA	200
7.2.1.2	Automatic generalisation	200

7.2.1.3	Clone detection and elimination	201
7.2.1.4	Remodularisation	201
7.2.2	Model parametrisation	202
7.3	Modelling control and data	202
8	Conclusions	205
8.1	Generalisation	205
8.1.1	Components	205
8.1.2	Comparison	206
8.1.3	State merging	206
8.2	Models	206
8.3	Future work	207
8.3.1	Components	208
8.3.2	Comparison	208
8.3.3	State merging	208
	Bibliography	210
	A Frequency server base implementation	222
	B Frequency server web service main parts	225
	C JUnit tests by Interoud Innovation	231
	D James eqc_fsm for Freq Server	239

List of Tables

1	Influence of the minimum common cluster size parameter	95
2	Effect of lower and upper K on 2 run of pilot study	131
3	Diagram symbol legend	137
4	Colour legend for method nodes outline	138
5	Distribution of postconditions in tests generated	152
6	Distribution of methods that produce HTTP requests	153
7	Manual evaluation of interest for first 30 tests	153
8	Manual evaluation of positive or negative testing	153
9	Legend for parametrised automaton diagrams	171

List of Figures

1	Example prefix tree acceptor	18
2	Possible state machine for the state of the stop-watch model . . .	23
3	Example of template design pattern	25
4	Example of façade design pattern	26
5	FSM for collection model	44
6	Representation of model and façade over Storage Room	53
7	Initial code	67
8	Quasi-behaviour definition and instance after extraction	68
9	Behaviour definition and instance after adjustments	69
10	Alternative adjustments for generalisation 1	70
11	Alternative adjustments for generalisation 2	70
12	Result of inlining	71
13	Result of removing spurious parameters	72
14	Example input for the integrated refactoring	74
15	Result of applying the integrated refactoring to Figure 14	75
16	Example of pair of nodes with contiguous mapping	77
17	Tree matching and cluster construction	78
18	Example of linking	81
19	Example of fragmentation by horizontal border	84
20	Example solution to fragmentation	85
21	Example of exported variables and result combined	87
22	Output of our refactoring for modules in Figure 21	88
23	Original version of similar functions to be unified	99
24	Result of unifying the functions	101
25	Modifications for noop deallocation	106
26	Both deallocation behaviours side by side	107

27	Differences between <code>cannot</code> and <code>noop</code> deallocation	107
28	Modifications for <code>lifo</code> allocation	108
29	Both allocation behaviours side by side	109
30	Differences between <code>smallf</code> and <code>lifo</code> allocation	109
31	Modifications for configuration with 3 available frequencies	110
32	Behaviour for 3 frequencies: <code>{3, cannot, lifo}</code>	111
33	Differences between 2 and 3 available frequencies	111
34	Frequency server state machine with data and control flow	115
35	Architecture of James	122
36	Example of abstracted out code	126
37	Example of external code included because of dependencies	128
38	Merging algorithm base function pseudo-code	132
39	Equivalent subtree initialisation pseudo-code	133
40	Example subtree data type pseudo-code	133
41	Equivalent subtree search pseudo-code	134
42	Diagram extracted by James from the Frequency server	140
43	Slice of diagram displaying exceptional behaviour	141
44	Chicken and egg problem in data generation	143
45	Diagram representation of test generated by James	145
46	Example of test generated by James	146
47	Representation of example parametrised automaton	172
48	APTA tree as it would be generated by Blue-Fringe	179
49	APTA tree as generated by the parametrised algorithm	181
50	State machine after state merging process	185
51	State machine after transition merging process	190
52	Parametrised automaton for finite Frequency server	192
53	Parametrised automaton for infinite Frequency server	194
54	Linked model diagram	259

Chapter 1

Introduction

During the last few decades, we have become increasingly dependent on software. We are aware of its presence in obvious places like in smart-phones, ATMs, and check-out machines; but the list is even longer for software that has sneaked into appliances that we traditionally tend to think of as being analogue, like cars, airplanes, microwaves, televisions, elevators, and even inside some of our passports. Because of this, the need for some pieces of software to behave as expected is critical: errors in source code may translate into a lot of frustration, huge economic losses, or worse (Lyu et al. 1996).

1.1 Testing

The most common way of ensuring software complies with our expectations is through the use of *testing*. Testing allows us to compare the behaviour of a system to our expectations of it; by considering a particular experiment, carrying out the experiment both in our mental model of the system and in the actual system, and comparing both results. When made concrete, each of these experiments is called a *test case* and can be usually thought of as a set of inputs, and a set of expectations about the result of running the system on those inputs (*executing* the test case).

In particular, the most common type of test are *unit tests*. *Unit tests* are tests targeted at a single component of a system (or unit). When tests are aimed at checking the interactions of several different components they are called *integration tests*.

When we execute a test case, if the results of the execution do not comply with the expectations described by the test case, we can conclude that, either we have found an error in the system (a bug), an error in the test case (an *incorrect test case*), or an error in our mental model. We can use this information to fix the error. Testing increases our confidence in the system, since it increases the likelihood of it conforming to our expectations; and we can always repeat the process (Walkinshaw, Derrick and Guo 2009) until we are confident enough.

The entity we use as reference to decide whether a test is correct or not is also called an *oracle*¹. In the case when the reference is a mental model, we consider the *oracle* to be a human oracle; but we can also use other kinds of oracle, for example, we can use models, or other implementations.

1.2 Models

Following Dijkstra’s famous dictum², unit tests have a limited effectiveness. They can only cover a finite and predefined set of scenarios, and they are also costly to write. Techniques like model-checking, formal verification, and property-based testing, have proved to be more powerful and effective than unit testing. Nevertheless, all of these techniques require users to specify some kind of model or property of the system. Independently of the cost required for checking the equivalence or conformity between the system and the formal specification, the problem of transferring our mental model into some formal specification remains.

The concept of “model” is quite broad and it can range from a brief incomplete specification to a detailed description of a system. In this work, we use the words specification and model interchangeably; but, often, the word “model” implies a particular representation format or methodology, whereas “specification” is seen as more general.

In the case of software, writing specifications can be seen as similar to writing programs. In fact, some of the models presented in this work are represented as code; and, in many cases, it is possible to transform specifications automatically into programs (more or less efficiently) through specific compilers or through machine learning (by using supervised learning we can train an AI system to produce

¹[https://en.wikipedia.org/wiki/Oracle_\(software_testing\)](https://en.wikipedia.org/wiki/Oracle_(software_testing)) [last accessed 13-09-17]

²“Testing shows the presence, not the absence of bugs.” (Naur and Randell 1969)

an output that minimises an error function). On the other hand, and for the same reason, writing specifications also suffers from the same problem as writing programs (the problem we are trying to solve): we can make mistakes while writing them.

That is not to say it does not make sense to write properties, specifications, or models. At the very least, they are useful in that they provide redundancy, that is, it is harder to make the same error twice, especially if the process for creating them is different. But this realization gives us insight about important properties that models must have. We can imagine they must be:

- Simple: the simpler they are, the harder it will be to make mistakes while creating them, the easier it will be to understand them, and the more likely we will be to spot errors when looking at them.
- Different from the implementation: they must provide a different perspective to the one provided by the implementation, otherwise we lose redundancy. If we use the same perspective to create both the model and the implementation then we will be more likely to make the same mistakes in both.
- Flexible: they must have enough expressive power to represent whatever we want to represent (this tends to trade-off with simplicity).
- Modifiable: we must be able to modify the model easily whenever we find there is an error in it, or if we change our mind about how the system must behave.

In the end, we look for consistency between a model and a system: an inconsistency can reveal an error either in the model or in the system.

1.3 Simplifying model creation

The use of models for system validation is more efficient than the use of tests because they are more expressive (as we said, a test can only represent a particular scenario). But the creation of models is harder and more error prone, since it is harder for humans to think about abstractions than about concrete examples. Because of this, in this thesis we focus on making the task of creating models easier. With that aim, we explore three main approaches:

- Component reuse, applied to web services, in Chapter 3.
- Example comparison and parametrisation, applied to source-code and deterministic finite automata, in Chapter 4.
- Model inference, in Chapters 5 and 6.

Another common theme throughout this thesis is that it explores different mechanisms for abstraction and generalisation. Chapter 3 studies how to model the behaviour of a type of component generically, so that the model can be reused for various web services. Chapter 4 illustrates how to generalise commonalities automatically, between pairs of similar models and between pairs of similar implementations. And Chapters 5 and 6 build upon existing regular inference techniques to improve their ability to automate the creation of models from examples of their behaviour. By regular inference we mean the inference of finite-state machines (FSMs) that represent regular languages (usually from examples of words in the language and, in some cases, also words not in the language).

1.4 System under test

In order to be able to think generically about a system that we want to model or test, it is useful to define an interface to articulate some assumptions that we can use to interact with the system.

All experiments in this thesis treat the system under test (SUT) as if it were a black-box, this way we avoid relying on implementation details of the SUT. Whenever we try an approach that relies on the existence of some “source code”, we use code from models or tests of the system (not the code from the actual system).

Nevertheless, we often use knowledge about their implementation in order to guide the generation of tests and the active inference of models. Furthermore, we usually require systems under test to have three important properties:

- Stateful: we focus on modelling systems that have state. One of the most basic examples of stateful systems is the electronic flip-flop: even if you know what the current inputs to the flip-flop are, you cannot predict the outputs of the flip-flop without knowing its internal state or the previous inputs (the

input history). There is work on inference of properties for functions that pays attention only to their inputs and outputs (Claessen, Smallbone and Hughes 2010; Ernst et al. 2007), but this type of analysis is out of the scope of this thesis. We will not try to find specific properties encoded on inputs but properties about the way different inputs affect the state of the system.

- **Deterministic:** we assume that, for equal inputs, the response of the system will always be the same.
- **Resettable:** knowing that a system is deterministic does not make any difference if we cannot check what might have happened if we had used a different input instead. Thus, we also require knowing in advance an action or set of actions that, when carried out, are guaranteed to bring the system under test to its original state.

The last two properties do not lose generality, we can theoretically wrap any non-deterministic software system into a virtual machine that is deterministic and can be reset – we can use a software that emulates the architecture where the non-deterministic software was supposed to run and, since we have control over the emulator, we can record the initial state and every source of non-determinism and replay it at will. Thus, with some additional effort and complexity, we can potentially apply our approaches to any software over which we have a minimum amount of control.

On the other hand, many of the techniques described in this thesis will be ineffective for stateless systems, and for parts of systems that rely heavily on structure and abstract properties about the inputs. It is open to future work to combine the work in this thesis with stateless techniques.

1.5 Property-based Testing

We have mentioned that models can be used for several validation techniques. But, in this work, we focus on property-based testing: generation of random tests from properties (in our case from models).

In QuickCheck style (see Section 2.4 on page 20), property-based testing is done through two components:

- Generators – produce random data that can be used as input for the system under test (SUT). Generators can often be created automatically from type definitions but, in cases where input is complex and we are only interested in small subsets of it, we can also define them manually to obtain a higher probability of “interesting” inputs.
- Properties – are “predicates” (usually functions written in terms of the output, and possibly the input, of the SUT) that decide whether a generic test succeeds or not. Properties must be *necessary* but not necessarily *sufficient*, in other words: they may not convey the complete semantics of a system. Several complementary properties can be combined in order to obtain a better “coverage” of the semantics of the system.

For example, if we want to test a function that is supposed to reverse a list, we can write:

- a generator that produces random lists with random elements as input (for example, random numbers);
- a property that checks that applying the function twice to any input produced by the generator will produce the same input as a result:

```
reverse(reverse(Input)) == Input
```

The previous property would not be sufficient since, for example, applying the identity function twice will also give the input back. But it would be necessary, since reversing a list twice necessarily produces the original list.

In general, the system can be tested automatically by using the generators to produce an input, then executing the system under test with the generated input (in some way), and then checking whether property holds for the obtained output and, possibly, the input (see Section 2.4 on page 20).

1.5.1 Models and property-based testing

In property-based testing, models can be used as a way of expressing some aspects of properties and generators. These are useful, for example, if the system under test (SUT) has an internal state. We can still see input to such a system as a series

of calls, but it is easier to create a model of its internal state separately, define how to generate the calls depending on the state, and model the interactions between the state and the different calls and results instead. This is what `eqc_state` and `eqc_fsm` modules in QuickCheck do (see Sections 2.4.1 and 2.4.2 on pages 21 and 23).

1.6 Control and data flow

Through out this thesis, we often make reference to the concepts of control and data flow.

Control flow When we talk about control flow, we are making reference to how the order of events or actions in a system affects its state. For example, if we first put a glass under a tap, open the tap, close the tap, and then remove the glass from under the tap; we get a different result than if we open the tap, close the tap, put the glass under the tap, and then remove the glass from under the tap.

In particular, we usually talk about control flow in the way it is represented by a deterministic finite automaton, that is, we assume there are a number of states in which the system can be, and we describe control flow as the transitions of the system from one state to another triggered by some event or action. When talking about the control flow of a program, we usually mean the order in which the different methods, functions, or operations are executed or evaluated.

Data flow When we talk about data flow, we are making reference to how information propagates throughout a system. For example, when ordering food in a restaurant, the waiter may ask the client about the order, the client may communicate the order to the waiter, then the waiter may communicate the order to the chef, then the waiter may communicate the bill to the client, the client may communicate the intention to pay to the bank through a debit card transaction, the bank may communicate the validity of the payment to the waiter, and the waiter may confirm the transaction by giving the client a receipt.

In particular, we usually talk about data flow in the way it is represented by a data-dependency graph. When talking about the data flow of a program we

usually refer to how the result of function calls is passed as a parameter (or other mechanisms) to other function calls.

1.7 Summary

In this thesis we will discuss the following topics:

- In Chapter 2, we present previous work upon which this thesis builds.
- In Chapter 3, we study how to make model creation easier by creating reusable components. In particular, we create a reusable FSM model for a “collection component” and we apply it to two different production web services: Storage Room and Google Tasks. The “collection component” is an idiom that is used to represent a set (or collection) of resources, for example: a blog can be seen as a collection of posts.
- In Chapter 4, we study how to find commonalities between pairs of specific models with the aim of creating more general meta-models and to support the evolution of models. We do that by comparing similar models and isolating their similarities and differences. We follow two approaches:
 - In Section 4.3 (on page 61), we try to automate the introduction of abstraction, both by combining smaller refactorings and by helping the user identify commonalities between the structure of two modules and how to abstract them out.
 - In Section 4.4 (on page 103), we describe a series of experiments where we compare the FSM models of different configurations for a particular system and we analyse how these changes affect the behaviour of the system.
- In Chapter 5, we study new ways of obtaining general models from deterministic unit tests, by combining control and data flow information extracted from legacy JUnit tests and we illustrate our new mechanisms with examples from our implementation James (a tool that extracts models from JUnit tests and generates new tests from the extracted models). We also show how to use those models to generate new JUnit tests automatically.

- In Chapter 6, we refine the inference techniques from Chapter 5 to construct a new type of model (the parametrised state machine) that is built as an extension to the FSM model. We also describe a sound algorithm for inferring such a model in terms of its similarities with the algorithm Blue-Fringe.
- In Chapter 7, we present work that is related to our work as reported in this thesis.
- In Chapter 8, we comment on our conclusions and suggest some ideas for future work.
- In Appendix A, we provide the base code for the Frequency server, used in Chapter 4.
- In Appendix B, we present the classes that implement the main functionality of the web service version of the Frequency server, used in Chapter 5.
- In Appendix C, we include the source code of the tests provided by Interoud Innovation for the Frequency server web service, used in Chapter 5.
- In Appendix D, we present the main parts of the `eqc_fsm` model generated by James, resulting from work described in Chapter 5.

1.8 Contributions

The main contributions of this thesis are:

- A general model for representing the behaviour of the collection components (Section 3.5 on page 36).
- A technique for applying the model for collection components to web services that follow the REST architectural-style (Section 3.6 on page 49).
- Insights from two independent examples that illustrate how to adapt the model for collection components to real web services (Section 3.6 on page 49).
- A set of refactorings and techniques for automating the abstraction of common behaviour in concrete implementations with examples (Section 4.3 on page 61).

- Study, through experiments, of the effect of different configurations on the state machine representation of the behaviour of a system (Section 4.4 on page 103).
- Insight on one of the causes of state explosion in regular inference (Section 4.5 on page 112).
- An approach based on regular inference for modelling data and control flow of a system from an existing test suite (Section 5.4 on page 124).
- A technique for using models with data and control to generate new tests (Section 5.6 on page 139).
- A pilot study that evaluates the effectiveness of the technique for generating new tests (Section 5.7 on page 147).
- The formal definition of an extension to the regular automaton model (called parametrised automaton) that allows direct representation of symbol parameter dependency (Section 6.2 on page 164).
- An informal explanation of how to extend the Blue-Fringe algorithm to learn parametrised automata (Section 6.3 on page 176).

1.8.1 Software contributions

- As part of the experiments carried out during Chapter 3, I contributed to the `inets` module of the standard library of Erlang by adding support for the method `PATCH`. This contribution can be seen in the pull request³, which was accepted for inclusion in the `maint` branch. The `maint` branch contains small changes and is used for minor revisions, the `master` branch includes everything that is added to the `maint` branch⁴.
- I have implemented all the new refactorings described in Section 4.3 (on page 61) and are now part of Wrangler’s API⁵; but there are some refactorings described in Section 4.3 that already existed previous to this work (this

³<https://github.com/erlang/otp/pull/917> [last accessed 05-07-17]

⁴<https://github.com/erlang/otp/wiki/Branches> [last accessed 04-10-17]

⁵<https://github.com/RefactoringTools/wrangler/pull/69> [last accessed 03-08-17]

is explicitly specified when describing the particular refactorings), and there are some pre-existent generic parts of Wrangler and EDoc (Carlsson 2009), that I reused when implementing the refactorings. The input and results of the case studies have also been published in the examples folder of Wrangler. And the main refactorings have been added to the menu [Wrangler > Refactor > Behaviour Refactorings] of Wrangler’s GUI for Emacs.

- As part of the development of the refactorings described in Section 4.3 (on page 61), I also contributed with small improvements to Wrangler, like the deletion from export declarations of functions removed by transformations done using the DSL and concrete syntax⁶.
- As part of the work in Chapter 5, I implemented the whole of James, whose source code is publicly available at (Lamela Seijas and Thompson 2014).
- In order to test James previous to the pilot study, I implemented a web service version of the Frequency server, whose source-code is available in (Lamela Seijas 2014b) and partially in Appendix B, together with some examples of tests for the web service version of the Frequency server that were not used in the pilot study (Lamela Seijas 2014a). Note that there also exists a separate set of tests for the web service version of the Frequency server which were implemented by Interoud Innovation, whose source-code is available in (Francisco 2014a) and also in Appendix C.
- In order to automate the feedback between James and the system under test, I implemented a Java Erlang Bridge interface that was used to automatically classify the tests generated by James during the second run of the pilot study (Lamela Seijas 2014c).
- I implemented prototypes in Haskell for the inference and classification algorithms described in Chapter 6, and their source is publicly available at (Lamela Seijas and Thompson 2016b).

⁶<https://github.com/RefactoringTools/wrangler/pull/69/commits/595496649c118c935a4af302a0b5ac91ec6aa13f> [last accessed 12-09-17]

Chapter 2

Background

In this chapter, we provide some background information about previous work that may not necessarily be similar to the work presented in this thesis, but is, nevertheless, useful in order to understand it, since it represents the foundation upon which it is built. The reader may safely skip part or all of the sections in this chapter if they are already familiar with their contents. In addition, the following chapters have references to the sections in this chapter, so it should be safe to “read it lazily”, by delaying the reading of each section until needed.

Throughout this chapter, we will briefly introduce Erlang (in Section 2.1), finite state machines (in Section 2.2), regular inference (in Section 2.3), QuickCheck (in Section 2.4), Graphviz (in Section 2.5), and the design patterns used throughout this thesis (in Section 2.6).

2.1 Erlang

Most of the code discussed in this thesis was written using Erlang (Armstrong 2007; Cesarini and Thompson 2009). Erlang is a strict, dynamically typed, functional programming language with support for higher-order functions, pattern matching, concurrency, distribution, fault-tolerance, and dynamic code reloading. Erlang’s data types include *atoms*, *numbers*, and *process identifiers*, and the compound data types *tuple*, *list*, and, more recently, *map* (Ericsson AB 1999).

Structured data such as trees can be represented through the use of compound data types. Additionally, Erlang’s preprocessor provides a mechanism to define *records* at compile time. Erlang records behave like *tuples* during runtime, but

allow developers to give identifiers to their elements. The use of *records* simplifies maintenance of code that uses them, since their elements are accessed by name instead of by position (as is the case with *tuples*).

Variables in Erlang are single assignment and most of the features of the language are side-effect free, but side-effects are possible through, for example, the use of built-in functions (BIFs) and concurrency primitives.

In the following sections, we briefly introduce some features of Erlang and related systems that are useful in order to understand the rest of this thesis.

2.1.1 Variables and atoms

In Erlang, atoms represent a distinguished value, which is immutable and simply stands for itself. The only permissible operations on atoms are checks for equality, inequality, and ordering. Erlang uses syntax to distinguish between atoms and variables: if it starts with a small letter it is an atom, if it starts with a capital letter or an underscore then it is a variable (unless it only consists of an underscore, in that case it is just a way of ignoring a value). An exception to this rule is when the name is enclosed in single quotes; for example: `'Foo'` represents the atom called `Foo`, not the variable called `Foo`.

2.1.2 Block expressions

A sequence of expressions can be grouped in a block, in a way that the block can be placed wherever a single expression can. This is done by surrounding them with the keywords `begin` and `end`. The value of the block of expressions is given by the value of the last expression in the block, but previous expressions can be used for their side-effects and because bindings created in previous expressions can be used inside the block.

2.1.3 Preprocessor directives and macros

Erlang allows the use of some directives that are resolved by the preprocessor during compilation. Some of the most common ones include:

- `define` directive can be used to create new macros.

- `include` and `include_lib` directives inline the contents of the file passed as a parameter to them.
- Conditional directives: `ifdef`, `ifndef`, `else`, and `endif`, allow the inclusion or omission of a part of the code depending on whether a particular macro has been defined at some point before in the code, analogously to their C counterparts.

In addition to the ones defined through the use of directives, there exists a number of predefined macros, but here we will only consider one:

- `?MODULE` macro is replaced by the name of the module when preprocessing its source.

2.1.4 Meta-programming

In the context of programming languages, meta-programming refers to the ability of programs to treat themselves (or other programs) as data¹; this ability, in some cases, can allow a program to modify itself in execution time. The ability of a program to modify itself makes it harder to predict what it can do. In the general case, refactoring tools, when applied to programs that modify themselves, do not have a decidable way of statically guaranteeing that a particular modification of the source of a program will not have any effect on its behaviour.

Even if we assume that the program is sandboxed (isolated from its operating system and other programs), some programming languages provide mechanisms that:

- Allow programs to inspect their own structure (reflection).
- Allow programs to dynamically decide their own control flow (meta-calls). For example, the name of the target function may be determined by a string which, in turn, may depend on the result of an arbitrary computation.

Erlang allows both reflection and meta-calls. Erlang allows both of these functionalities through the use of BIFs (Built-In Functions) including:

¹<https://en.wikipedia.org/wiki/Metaprogramming> [last accessed 03-10-17]

- `apply/2` and `apply/3` – which together with `list_to_atom/1`, allow the execution of arbitrary functions that may be anonymous or their name may not necessarily be known at compilation time.
- `module_info/0` – allows to inspect a compiled module for exported functions.

In addition, qualified function calls can be parametrised with variables, so the target module can potentially be defined at runtime; typically, function names and module names are specified through atoms. For example, the call `foo:bar()` is static (to the function `bar` in the module `foo`), but the call `Foo:bar()`, is dynamic, and the target module depends on the value of the variable `Foo`.

2.1.5 Frequency server

Throughout the whole thesis (with exception of Chapter 3), we use Frequency server as a common case study. Frequency server is a toy example extracted from the book (Cesarini and Thompson 2009). It simulates a “spectrum management” system that allows clients to allocate and deallocate frequencies while ensuring that each frequency is allocated by at most one client at a time.

The Frequency server API provides four commands:

- `start/0` – Has no parameters and starts the server unless it is already running; if it is already running it produces an exception.
- `stop/0` – Has no parameters and stops the server if it is running; if it is not running it produces an exception. When the server is stopped all occupied frequencies are marked as free.
- `allocate/0` – Has no parameters, it returns a free frequency (expressed as a number), and marks it as occupied. It returns an error if there are no free frequencies, and it produces an exception if the server is not running.
- `deallocate/1` – Takes as its only parameter a frequency (expressed as a number) and marks it as free. It produces an exception if the server is not running.

The original source code of the Frequency server can be found in the book (Cesarini and Thompson 2009), a slightly modified version can be found in Appendix A. As part of this work, I have developed a web service version of it in Java (which we use as an example in Chapter 5), the most representative parts of the source code for the implementation of the web service can be found in Appendix B, and the full source code of the web service is available at (Lamela Seijas 2014b).

2.2 Finite State Machine

A finite state machine (FSM) is a model (Hopcroft, Motwani and Ullman 2006) that can be used, for example: for design and validation of digital circuits, for lexical analysis, for search in large bodies of text, and for verification of systems like communication protocols. More generally, an FSM is a formalism that can be used to classify strings over a given alphabet (or set of symbols). The set of all languages that can be classified by an FSM is called *regular languages*.

In general, in this thesis, when we talk about FSMs we usually refer to a specific type of FSM called deterministic finite automaton (DFA). A DFA is formally defined as a “five-tuple” $(Q, \Sigma, \delta, q_0, F)$ where:

1. Q is a finite set of *states*.
2. Σ is a finite set of *input symbols*.
3. $\delta : Q \times \Sigma \rightarrow Q$ is a *transition function* that takes as argument a *state* and an *input symbol* and returns a *state*. Usually transitions are represented as arcs between states that have the labels (*input symbols*) on the arcs.
4. q_0 is a *start state* (or *initial state*), which must be one of the *states* in Q ($q_0 \in Q$).
5. F is a set of final or *accepting states*, which must be a subset of Q ($F \subseteq Q$).

For deciding whether a *string* (a sequence of *symbols*) is accepted by a DFA we run it. We start with the *initial state* (q_0) and apply the *transition function* δ to both the *state* and the first *symbol* of the *string* we want to classify. Then, we apply the *transition function* δ again to the *state* resulting of the previous application together with the second *symbol* of the *string*, and we repeat the process until

we have applied the *transition function* to the last symbol of the *string*. If the resulting *state* belongs to F then we say the DFA accepts the *string*.

Whereas the notion of accepting state is appropriate for talking about which “strings” belong to a language, when using DFAs to model software we may refer to states as normal or failing instead. In those cases, we will consider sequences of events instead of sequences of symbols or “strings” and we may refer to sequences of events as *traces*. We will also use a set of *failing states* instead of a set of *accepting states*, and thus we will consider a *positive trace* any trace represented by the DFA that does not go through any failing state, and we will consider a *negative traces* any trace that ends in a failing state. We will usually not consider traces that go through a failing state and then continue, since a failing state will typically stop the execution of the system; as a consequence, failing states will be sinks, which in turn allows us to merge all failing states into one without altering the meaning of the DFA.

2.3 State merging FSM inference

The inference algorithms used in this thesis are based on two main existing regular learning state-merging algorithms: Blue-Fringe (Lang, Pearlmutter and Price 1998), and k-tails (Biermann and Feldman 1972). Regular learning tries to find an FSM that represents a given regular language, usually from examples of words that belong to that language (or traces).

In general, state-merging algorithms start by creating a prefix tree acceptor (PTA) that accepts all the traces provided as input to the algorithm. A PTA is an FSM structured as a tree (in particular a trie) that contains all the traces provided as input to the algorithm. In a PTA, every state represents a prefix of the traces accepted, every distinct prefix is represented by a unique state, and the ancestor of each state represents the same prefix but without the last symbol; the root of the PTA represents the empty prefix. For example, in Figure 1 we show a PTA that accepts the words “seal”, “seam”, “star”, “starry”, “state”, “static”, “steam”, “step”, and “team”. Note that it is not necessary to store the whole prefix in each state, only the last symbol, since the path from the root to the node contains the rest of the prefix. Also note that, when using state merging, we usually store symbols in the transitions instead of in the nodes, this way, if symbols represent events,

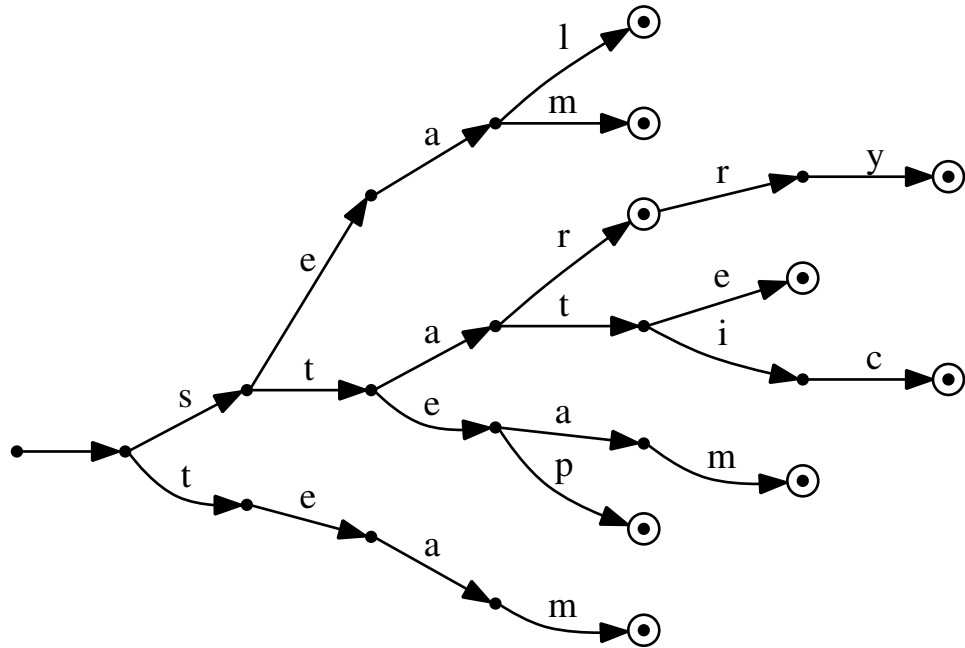


Figure 1: Example prefix tree acceptor

we can relate the states of the FSM to the states of the system.

Optionally, the PTA can be augmented with words that do not belong to the language and then it is considered an augmented prefix tree acceptor (APTA). In this thesis, we represent words that do not belong to the language by labelling the state representing the whole trace in the APTA as “negative state”.

After obtaining a PTA or an APTA, the automaton is generalised by merging states (or tree nodes) considered equivalent. Different algorithms may have different strategies for deciding which states to merge and when. In particular, Blue-Fringe and k-tails differ precisely in this strategy, and in that k-tails begins with a PTA and Blue-Fringe begins with an APTA.

2.3.1 Blue-Fringe algorithm

The Blue-Fringe algorithm (Lang, Pearlmutter and Price 1998) starts with an APTA, and iteratively chooses two states and merges them until no more states can be merged.

Merging a pair of nodes A and B consists of:

- Pointing the destination of incoming transitions of the state A to the state

B.

- Pointing the source of outgoing transitions of the state A to the state B.
- Removing the state A.
- Removing the redundant transitions in and out of the node B.
- Resolving ambiguous outgoing transitions of node B (transitions that are equal but go to different nodes from node B) by recursively merging the destination of all pairs of ambiguous transitions.

The score for merging a pair of nodes is defined as:

- If the nodes are mergeable: the number of overlapping nodes among the subtrees that have each of the nodes as a root.
- If the nodes are unmergeable: their score for merging is considered -1.

The pair of nodes to merge is chosen through the following procedure:

- We consider three types of nodes for which the following invariants hold:
 - Red nodes:
 - * Red nodes form a connected graph.
 - * Any pair of two red nodes is unmergeable.
 - Blue nodes:
 - * All non-red children of red nodes are considered blue nodes.
 - * Blue nodes are roots of isolated trees.
 - Non coloured:
 - * Any node that is not red nor blue is considered uncoloured.
- We start by setting the root of the APTA tree to red.
- We repeatedly do the first of the following actions that can be done and has not been done already:
 - Compute the score for merging a pair of one red and one blue node.

- Promote a blue node to red if it is unmergeable with any red node.
- Merge the pair of one red and one blue nodes that has the highest score.

The process finishes when all nodes have been promoted to red.

Even though this description of the Blue-Fringe algorithm is based on (Lang, Pearlmutter and Price 1998), which is very concise, we have learned many of the details about Blue-Fringe from the description provided in (Dupont et al. 2008), which is in our view more detailed.

2.3.2 K-tails algorithm

The k-tails algorithm (Biermann and Feldman 1972) is based on an equivalence relation called Nerode’s equivalence, that implies that given any regular language, strings can be classified into a finite number of equivalence classes. The equivalence class to which a string belongs is given by the set of all “extensions” (strings) that when appended to the string will produce another string that belongs to the regular language (words with the same set of valid “extensions” belong to the same equivalence class).

In k-tails algorithm, this equivalence relation is modified to consider only extensions that have at most length k (for some k). The algorithm takes a set of examples of the language to infer and a positive constant k , it generates a PTA, and then it merges states that are equivalent according to the modified equivalence relation.

If the available input traces “sufficiently characterize” some regular language, and if k is “appropriately adjusted”, the method is guaranteed to find a machine that produces the language (Biermann and Feldman 1972).

2.4 QuickCheck

QuickCheck is a tool that aids the use of property-based testing (see Section 1.5 on page 5). At the time of writing, there is an open-source version of QuickCheck for Haskell (Claessen and Hughes 2011) and a commercial version for Erlang, maintained by QuviQ (Arts et al. 2006); throughout the work described in this thesis we have used both of them. From now on, whenever we talk about QuickCheck

in this paper we mean QuviQ’s QuickCheck, except in Chapter 6, where we mean QuickCheck for Haskell instead. PropEr (Papadakis and Sagonas 2011) is an open-source property-based testing tool for Erlang that was inspired by QuickCheck and covers a number of its use cases; it also supports the creation of generators from type declarations.

In general, QuickCheck takes a term that represents a property (that may be defined by using the API provided by QuickCheck) and it tries to return a counter example that falsifies the property by trying random inputs, in the case of Erlang it is possible for this process to produce side-effects too.

Properties can be defined as functions, and random inputs are generated by a special type of term called a generator. In the case of QuickCheck for Haskell, generators can, in some cases, be created automatically by using the parameter types of the function to test; in both versions of QuickCheck it is possible to define custom generators by combining existing ones.

In the Erlang version of QuickCheck, by using the `eqc_statem` and `eqc_fsm` modules, users can describe stateful systems by providing a state machine instead of a property, we look at this in more detail in the rest of this section.

2.4.1 QuickCheck `eqc_statem`

The `eqc_statem` module (`eqc_statem` 2004) allows users to describe the properties of a stateful system by describing part of the behaviour of the system using an abstraction of its state. Thus, the model defines how the different commands affect the abstraction of the state, and provides predictions and properties on the inputs and outputs of the commands for particular executions of the system.

For example, if we want to model a stop-watch, we could define a state that has the following fields:

- `is_counting` – a boolean that indicates whether the stop-watch is timing something or not.
- `is_zero` – a boolean that indicates whether the counter of the stop-watch is in its initial value (no time has passed).
- `lap` – a time-stamp that stores when the last lap was completed.

And then we may define a series of commands (or transitions), for example: we can define a transition `start_counting`, that can occur when the stop-watch is in clear state and the user presses the `start/stop` button. Roughly speaking, we could define `start_counting` by implementing the following callbacks:

- **precondition** (when can the command be used?): `is_counting` must be `false`, `is_zero` must be `true`, and `lap` must be empty.
- **command** (how do we cause the transition on the SUT?): we press the `start/stop` button in the `stop_watch`.
- **postcondition** (what do we know about the system afterwards?): after pressing the `start/stop` button, the time in the display of the stop-watch increases from its previous value.
- **next_state** (how do we update the state of the model?): we change the value of `is_counting` to `true` and `is_zero` to `false`.

By defining a number of events like `start_counting`, we would eventually obtain a model of the stop-watch that describes its supposed behaviour.

We have omitted and simplified some details for the sake of clarity; they can be found in the documentation of QuviQ's QuickCheck (QuickCheck documentation 2006). But, in general, the model is implemented as a series of callbacks.

It is worth mentioning that, initially, QuickCheck generates test cases (sequences of commands or transitions) symbolically. Because of this, neither the **precondition** nor **next_state** callbacks must rely on the concrete results produced by the SUT, but on the state representation. After a valid symbolic test case has been generated, its commands are executed against the SUT (by using the command callback) and the postconditions are checked against the actual result produced by the SUT.

The moment a postcondition fails, or the SUT crashes, QuickCheck will try to find the smallest sequence of commands that fails (this process is called shrinking), and it will report it as a counterexample (showing that the model and the SUT are behaving in an inconsistent manner).

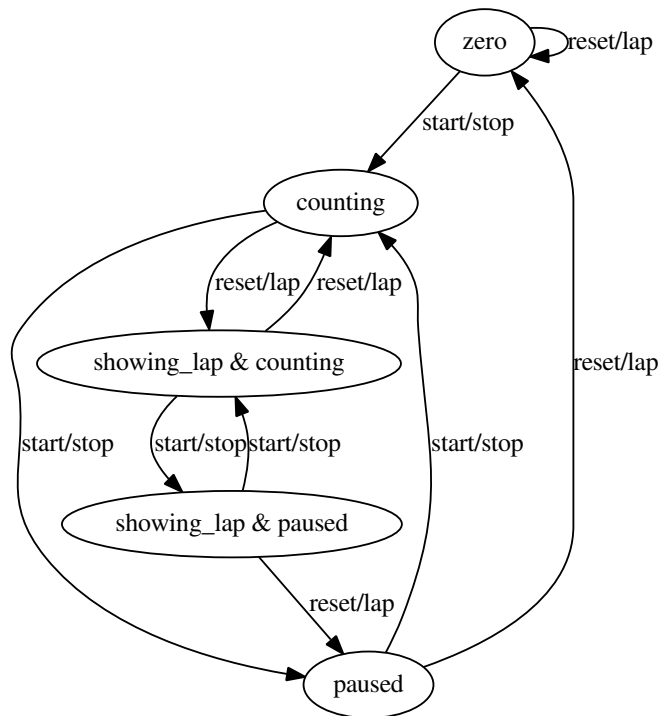


Figure 2: Possible state machine for the state of the stop-watch model

2.4.2 QuickCheck `eqc_fsm`

The `eqc_fsm` module (`eqc_fsm 2006`) provides roughly the same functionality as `eqc_statem`, but it has a different way of representing the SUT: part of the state is represented through an FSM, and the transitions that can be applied are constrained by the FSM. Since the state machine is finite, the possible values for this part of the state must be specified in advance and there must be a finite number of them. In exchange for this extra requirement, `eqc_fsm` allows the user to generate weights for transitions automatically (in order to improve the coverage of the tests generated) and to generate a graphical representation of the FSM and weights in the model.

Returning to the example of the stop-watch, we could define the state in terms of the FSM shown in Figure 2. We can imagine that, if we use the `eqc_statem` model, the state “`showing_lap & paused`” will not be explored as often as other states by the tests generated by QuickCheck, since the number of paths of a given length that reach it is smaller (the shortest is: `start/stop`, `reset/lap`, and `start/stop`). By providing the full state machine, QuickCheck is able to

generate weights for transitions automatically, in order to compensate for this effect.

2.5 Graphviz and dot

Graphviz is a set of open-source graph visualisation software tools. Graphviz consists of several tools that generate different types of graphs from descriptions written in a domain specific text language (Graphviz 1999).

Throughout this thesis we have made extensive use of the `dot` tool and many of the figures in this dissertation have been directly generated by it. `dot` is distributed as part of the Graphviz suite and specialises in “hierarchical” or layered drawings of directed graphs.

Graphviz tools automatically calculate appropriate layouts for graphs and allow the rendering of these in many formats, including: `pdf`, `svg`, `png`, and direct display to an X11 compatible server.

2.6 Design patterns

Design patterns are reusable solutions to common problems (Vlissides et al. 1995). When working on this thesis we have encountered problems that have been documented before together with their solutions and, when possible, we have tried to reuse existing tested solutions. This approach allows us to spend more time in the problem we are trying to solve, and less time in those problems that have been solved already. In particular, there exists a series of design patterns that are used in programming and system design, often in an object-oriented context. We have used at least two of these patterns (namely template and façade patterns), and in this section we briefly introduce them in the context of Erlang.

There also exist design patterns specific to Erlang OTP² that define how to structure Erlang code in terms of processes, modules, and directories; but we will not be covering those in this thesis. Nevertheless, it is worth noting that the Erlang OTP behaviour pattern is similar to the template pattern, even though Erlang behaviours are used to model processes instead of arbitrary code; for that

²http://erlang.org/doc/design_principles/des_princ.html [last accessed 04-10-17]

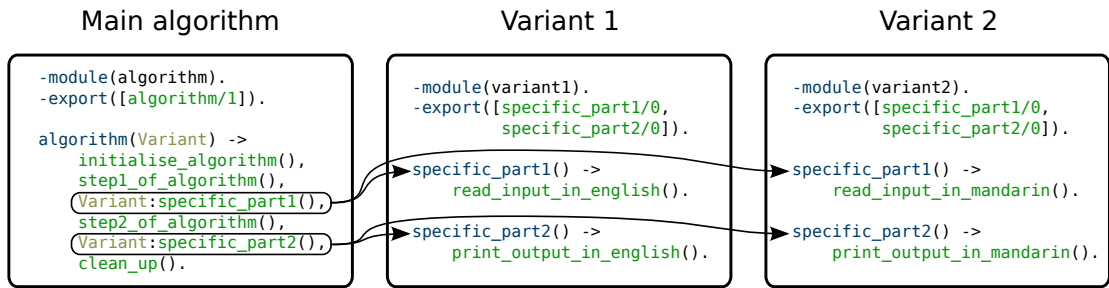


Figure 3: Example of template design pattern

reason, in Chapter 4, we use the infrastructure provided by Erlang behaviours to implement the template pattern.

2.6.1 Template pattern

The template pattern allows us to provide several alternatives for steps of an algorithm, while still describing the common parts and the main structure of the algorithm in a single place, as well as to choose between the different alternatives at runtime.

In object-oriented languages, template pattern is often implemented by inserting calls to virtual methods of the superclass that contain the parts of the algorithm that vary, these calls represent the “gaps” to fill in the algorithm variants. Subclasses that represent the different variants can establish their own content for the “gaps” by implementing or overriding the virtual methods. In Erlang, an analogous design can be achieved by using calls whose target module is dynamically qualified (see Figure 3).

In the example (Figure 3), we can see that the algorithm takes the module implementing the variant as a parameter, and variants only need to implement the parts of the algorithm that are not generic.

2.6.2 Façade pattern

The façade pattern allows us to use a simple interface to control a more complicated one (Freeman et al. 2004). This is sometimes done by creating a class or module that has a simple interface representing the high-level functionality we are interested in, and this class or module implements the high-level functionality

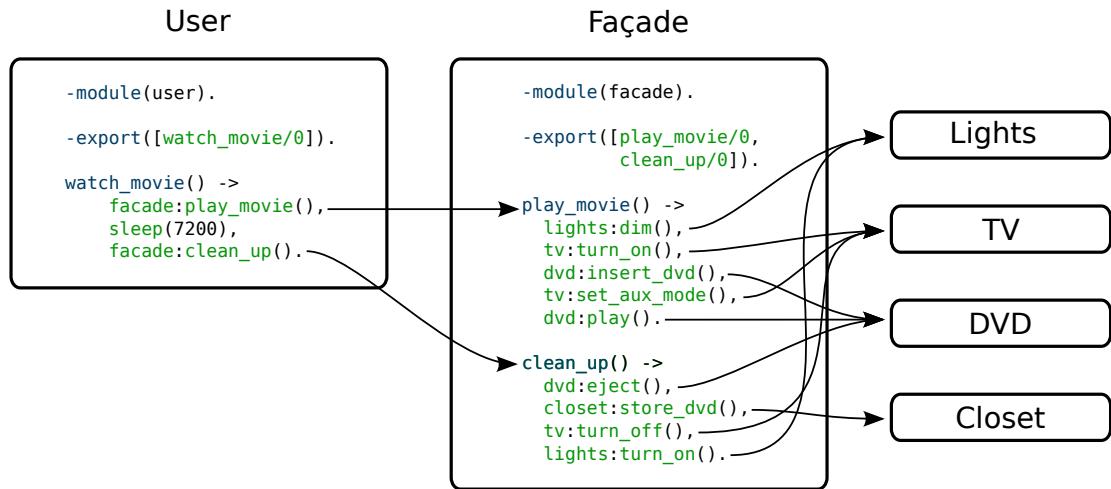


Figure 4: Example of façade design pattern

in terms of low-level functionality which may be more complex and specific (see Figure 4).

In the example (Figure 4), we can see that the `facade` module provides a simple interface that combines a series of calls to several systems, and allows the module `user` to describe the algorithm in terms of higher level calls.

Chapter 3

Modelling components

The first way through which we can make the creation of models easier is by reuse. Reuse is something that is taken for granted in the creation of software, but not so much in testing, especially in the creation of models. There exist generic testing and modelling frameworks, but it is hard to find libraries of tests suites that can be composed to create bigger test suites; or libraries of meta-models that can be composed to form bigger models.

But there are patterns (see design patterns in Section 2.6 on page 24), and there are reusable components. For example, REST architectural style (see Section 3.2.1) encourages the reuse of standardised components and microformats (Richardson, Amundsen and Ruby 2013).

Thus, in this chapter, we experiment with reusing models for components. In particular, we have chosen to model CRUD behaviour (see Section 3.2.1.1) of web services that follow the architectural style REST, that is, RESTful web services (see Section 3.2.1).

The reason why we thought in the first place that REST and CRUD would allow us to automate testing is that they provide a series of pre-established requirements and restrictions.

In general, restrictions translate into:

- Common behaviour – behaviour whose tests and models can potentially be reused between different systems.
- Common assumptions – that allow models to be defined on a higher level (since some aspects are already assumed), and thus in a more concise way.

- Common constraints – that offer “properties” that conforming systems must satisfy (and thus we can and should test). Because the constraints are common, we should also be able to reuse this testing effort (Chakrabarti and Kumar 2009).

In this section, we give some examples of these aspects, but the work in this chapter is focused on modelling the behaviour of collections in REST web services.

3.1 Contributions

In this chapter, we provide three main contributions:

- We describe a reusable FSM model for a reusable component, namely, the CRUD behaviour of a collection component in web services that follow the REST architectural-style.
- We show a technique for adapting the model to different scenarios.
- We present two examples that illustrate how the technique can be applied for adapting the model to two unrelated web services by writing small adaptors.

We also describe how, during our experiments, our models helped us learn about implementation details, previously unknown to us, about the two web services we used as targets.

The ideas described in this chapter are based on the ones presented in the paper (Lamela Seijas, Li and Thompson 2013). I contributed (both to the paper and to this chapter) with the execution of all the experiments and with most of the work on writing; co-authors of the paper Huiqing Li and Simon Thompson contributed with ideas, suggestions, guidance, advice, revisions, and editing.

3.1.1 Software contributions

As part of the experiments carried out during this chapter, I contributed to the `inets` module of the standard library of Erlang by adding support for the method `PATCH`. This contribution can be seen in the pull request¹, which was accepted for inclusion in the `maint` branch.

¹<https://github.com/erlang/otp/pull/917> [last accessed 05-07-17]

3.2 Background

In this section, we describe previously existing work upon which this chapter builds.

3.2.1 Representational State Transfer (REST)

Representational State Transfer (or REST) is an architectural style described by Roy Fielding in his doctoral dissertation (Fielding 2000). REST provides a series of principles for design of distributed systems that aim to encourage a series of properties, the main of which are: performance, scalability, simplicity, modifiability, visibility, portability, and reliability. The principles supported by the REST architectural style reflect the fundamentals that guided the design of the HTTP protocol, but the construction of web services that use HTTP does not guarantee that these will still follow the REST guidelines, thus, those web services that are considered to follow the REST architectural style are considered “RESTful”.

Because REST is a radical perspective, which has been highly influential, the terms “REST” and “RESTful” have become overused buzzwords, and despite the fact that the original principles identified by Fielding in his thesis (Fielding 2000) are clear and unambiguous, there is controversy about what is and is not a RESTful system in practice; see Fielding’s blog post *REST APIs must be hypertext-driven* (Fielding 2008) for an example.

3.2.1.1 REST vs CRUD

CRUD stands for create, read, update, and delete, which are the most common functions of persistent storage (Heller 2007). These functions can, for example, be found in SQL databases (as the statements `CREATE`, `SELECT`, `INSERT`, and `DELETE`), and can be mapped to four common HTTP methods (`POST`, `GET`, `PUT`, and `DELETE`).

REST does not imply nor is limited to CRUD behaviour. REST web services are usually built on top of the HTTP protocol, but even though four of the most common HTTP methods (`POST`, `GET`, `PUT`, and `DELETE`) can straightforwardly be mapped to the four CRUD actions, REST principles are not limited to, nor necessarily require, the usage of these methods.

3.2.2 JSON

JavaScript Object Notation (JSON) is a text-based format for exchanging semi-structured information. It is designed so that it is human readable, simple, and can be easily parsed with JavaScript, since JSON structures can directly be interpreted as a subset of JavaScript's expression language.

Basically, a JSON text is a semi-structured value composed using the 6 data types possible; 4 simple, and 2 composite:

- Simple:
 - Strings: strings of characters are enclosed in double quotes, they may include escape characters, for example: `"this is a string\n"`.
 - Numbers: representation of a numeric value, it may be integer, floating point, or in scientific notation, for example: `-2017`, `2.718`, and `6.022e23`.
 - Boolean: one of the literals `true` or `false`.
 - `null`: is a literal that represents the absence of information.
- Composite:
 - Objects: are dictionaries between strings and JSON values of any type, they are represented by a list of pairs enclosed in curly brackets, for example:

```
{
  "grapes": 12,
  "null": null,
  "string": "this is a string\n"
}
```
 - Arrays: are ordered lists of JSON values separated by commas and enclosed in square brackets, for example:

```
[-2017, "string", { "grapes": 12 }]
```

The different types of JSON can be nested arbitrarily.

JSON is probably, together with XML (see Section 3.2.3), one of the most common ways of representing data among web services. A detailed specification of JSON can be found in (RFC 7159 2014).

3.2.3 XML

Extensible Markup Language (XML) is a text-based format, subset of SGML, for exchanging semi-structured information. It is designed to be straightforward to use over the Internet, multi-purpose, human readable, and easy to read and write by programs (W3 XML 2008).

XML is probably, together with JSON (see Section 3.2.2), one of the most common ways of representing data among web services. In this context, XML is extensively used as part of the Simple Object Access Protocol (SOAP).

3.2.4 JVMTI

JVM-TI (2006) is a standard interface that allows external tools to analyse and control the state of applications that run in a JVM (Java Virtual Machine). This is done through the creation of a dynamic library or JVMTI agent that can be passed as a parameter to the JVM, or by setting the environment variable `_JAVA_OPTIONS`.

A carefully designed JVMTI agent is able to observe the execution of Java code without changing the result of the execution. Because it runs transparently, it has the ability to observe any Java execution seamlessly, regardless of the framework or configuration used to execute them (for example: ant, maven).

JVMTI agents can request to be notified whenever a set of events occur during the execution of a Java program, such as when a method is entered or exited, or when the garbage collector is called. The Java Native Interface (JNI) can be used to call arbitrary Java methods from within the JVMTI callbacks, which allows the use of reflection and meta-calls, and could also be used to alter the behaviour of the target program.

Another advantage of using JVMTI is that it is a standard that may be implemented by different JVMs. This allows our approach to work potentially in different Java environments too.

3.3 Universal interface

In order for software to be testable, it needs to have clearly defined interfaces. When testing RESTful web services we get two advantages inherently:

- RESTful web services are expected to have a well-defined interface.
- The interfaces for RESTful web services are expected to be interoperable. Neither the implementation language of the server and client, nor the operative system in which they run, should be an obstacle in their ability to interoperate.

But because we have a protocol in the middle (in the case of REST web services it is usually HTTP), the size and complexity of the code of web services and, thus, the testing effort needed, are higher than in the case of traditional libraries because:

- We must test the constraints of the protocol supporting access to the services; in our case this is the REST architectural style implemented over HTTP.
- We have to use sockets or other intermediate libraries which add complexity to the system and, consequently, to the tests.
- Web services are usually exposed to a more hostile environment – the Internet – so they need to be tested for safety against malformed or maliciously constructed requests, for example: injection attacks, DoS (Denial of Service) attacks.
- Web services may be accessed from very diverse environments and contexts, because of this, it is often necessary that they support many different configurations and standards (for example: encodings and subtle variations in protocols).

By providing reusable test models we can prevent the user from having to worry about generic low level details like socket usage, HTTP request formats, and even reduce the effort required by protocols like OAuth. Even though this is not covered in this thesis, it would also be possible to create generators that systematically test

for known vulnerabilities like SQL injection or buffer overrun, by testing whether injecting exploits in the different fields causes anomalous behaviour on the SUT (Kieyzun et al. 2009).

3.3.1 REST assumptions

The REST architectural style suggests that resources must be identified by unique URIs, that a RESTful web service must be self-documenting, and that resources must be interconnected through hyperlinks. Theoretically, it should be possible for a machine to automatically discover a web service just by following hyperlinks from its *base URL* (Richardson, Amundsen and Ruby 2013).

In HTTP, different actions on resources are encoded as different HTTP methods, for example: actions related to retrieving a representation of a resource can be achieved through the method `GET` and actions related to deleting the representation of a resource can be achieved through the method `DELETE`. In addition, popular methods have well-established properties (RFC 2616 1999), for example: a `GET` request should not modify the state of the server (from the point of view of the client), and a `PUT` request should be idempotent (the same `PUT` request should only modify the state of the server once, even if it is received several times in a row).

With this kind of constraint we can already imagine it should be possible to create reusable software that will, for example, check for:

- Broken links.
- Responses with status code 500; we know *a priori* that these correspond to unexpected behaviour (RFC 2616 1999).
- Illegal modifications of the visible state of the server (for example: as a result of a `GET` request).
- Idempotency of the relevant methods (like `PUT` and `DELETE`).
- Or even basic expected functionality of methods (for example: is a resource accessible after sending it a `DELETE` request?).

In practice, resource representation and functionality may change slightly from one web service to another, and implementing only the commonalities is not straightforward. Nevertheless, there exists a relatively small number of standard formats that are used by a big part of web services, for example: XML (see Section 3.2.3 on page 31) and JSON (see Section 3.2.2 on page 30); there also exists a number of broadly used design patterns that are used in web services, for example: RSS and Atom (Richardson, Amundsen and Ruby 2013).

3.3.2 Practical difficulties of REST in HTTP

In practice, there are also technical and bureaucratic difficulties that may prevent HTTP web services from being purely “RESTful” and that may push web service designers into workarounds that may be considered to break the REST principles. One clear example of this kind of workaround could be seen in Ruby on Rails, as pointed out by Martin Heller in his post (Heller 2007): “[...] Rails uses JavaScript to generate a dynamic form with a hidden field named `_method`, and sets the value of the field to `delete`. When a Rails application receives a form with a `_method` parameter, it causes the parameter value to override the real HTTP method verb”. Using this work-around, Ruby on Rails avoided the incompatibility with browsers that did not implement the DELETE method at the time.

The usage of a small number of standardised methods for most scenarios is a design decision of REST that is aimed at promoting interoperability between systems (Fielding 2000). In the case of HTTP this is clearly observable from the small number of methods defined by the (RFC 2616 1999) and (RFC 5789 2010).

It is possible to define new methods for new actions (Richardson, Amundsen and Ruby 2013). But, in practice, using custom HTTP methods is impractical: many tools and programs do not support methods that are not standard, this often pushes web service designers to fall-back on the POST (or even GET) method for actions that are not represented by standard methods.

We faced this problem when testing the PATCH method (which is standard but a recent addition to RFC, see RFC 5789 2010): the library we used for our experiments (`inets` from the Erlang OTP distribution) did not support PATCH at the moment of the experiments. In our case, the Erlang OTP team accepted my

contribution to the `inets` library², which solved the problem for us, but we can imagine adding new methods to a tool would be harder to justify in cases where the method is not supported by a standard (ad-hoc methods).

Note that this limitation does not only apply to libraries, but also potentially affects intermediate servers like proxies and firewalls, which would directly limit the usability of the web services affected.

There are at least two “RESTful” ways of approaching this limitation that do not require adding new methods to the standard:

- One is to create resources that represent actions: instead of using a method `PURCHASE` we can create a type or resource that represents a purchase (in the “noun” sense of the word), and the action would be triggered by the creation of this kind of resource. One downside of this solution is that it may require the creation of a large number of small resources for the implementation of trivial actions.
- Another solution is to let actions be triggered by “fields” (in some sense) in the resource representations, (see the example of the `trash` and `deleted` attributes in Section 3.6.1.1 on page 50).

In any case, actions defined in any of these ways will still not be understood by most systems unless their behaviour is described by standardised mime-types or microformats, but at least they will not foreseeably reduce interoperability.

3.4 Systems under test

In order to validate the models we will describe in this chapter, we have used two different and independent web services: Google Tasks and Storage Room. We have chosen these web services for several reasons, some of them are:

- Both provided an API that was described as RESTful by its documentation.
- In both cases, the API was accessible free of charge (subject to some usage restrictions).
- Both were well documented.

²<https://github.com/erlang/otp/pull/917> [last accessed 05-07-17]

- Both clearly implement the collection component in some form.

In this section, we describe both web services briefly.

3.4.1 Storage Room

Storage Room (Storage Room 2010) was a service that offered content management for web and mobile applications. Its web interface allowed users to define and maintain a data model, and data could be added, removed, and modified through the use of a RESTful JSON API that was generated automatically from the data model. The experiments described in this chapter were aimed at testing the RESTful JSON API.

Since the beginning of 2015, Storage Room has been replaced by Contentful (Contentful 2013) and is no longer available (Konietzke 2013).

3.4.2 Google Tasks

Google Tasks³ is a service that allows users to keep track of lists of items and to cross-out items in those lists easily. This system can be used, for example, to keep a to-do list or a shopping list. Google Tasks can be accessed in a number of ways, for example, through the own web application of Google Tasks, through Gmail, through Google Calendar, and directly or indirectly through the API of Google Tasks⁴.

3.5 Collection model

In this section, we describe an FSM model for the collection pattern that we have applied to both Google Tasks and Storage Room (see Section 3.4). The model relies on four of the most broadly used methods defined by the HTTP protocol:

- **GET** – is used to retrieve a resource and should be safe (it should not produce side effects),

³<https://support.google.com/mail/answer/106237> [last accessed 05-07-17]

⁴<https://developers.google.com/google-apps/tasks/> [last accessed 05-07-17]

- **DELETE** – is used to delete a resource and should be idempotent (sending several identical **DELETE** requests should have the same side-effects than sending only one),
- **PUT** – is used to change the contents of a resource (it should be idempotent),
- **POST** – is used to create a new resource in a collection (it does not need to be neither safe nor idempotent).

In addition, for Google Tasks API, we also modelled the method **PATCH**:

- **PATCH** – is used to modify the contents of a resource partially (it does not need to be neither safe nor idempotent).

The **PATCH** method allows us to send only the parts of the resource representation that we want to modify, while **PUT** expects a complete representation.

It is worth mentioning that, because URLs represent arbitrary resources, HTTP methods may have side-effects (in other words: they may affect more resource representations than the one represented by the target URL, and potentially more things than resource representations, for example: posting an order in the web service of an online shop could result in a decrease of the balance of your bank account).

Side-effects are difficult to predict and, in the case of real world side-effects, they are usually undesirable when testing. For these reasons, in this chapter, we will not address these kinds of unrelated side-effects. We will still address those side-effects that are intrinsic to the CRUD behaviour of HTTP methods, like the creation, elimination, and update of resource representations. Future work may extend the models described here to reflect side-effects. Nevertheless, the models that we study in the following chapters will consider local side-effects to some extent.

3.5.1 Locating resources

As we said earlier, it should be possible to discover the functionality of REST web services and the URL of their resources by following hyperlinks from an initial base URL (Richardson, Amundsen and Ruby 2013). Because of this, it should

not be necessary to make any assumptions about the location (the URLs) of the resources in a web service.

Nevertheless, in our implementation we assume, for simplicity, that collections are organised hierarchically, in other words: resources contained within a collection will be represented with a URL that has the URL of the collection as prefix. Thus, our implementation takes two URLs as parameters (where we use “[...]” to mark the places where we have omitted text for clarity):

- The URL of the collection resource, for example:
`http://restsr [...] collections/book_collection`
- A base URL prefix for the entries of the collection (resources contained in the collection), for example:
`http://restsr [...] collections/book_collection/entries`

And we assume that the URL of every entry in the collection will be the result of appending a slash (“/”) and the identifier of the resource to the base URL prefix, for example:

```
http://restsr [...] book_collection/entries/Cinderella
http://restsr [...] book_collection/entries/Hansel_and_Gretel
http://restsr [...] book_collection/entries/Snow_White
...
```

The reason for these assumptions is that it seems that RESTful web services tend to follow them (for example, in addition to Google Tasks and Storage Room, it is also true for at least some parts of the REST API provided by PayPal⁵ and by GitHub⁶) and, while finding hyperlinks in HTML is straightforward, in other formats (like in our case JSON) there is no wide consensus about how to represent them (even though there exist some standards for it they do not seem to be followed as often as our assumptions). Thus, we considered that relying on this mechanism would have made our models less portable in practice.

⁵<https://developer.paypal.com/docs/api/> [last accessed 06-07-17]

⁶<https://developer.github.com/v3/> [last accessed 06-07-17]

3.5.2 Functions of the model

With all these assumptions in mind, we can define a generic model for a collection of resources in a similar way to how we would model a database table: each entry of the collection would correspond to a row in the database table, `SELECT` would be implemented by `GET`, `INSERT` would be implemented by `POST`, `UPDATE` would be implemented by either `PUT` or `PATCH`, and `DELETE` would be implemented by `DELETE`.

In our model, we define six operations, five correspond to the CRUD operations for the elements of the collection, and one for listing the elements of the collection itself. For listing the elements of the collection itself, we use the function `list()`. In practice, `list()` uses the same HTTP method as `get()` (that is: the `GET` method) but targets the URL of the collection instead of the URL of a resource.

In the rest of this section, we illustrate the actions of the proposed model, their signatures, and how they are mapped to HTTP requests. We present them in the form:

```
function(Parameter1, [...], ParameterN) -> Result
```

In these examples, we use books as hypothetical resources and their titles as keys; but, in real systems, keys may be arbitrary hashes or indexes. Both in the examples and in the two systems we tested, the resources are represented using JSON (see Section 3.2.2 on page 30), but they could potentially be represented in any other format instead.

For describing the functions of our model, we use the following format:

```
function_name(Arg1, Arg2, ..., ArgN) -> Result
```

Where `function_name` is the name of the function, the comma separated list in parenthesis represents the arguments that each function takes, and `Result` is an Erlang pattern that demonstrates the structure of the result in normal conditions (no errors are produced). We use syntax similar to the one used in Erlang because we implemented our model in Erlang. We also include ellipsis “[...]” for simplicity, they are not part of Erlang nor JSON but they are a meta notation that indicates that we have omitted part of the text of the example.

The functions of our model are:

- `get(Key) -> {ok, Entry}` – Retrieves the entry with key `Key`. For example:

```

get("Cinderella") ->
  {ok, " {
      [...]
      title: 'Cinderella';
      author: 'Charles Perrault'
      [...]
    } "}

```

The calls to `get/1` are translated into GET HTTP requests to the URL of the entry with key `Key`:

```
GET [...] /book_collection/entries/Cinderella
```

`get/1` returns `{error, not_found}` if the entry does not exist.

- `delete(Key)` -> `ok` – Removes the entry with key `Key`. For example:

```
delete("Cinderella") -> ok
```

The calls to `delete/1` are translated into DELETE HTTP requests to the URL of the entry with key `Key`:

```
DELETE [...] /book_collection/entries/Cinderella
```

`delete/1` returns `{error, not_found}` if the entry does not exist.

- `post(Entry)` -> `Key` – Adds the entry represented `Entry` to the collection and returns the key used to store it. For example:

```

post(" {
  [...]
  title: 'Cinderella';
  author: 'Charles Perrault'
  [...]
} ") -> "Cinderella"

```

The calls to `post/1` are translated into POST HTTP requests to the base URL prefix of the entries of the collection:

```

POST [...] /book_collection/entries
{
  [...]
  title: 'Cinderella';
  author: 'Charles Perrault'
  [...]
}

```

- `put(Key, NewEntry) -> ok` – Replaces the representation of the entry with key `Key` with `NewEntry`. For example:

```

put("Cinderella",
    " {
      [...]
      title: 'Cinderella Man';
      author: 'Ron Howard'
      [...]
    } ") -> ok

```

The calls to `put/2` are translated into `PUT` HTTP requests to the URL of the entry with key `Key`.

```

PUT [...] /book_collection/entries/Cinderella
{
  [...]
  title: 'Cinderella Man';
  author: 'Ron Howard'
  [...]
}

```

`put/2` returns `{error, not_found}` if the entry does not exist.

- `patch(Key, RepPatch) -> ok` – Updates the representation of the entry with key `Key` by following the changes represented in the patch `RepPatch` (in the case of Google Tasks API we used a partial representation of the resource to modify, that is: a representation that only contains the fields that changed). For example:

```

patch("Cinderella",
      " {
        author: 'Brothers Grimm'
      } ") -> ok

```

The calls to `patch/2` are translated into `PATCH` HTTP requests to the URL of the entry with key `Key`.

```

PATCH [...] /book_collection/entries/Cinderella
{
  author: 'Brothers Grimm'
}

```

`patch/2` returns `{error, not_found}` if the entry does not exist.

- `list()` -> `[{Key, Value}]` – Returns a list with all the pairs of keys and entries in the collection, (in our implementation, a list of tuples). For example:

```

list() ->
[ {"Cinderella", "{ title: [...] }"},
  {"Hansel_and_Gretel", "{ title: [...] }"},
  [...],
  {"Snow_White", "{ title: [...] }"} ]

```

The calls to `list/0` are translated into `GET` HTTP requests to the base URL prefix of the entries of the collection:

```

GET [...] /book_collection/entries

```

Example

Since the number of requests we were allowed to issue to the real web services in our experiments was limited, we first implemented a simplified version of our idealised CRUD behaviour to test our model. We now provide a simple example of usage of this implementation in order to illustrate the behaviour described above:

```
1> model:start_link().
{ok,<0.33.0>}
2> HashCinderella = model:post("Cinderella").
"fe3d73ebe0045732f200d90b"
3> model:get(HashCinderella).
{ok,"Cinderella"}
4> model:list().
[{"fe3d73ebe0045732f200d90b", "Cinderella"}]
5> model:delete(HashCinderella).
ok
6> model:delete(HashCinderella).
{error,not_found}
7> model:get(HashCinderella).
{error,not_found}
8> model:list().
[]
9> model:stop().
ok
10> q().
ok
```

Note that the hexadecimal code in this example corresponds to the *Key*, and the string "Cinderella" corresponds to the *Entry*. For clarity, we have used a very short *Entry*, but entries will often be long strings containing the representation of each resource in a format like JSON (see Section 3.2.2 on page 30) or XML (see Section 3.2.3 on page 31).

As we stated earlier, this behaviour is very similar to the one that could be found in a database, because of that, one of our two test models follows a very similar approach to the one presented by (Castro and Arts 2011) for testing databases, but in our case we use a different version of QuickCheck's `eqc_statem` (the grouped version, see Section 2.4.1 on page 21). In the next section, we also refine the model through the introduction of an FSM which allowed us to write a second test model using QuickCheck's `eqc_fsm` module (see Section 2.4.2 on page 23).

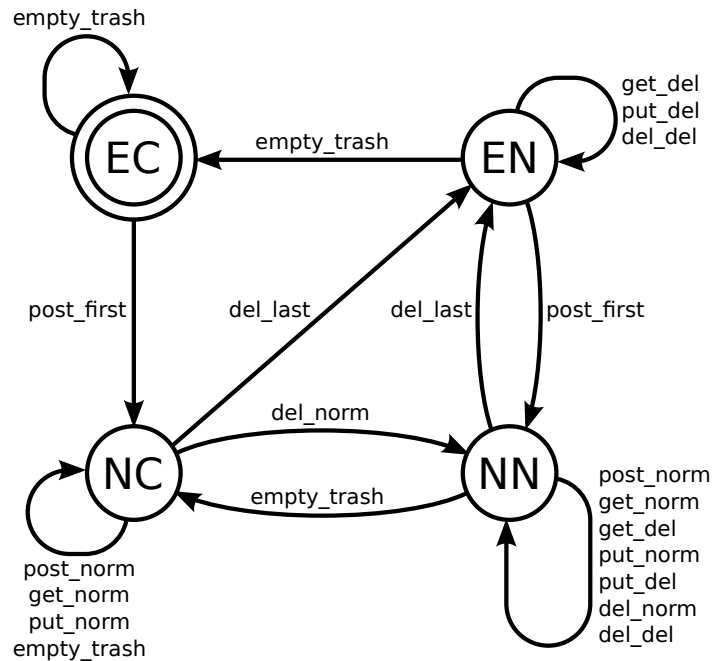


Figure 5: FSM for collection model

3.5.3 Finite-state machine

With the aim of simplifying the preconditions of each command in the model and providing a visual way of checking its correctness, we have developed an abstraction of the methods described in Section 3.5.2 that can be represented in the form of an FSM.

In addition to these advantages, using an FSM allows us to rewrite the model using QuickCheck’s `eqc_fsm` module (see Section 2.4.2 on page 23). Using this behaviour has some advantages of its own: one important example is that it allows QuickCheck to compute weights for transitions automatically, in a way that states of the model have a better coverage when it is used for generating random tests.

Figure 5, shows the graphical representation of the FSM model, the rest of this section describes the meaning of its states and transitions. We have decided to not include the transitions corresponding to the method `PATCH` since, from the point of view of the FSM, both the `PUT` and `PATCH` methods have the same transitions (adding them to the diagram would only have made it more confusing).

The FSM described in this section reflects the existence of a “trash” or “recycling bin” that contains the entries that have been deleted. We discovered this

aspect thanks to failing tests generated using our first version of the model that did not include a trash, (more information on how we used the initial model to discover this behaviour can be found in Section 3.6.1 on page 50).

The FSM has four states (we use the word “normal” to refer to entries that have not been “trashed”):

- EC – Empty & Canonical – No entries are stored (nor trashed, nor normal).
- NC – Non-empty & Canonical – There are normal entries, but trash is empty.
- NN – Non-empty & Non-canonical – Both normal and trashed entries exist.
- EN – Empty & Non-canonical – There are only trashed entries.

The model adds a theoretical method “`empty_trash`” and divides the five HTTP methods into more specific transitions. Some of these additional divisions are necessary in order to keep the FSM model in Figure 5 deterministic, others are just convenient because they simplify the implementation of the model (they separate concerns and not all the functionality for every method is used in every state). For example, if `del_norm` and `del_last` were both `del`, we would have two `del` transitions departing from NC that go to two different states (namely EN and NN):

- GET – Retrieves an existing entry.
 - `get_norm` – Retrieves a normal entry.
 - `get_del` – Retrieves a trashed entry.
- DELETE – Trashes an existing entry.
 - `del_last` – Trashes the last remaining normal entry.
 - `del_norm` – Trashes one normal entry when there are several.
 - `del_del` – Tries to trash an already trashed entry (does nothing).
- POST – Creates a new entry.
 - `post_first` – Creates the first normal entry.
 - `post_norm` – Creates an additional normal entry.

- `PUT` – Replaces an existing entry with another one.
 - `put_norm` – Replaces a normal entry.
 - `put_del` – Replaces a trashed entry.
- `PATCH` (analogous to `PUT`) – Modifies part of an existing entry.
 - `patch_norm` – Modifies a normal entry.
 - `patch_del` – Modifies a trashed entry.
- `empty_trash` – Removes definitely all trashed entries. This command is not an API method neither in Storage Room nor in Google Tasks, but we include it for completion.

Note that the transitions with suffix “`_del`” and the transition “`empty_trash`” are not part of the original model but a result of adapting the model to the behaviour observed both in Storage Room and Google Tasks (see Section 3.6.1 on page 50).

3.5.4 Generators and JSON

In order to test that the implementations comply with the model we described, we need to be able to provide the system with entries. In property-based testing (see Section 1.5 on page 5), inputs are usually created by artefacts called generators. Generators model possible inputs for the system and can be used to automatically generate sets of random inputs. In the case of Storage Room and Google Tasks, the entries must be defined by using JSON (see Section 3.2.2 on page 30). Thus, we need a way to make JSON generators that comply with the constraints of each target web service.

Of course, we could directly write a function that generates the JSON strings that we need in each case, but since JSON is a popular standard, it would be useful to have a general way of generating random JSON (that follows some particular system constraints), for reuse in other systems.

Our solution was to establish a series of keywords, tags or “hooks” within the JSON language so that we can easily transform examples of valid JSON into “JSON templates” that are more general; this idea is inspired by the way JSP is used as a template for HTML, or how PHP is embedded.

This way of inserting “hooks” in JSON documents is foreseeably more readable and easier to maintain than trying to create a hybrid between JSON and another programming language, and, since we do not break the rules of the JSON language, we can reuse existing JSON parsers to interpret documents with “hooks” (we do not need to make our own parser).

The tags or “hooks” are used to specify which kind of generator should be placed where within the JSON template:

- `bool()` – This tag is replaced by either “true” or “false”. It is implemented by the built-in QuickCheck generator `bool/0`.
- `string()` – This tag is replaced with a random string (represented as a binary in Erlang) of visible characters (including spaces). It is implemented as the following QuickCheck generator:

```
?LET(String, list(choose(32, 127)), list_to_binary(String));
```

- `nonempty_string()` – This tag is replaced with a random string (represented as a binary in Erlang) with at least one character and a first character that is printable. It is implemented as the following QuickCheck generator:

```
nonempty_string_gen() ->
  ?LET(String,
        [choose(33, 126)|list(choose(32, 126))],
        list_to_binary(String));
```

we found out that when marking a field as a required string in Storage Room it was necessary for it to have at least one visible character (see Section 3.6.1.3 on page 51).

- `optional()` – If a JSON element `%EL%` is replaced by the following snippet:

```
{ "optional": %EL% }
```

The JSON produced by this generator may or may not contain the element `%EL%`. If the element replaced by this snippet is the value of a JSON object, the key of this value will be removed from the object too (see example below). This mechanism allows us to define optional arguments (see example at the end of this section).

More complicated tags could be defined easily but these were enough for our experiments.

Example

One example of JSON document (representing information about a review on a book) that we could store using a hypothetical web service could look as follows:

```
{
  "entry": {
    "name": "Insensitive comment",
    "liked_it": false,
    "was_read": false,
    "ideas": [
      "boring",
      "ugly",
      "unintersting",
      "disappointing",
      "tedious"
    ],
    "comment": "I don't like the book at all."
  }
}
```

If we make a few changes (by using the hooks described above) we can transform it into a template generator like follows:

```
{
  "entry": {
    "name": "nonempty_string()",
    "liked_it": "bool()",
    "was_read": { "optional()": "bool()" },
    "ideas": [
      "nonempty_string()",
      "nonempty_string()",
      { "optional()": "nonempty_string()" },
    ]
  }
}
```

```
    { "optional()": "nonempty_string()" },
    { "optional()": "nonempty_string()" }
  ],
  "comment": "nonempty_string()",
}
}
```

Note that the key `was_read` is optional, but only the value is marked as such. This is because otherwise we would potentially get JSON whose semantics are defined as “unpredictable” by (RFC 7159 2014), as shown in the next example:

```
{
  "name": "nonempty_string()",
  "optional()": { "liked_it()": "bool()" },
  "optional()": { "was_read()": "bool()" },
}
```

The previous JSON snippet is unpredictable, as described in the (RFC 7159 2014), because it has the same key “`optional()`” several times in one object (the root object), and objects act as dictionaries in JavaScript, so it does not make sense for a single key to have two different values.

3.6 Using the model in practice

The first attempts at modelling a system can easily disagree with the implementation, either because of a bug in the model, because of a bug in the implementation, or because each follows a different interpretation of a higher level specification which is ambiguous. This situation may require refinement of the model, but it could also be deliberately used as a way to adapt a generic model (this can be done automatically to a certain extent), or to learn about the implementation. If we want to adapt our model to different web services, we usually need to account for context-specific aspects like representation formats, location of resources, and internal conventions. In this section, we report on some of the adjustments made and the techniques that we used in order to apply our model to Storage Room and Google Tasks in practice.

3.6.1 Discovering by testing

In our experiments, our preconception of the system turned out to disagree with the actual implementations in a number of aspects. We illustrate some of those aspects in the rest of this section.

3.6.1.1 The existence of a trash

The most fundamental difference we found was that, while executing the command `DELETE` in our original model would result in the definitive elimination of a resource representation, that was not the case in Storage Room nor in Google Tasks. We found this discrepancy when testing Storage Room (which sets an attribute called `trash` in the resource to `true`), but we observed analogous behaviour in Google Tasks (which adds an attribute called `deleted` to tasks when they are deleted).

When executing our initial model against Storage Room, we eventually got the following output:

```
Shrinking.....(11 times)
[{set,{var,5},{call,dbtest,post,
      [{struct, [ENTRY_1]]}],
 {set,{var,7},{call,dbtest,delete,[{var,5}]},
 {set,{var,11},{call,dbtest,get,[{var,5}]}}]

dbtest:post({struct, [ENTRY_1]}) ->
  "5190fa9b0f66027a4900046e"
dbtest:delete("5190fa9b0f66027a4900046e") -> ok
dbtest:get("5190fa9b0f66027a4900046e") ->
  {ok, {struct, [ENTRY_1]}}

Reason: {postcondition, false}
```

Where we have replaced the JSON representation of the entry with the literal “`ENTRY_1`” for clarity. We can see that QuickCheck has found a counter-example and through shrinking found a reduced trace, which in this case is minimal, and produces the error: `post, delete, get`. It is clear that the discrepancy occurs whenever we fetch an entry that we have already deleted (the result that Storage Room produces continues to be a tuple with the atom `ok` and the entry),

which goes against our initial expectations and model. The model described in Section 3.5.3 (on page 44) already reflects the refined behaviour.

3.6.1.2 The representation of PATCH

One particularity of the way Google Tasks API uses PATCH, that we found during our experiments is that it expects partial JSON documents as the body of PATCH requests. In fact, the (RFC 5789 2010) does not specify which format must be used with PATCH, but the (RFC 6902 2013) describes a format (with the media type “application/json-patch+json”) that can be used to describe the differences between two JSON documents (in analogous way to how a patch generated by the `diff` tool would for arbitrary text documents) and it seems designed to be used with PATCH method.

The use of partial JSON as done in Google Tasks API is dependent on the semantics of the particular context, because omitting a part of a JSON dictionary could mean that the entry omitted was omitted to signal deletion or may just be for conciseness. Nevertheless, the format chosen by Google Tasks API is appropriate because it is intuitive; in fact, it was the one we used in our first attempt (our first guess, because the choice is not documented). The fact that our default choice of a format for the PATCH method is not generic is probably an indicator that it may be too early to make any conclusions or generalisations about the use in practice of the method PATCH.

3.6.1.3 Interpretation of mandatory fields

Storage Room allows the definition of a schema for resources, and this allows users to define the expected type of fields and some other constraints. One of these constraints is the optionality of the fields (whether a resource must have a field or not).

When experimenting with these constraints and generators we found out that an empty string, or a string with only spaces, was not valid as the value of fields that were declared as mandatory (non-optional). This motivated the current definition of the generator `nonempty_string()` in Section 3.5.4 (on page 46).

3.6.2 The façade pattern

Each specific web service organises information differently. Even if both our web services use JSON, different systems use it differently. In the case of Storage Room, the structure of resources can be modified and, thus, meta-data is marked with the “@” symbol; whereas in the case of Google Tasks we know beforehand which fields will be returned, so this is not necessary.

In any case, we need an adapter that abstracts out everything but the CRUD collection behaviour we want to test, because we want the model to be independent from the interface that controls the web service. We have achieved this by using the façade pattern (see Section 2.6.2 on page 25) as shown in Figure 6, together with specific generators for each web service.

From the point of view of the user of the façade (for either Storage Room or Google Tasks), the web services behave the same as the mock, thus, the same model can be used to test both. The main job of the façade is to take care of authentication (which is done through a token in Storage Room and through OAuth in Google Tasks), to translate the error messages into a uniform format⁷, and to correctly format the requests (which may require specific formatting and specific headers for the HTTP requests issued to work as expected).

3.7 Chapter conclusion

In this chapter, we have shown how reuse can be applied to models too, in order to make the creation of models easier. In particular, we have shown how a generic model of the collection component can be adapted to work in two different contexts (the web services provided by Storage Room and Google Tasks). What we presented in this chapter is just an example, we can imagine there is potential to create a high-level framework that models a system by joining models of components as a jig-saw puzzle, which would make creating a testing skeleton much easier, but for that to make sense it would be necessary to be able to generalise many different types of components.

⁷The error codes used for each situation vary and in the case of the error “Service Unavailable” returned by Google Tasks, it may be solved just by waiting a moment and resending the request, so we do not really want to return an error directly

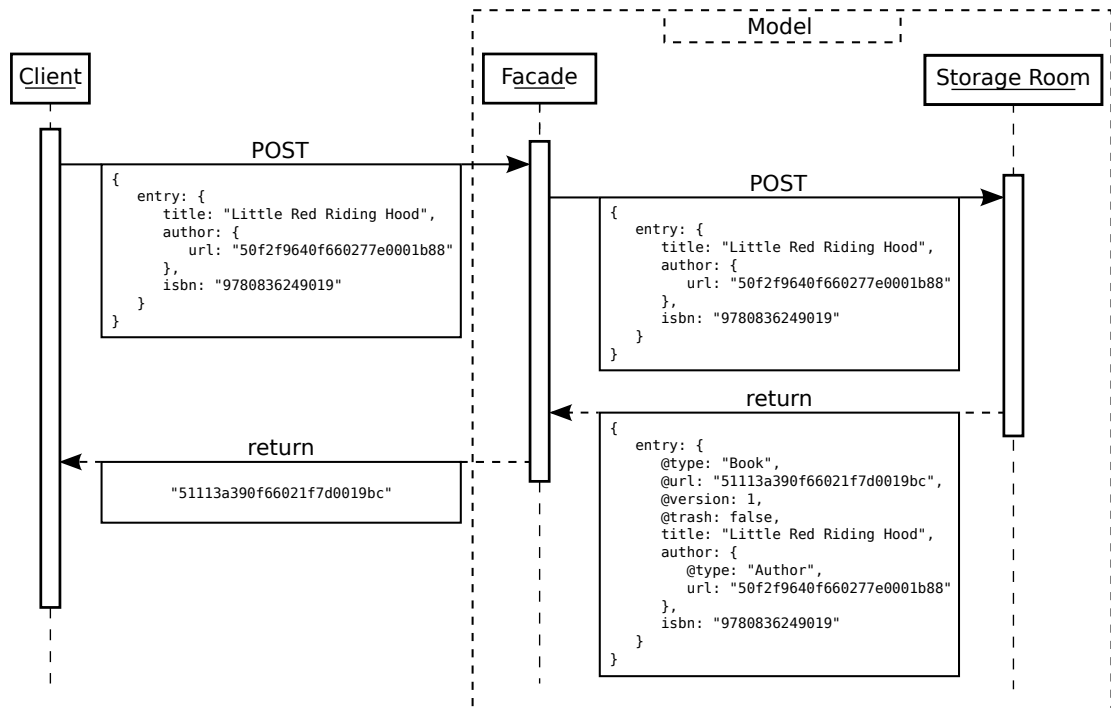


Figure 6: Representation of model and façade over Storage Room

While the creation of reusable meta-models clearly makes the creation of instances of those meta-models easier; the creation of the meta-models themselves is actually harder than the creation of normal models. Can we at least create meta-models automatically from specific models? Making it easier to generalise models would foreseeably also make the evolution of models easier, since generalisation disentangles the common parts from the specific parts, and makes changing the specific parts easier.

In Chapter 4, we try to compare concrete models to locate their general parts, with the aim of automating the creation of general models from specific ones. We will both look at the similarities and differences between the implementations of the models, and at the similarities and differences between the abstract behaviour of the models (expressed as FSMs).

Chapter 4

Differences and generalisation

Generalisation is a broadly adopted mechanism for reusing development effort, an example of this are reusable software libraries, which provide generic functionality that is reused across different projects.

Test models, like software, can also benefit from generalisation, since generalising allows us to reuse testing effort by allowing the same model to test similar systems, similar parts of systems, or similar versions of systems. Additionally, property-based testing can be seen as a generalisation of a set (finite or infinite) of unit tests, since property-based models can be used to generate many unit tests. Thus, in our goal of automating the transition from testing to property-based testing, it seems natural to research the process of generalisation and to try to automate it.

In Chapter 3, we have already explored the application of manual generalisation to reduce the effort required for testing web services. But generalisation requires development effort and does not provide new functionality, thus, generalisation, despite being beneficial, is not without cost.

In this chapter, we study mechanisms for automating the process of generalisation, both looking at the source code (Section 4.3 on page 61) and at the abstract behaviour of implementations through the use of their FSM representation (Section 4.4 on page 103). But first, in Section 4.2, we present some existing work that is necessary to understand the work described in this section.

4.1 Contributions

The main contributions of this chapter are as follows:

- The definition of incremental *interactive* refactorings that aid the introduction and elimination of Erlang behaviours; these are defined in terms of simpler refactorings and transformations, and through the use of Wrangler’s API and DSL (see Section 4.2.3.3 on page 60).
- A methodology for using the new *interactive* refactoring together with already existing refactorings for abstracting existing code by introducing a new Erlang behaviour, and the reverse process that inlines a *behaviour instance* in a *behaviour definition*.
- The definition of a complex *integrated* refactoring that, given a pair of similar modules and a partial mapping that identifies syntactically equal nodes, automatically abstracts their common code by introducing a new Erlang Behaviour; for an explanation of what we mean by syntactically equal nodes see Section 4.3.2.1 (on page 74).
- Validation and comparison of the *interactive* and *integrated* refactorings in two case studies.
- Validation of the design and implementation of Wrangler’s API and DSL.
- Experiments that show the effect of different changes in the configuration of a system to the representation of its behaviour as an FSM.
- Insight on the causes of one type of state explosion in FSMs.

The ideas described in this chapter are based on two papers:

The work described in Section 4.3 has been published in (Lamela Seijas and Thompson 2016a). I contributed (both to the paper and to Section 4.3) with the execution of all the experiments and with most of the work on writing; co-author Simon Thompson contributed with ideas, suggestions, guidance, advice, revisions, and editing. The work in Section 4.3 is built upon the pre-existing work on Wrangler, carried out mainly by Huiqing Li and Simon Thompson.

The work described in Section 4.4 is based on the work presented at Erlang Workshop 2014 (Lamela Seijas et al. 2014). I contributed (both to the paper and

to Section 4.4) with the experiments, with the writing of the description of the experiments, and with insights from the result; co-authors Ramsay Taylor, Kirill Bogdanov, contributed with the design and development of Statechum, PLTSDiff, and Synapse (see Section 4.2.4 on page 60), and helped us understand how to use them, the theory behind them, and the initial set-up; co-authors Simon Thompson and John Derrick contributed with guidance, coordination, evaluation, and advice; and all authors contributed with ideas, suggestions, revisions, and editing.

4.1.1 Software contributions

I have implemented all the refactorings described in this chapter and added them to Wrangler’s API¹; some generic parts of Wrangler and EDoc (Carlsson 2009), and some refactorings from Wrangler, existed previous to this work and I reused them (we explicitly indicate which refactorings existed previous to this work when we describe them later in this section). The input and results of the case studies have also been published in the examples folder of Wrangler. And the main refactorings have been added to the menu [Wrangler > Refactor > Behaviour Refactorings] of Wrangler’s GUI for Emacs.

As part of the development of the refactorings described in this section, I also contributed with small improvements to Wrangler, like the deletion from export declarations of functions removed by transformations done using the DSL and concrete syntax².

4.2 Background

In this section, we describe existing work upon which this chapter builds.

4.2.1 Erlang behaviours

Erlang behaviours are a standard Erlang programming convention which allows the formalization of a design pattern for a process. According to the Erlang documentation³, behaviours are a way of generalising processes: “The idea is to

¹<https://github.com/RefactoringTools/wrangler/pull/69> [last accessed 03-08-17]

²<https://github.com/RefactoringTools/wrangler/pull/69/commits/595496649c118c935a4af302a0b5ac91ec6aa13f> [last accessed 12-09-17]

³http://erlang.org/doc/design_principles/des_princ.html [last accessed 04-10-17]

divide the code for a process in a generic part (a behaviour module) and a specific part (a callback module)”.

In this thesis, we use the terms “behaviour” and “Erlang behaviour” to refer to the mechanism that allows this generalisation. Unlike in the previous definition, we do not restrict ourselves to its application to processes, but we use it as means for parametrisation of arbitrary source code at module level. We also consider that a behaviour is composed of two parts: a *behaviour definition* (often called “behaviour module”) and one or more *behaviour instances* (often called “callback module”).

In this section, we provide partial examples with only the parts specific to behaviours, but a simple complete example of their usage can be found in Figure 15 on page 75, and a real industrial example can be found in the module `gen_server` of the Erlang OTP source distribution.

A **behaviour definition** is an Erlang module that defines a series of callbacks that its *behaviour instances* must implement. This can be done in one of two ways:

- By adding `callback` declarations to the *behaviour definition*. A `callback` declaration is a directive that describes both the name of an expected callback and its type signature, for example:

```
-callback terminate(
    Reason :: (normal | shutdown | {shutdown, term()} |
              term()),
    State :: term()) -> term().
```

- By implementing the `behaviour_info/1` function in the *behaviour definition*. This function, when passed the atom `callbacks` as parameter, must return a list of tuples with the name and arity of the expected callbacks. For example:

```
behaviour_info(callbacks) ->
    [{init, 1}, {handle_call, 3}, {handle_cast, 2},
     {handle_info, 2}, {terminate, 2}, {code_change, 3}];
behaviour_info(_) -> undefined.
```

The work in this thesis only considers the variant that uses `behaviour_info/1` function because, at the time of writing, there is limited support for `callback` declarations by the abstract syntax tools from the standard Erlang distribution, in particular by the `erl_prettypr` module.

A **behaviour instance** is an Erlang module that provides a concrete implementation for the callbacks defined by the *behaviour definition*. *Behaviour instances* should contain among their headers a declaration that specifies which behaviour definitions they implement, in the form:

```
-behaviour(gen_server).
```

where `gen_server` is the name of the module containing the *behaviour definition*.

There is nothing preventing a module from being instance of more than one behaviour definition but, at the time of writing, the compiler shows a warning if a behaviour instance implements behaviour definitions that require callbacks with the same name and arity.

4.2.2 ets and dets

`ets` and `dets` are Erlang built-in term storage APIs (`ets` 1997; `dets` 1997). They both have a mostly similar interface and provide almost constant access time to the data. The main difference between `ets` and `dets` is that `ets` stores data in RAM memory while `dets` allows storage in files.

We have used these APIs for the case study in the Chapter 4 because:

- `dets` is a substantial piece of open source Erlang software, in use in a number of production systems built using Erlang
- Previous to this work, there already existed a QuickCheck model for `dets`, shipped as part of the QuickCheck distribution. The model provides automatic validation of the result of the refactorings we carry out as part of the pilot study.
- Some parts of `dets` can be compared to `ets`, since their interfaces are similar.

As part of this thesis, we have adapted the existing QuickCheck `dets` model for the `ets` system, in order to have two similar models to compare.

4.2.3 Wrangler

The Wrangler tool for Erlang (Li and Thompson 2006; Li et al. 2008) has three parts: a set of built-in refactorings, a set of inspection tools, and a set of extension mechanisms. In this thesis, we use and extend Wrangler’s refactoring capabilities. Refactoring is the process of changing the structure of a program without changing what it does (Thompson and Li 2013).

Source code transformations (which from now on we will call just *transformations*), as opposed to refactorings, may change the behaviour of a program; but several transformations can be combined to form a refactoring.

Wrangler also provides a GUI for both Emacs editor and Eclipse. The refactorings implemented as part of this thesis are both accessible from Emacs and from the programmatic API. In this thesis, we describe the input of the refactorings as seen from the Emacs GUI.

4.2.3.1 Built-in refactorings

Wrangler includes a series of built-in refactorings (Thompson and Li 2013) for Erlang programs. These include refactorings for structure, such as renaming, function extraction, and function generalisation; refactorings for concurrency and parallelism; and refactorings supporting unit testing and property-based testing.

4.2.3.2 Inspection tools

In order to help users identify the parts of systems that require refactoring, Wrangler includes a set of inspection tools. These range from “local” scale symptoms to larger scale properties such as the existence of duplicate code or clone detection (Li and Thompson 2009) and module structure problems like inter-module cyclic dependencies.

4.2.3.3 Extension mechanisms

Wrangler also provides mechanisms for adapting to specific situations and for extending itself through two extension mechanisms: an API for user defined refactorings, and a domain-specific language.

API for user-defined refactorings The API for user-defined refactorings (Li and Thompson 2011) allows new refactorings to be written from scratch without using Erlang’s abstract representation. This is possible thanks to the inclusion of a “template” language that allows transformations to be written using the concrete syntax of Erlang.

Domain-specific language Another approach to creating new refactorings is to use Wrangler’s domain-specific language (DSL). Wrangler’s DSL gives users the ability to define complex refactorings by combining simpler ones, to specify the user interaction required, to define transactional behaviour, and to describe classes of refactorings with a single statement (for example: rename all functions named `likeThis` to functions named `like_this`).

4.2.4 Statechum

Statechum is a framework written in Java that implements a number of regular grammar inference algorithms and allows users to visualise, analyse, and compare state machines, as well as reverse-engineer them from examples (Bogdanov, Walkinshaw and Taylor 2007). Statechum allows both active and passive inference, it can take as input a set of traces (examples), interactively issue queries and generate a diagram based on the answers, or use LTL constraints to answer queries automatically. The output is usually a state machine that is displayed graphically by the tool and can be rearranged interactively by using mouse and keyboard.

4.2.4.1 PLTSDiff

Statechum implements (among others) the algorithm PLTSDiff (as described in Bogdanov and Walkinshaw 2009), that allows users to create a “diff” state-machine from two similar ones. The algorithm tries to find a maximal mapping between both state-machines and highlights the nodes and transitions that have to be added or deleted to go from one of the input state-machines to the other.

4.2.4.2 Synapse

Synapse provides an interface for using FSM and EFSM learning tools from Erlang (Taylor 2013). The current version of Synapse allows:

- The inference of state machines from traces. Given a valid set of positive and negative traces, Synapse produces a state machine that accepts and rejects them respectively.
- The inference of a “diff” state machine from two state machines. Given two state machines produced by Synapse, Synapse produces a “diff” state machine that shows the differences between them.
- The visualisation of both state machines and “diff” state machines. Given a state machine or “diff” state machine produced by Synapse, it can be visualised by using its own API.
- The automatic reverse engineering of Erlang modules. The combination of Synapse and Statechum allows users to use Erlang modules as oracles, that way the user does not need to provide any examples or traces; the values to try are inferred by using the type specifications provided by running Typer⁴ on the module provided.

In Chapter 4, we use Synapse to apply Statechum algorithms to different configuration of the Erlang’s Frequency server implementation (see Section 2.1.5 on page 15).

4.3 Source parametrisation

In addition to promoting effort saving through reuse, generalisation helps reduce code replication within software implementations. While some code replication might have a sound reason for its existence (Cordy 2003; Kapsner and Godfrey 2006), in general, it is detrimental in several ways: it increases the compilation time, it increases the size of the source code and executables and, more critically, it increases the cost of maintenance and the probability of bugs being propagated during maintenance (Monden et al. 2002).

⁴<http://erlang.org/doc/man/typer.html> [last accessed 21-09-17]

As stated by (Pierce 2002) “Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the parts that vary”.

In this section, we explore the automation of the generalisation of source code through the introduction of Erlang behaviours (see Section 4.2.1 on page 56). In particular, we study two alternative approaches: an *interactive* set of refactorings and a fully automated (*integrated* version), and we compare the advantages and disadvantages of both approaches.

We have chosen Erlang for our experiments since it is often used for implementing state machine models of systems under test. In many application areas, system variants are prevalent, and so multiple models are developed, which differ in some aspects of behaviour but share overall structure as well as most aspects of behaviour. In such situations, it is desirable to develop parametric models, and the work reported here supports building parametric models from two existing concrete variants.

Even though the techniques described in this section have been implemented in Erlang, they could be easily adapted to other languages, since they rely on artefacts that are not exclusive to Erlang. For example, in the case of Java, interfaces or superclasses could be used instead of Erlang behaviours. In fact, the interactive approach reuses refactorings that are already available in other mainstream languages. However, the integrated approach, as described here, relies on the target language being dynamically typed.

4.3.1 Interactive refactoring

In this section, we present the interactive approach to code generalisation. In particular, we present a series of refactorings and transformations that automate the task of creating and inlining Erlang *behaviour instances* (see Sections 4.3.1.2 and 4.3.1.3).

These refactorings were implemented in terms of simpler refactorings and transformations (some existing and some new) that were composed using Wrangler’s DSL for composite refactorings (see Section 4.2.3.3 on page 60). For this

reason, we describe the “main” refactorings by first explaining the “simpler” refactorings and transformations, and then explaining how they are used from the “main” refactorings.

The “simpler” refactorings and transformations were implemented mainly using Wrangler’s callback interface for user-defined refactorings (see Section 4.2.3.3 on page 60), but some of the “simpler” refactorings used by the inlining refactoring were implemented as new internal extensions to Wrangler.

The main functionality described in this section can be summarised as follows: extraction of a single Erlang *behaviour instance* from either an existing function or an expression, and unfolding or inlining of a single *behaviour instance* against its *behaviour definition*.

In Section 4.3.1.1 we cover the elementary refactorings and transformations used through the rest of the section. We cover the actual implementation of extraction and inlining in Sections 4.3.1.2 and 4.3.1.3 respectively. Finally, Section 4.3.1.4 presents an example of both extraction and inlining using iterative refactorings.

4.3.1.1 Basic refactorings and transformations

Interactive refactorings are implemented using Wrangler’s DSL, which allows the construction of composite refactorings in terms of smaller ones. In this section, we provide an overview of the primitive refactorings and transformations that are used as basis for the construction of the refactorings described in Sections 4.3.1.2 and 4.3.1.3:

- *Copy module*: This is a new refactoring based on the existing refactoring *Rename module* and, as such, it was implemented as a new primitive refactoring. It creates a renamed copy of a module. Additionally, it can take a list of potentially dependent modules as input. If a list is provided, the refactoring updates the references of the modules in the list to point to the new copy of the module, the rest of modules remain unchanged (pointing to the original module).
- *Create behaviour instance file*: This is a new transformation implemented through Wrangler’s API for user-defined refactorings. It takes one module name for the new *behaviour instance* and the name of an existing module

to be used as *behaviour definition* (the last one is assumed to be the current module when using the transformation from the GUI).

- it creates the module if it does not exist,
 - it adds a *behaviour declaration* to the new module that sets the second module provided as input as *behaviour definition*.
- *Add function to behaviour_info/1*: This functionality is implemented as two new alternative user-defined refactorings:
 - *Add function to behaviour_info* – takes a function name and its arity and adds it to `behaviour_info/1` as a single tuple of the form `{FunctionName, Arity}`.
 - *Add function name to behaviour_info* – takes a function name and adds a tuple to `behaviour_info/1` for each of the functions in the module that have that name (in other words: independently of their arity).

Both refactorings create the `behaviour_info/1` function if it does not exist and add it to the `export` declaration.

- *Remove behaviour declaration*: This is a new transformation implemented through Wrangler’s API for user-defined refactoring. It removes the implementation of the `behaviour_info/1` function and deletes its entry from the `export` declaration.
- *Instantiate calls*: This is a new transformation implemented through Wrangler’s API for user-defined refactoring. It searches for calls with dynamic module qualifier that target the functions enumerated in `behaviour_info/1`, and it modifies those calls so that they point statically to a target module given as a parameter to the refactoring.
- *Move Function to Another Module*: This refactoring was already part of Wrangler. It takes a function and moves it and its dependencies (in the current module) to a different module while ensuring that all the references affected are updated to the new location. It will also add the functions to the `export` declaration if necessary.

- *Function Extraction*: This refactoring was already part of Wrangler. It takes an expression or sequence of expressions and moves them to become the body of a new function. A call to the new function is inserted in the place where the expressions were originally.

4.3.1.2 Behaviour extraction

The automation of behaviour extraction is provided through two alternative “main” refactorings: *Expression to behaviour instance* and *Function to behaviour instance*. The difference between them is that *Expression to behaviour instance* takes an expression and applies *Function Extraction* to it, while *Function to behaviour instance* takes a function directly.

In both cases, the functionality selected is abstracted as a new *behaviour callback* and its implementation is moved to the target module (that is: the module provided as argument to the refactoring) that becomes a *behaviour instance* of the current module (the original module of the function).

In addition to the expression or function to abstract, both refactorings take as input a target module name (for use as *behaviour instance*) and, in the case of *Expression to behaviour instance*, a name for the new function that will be created.

Internally, both are composite refactorings implemented using the Wrangler DSL and they call the following individual refactorings and transformations (that we already described in Section 4.3.1.1) in order:

1. *Create behaviour instance file* – creates the *behaviour instance* (that is: the target module) if it does not exist and adds the *behaviour declaration* if it does not have it.
2. *Function Extraction* (only for *Expression to behaviour instance*) – moves the selected expression to a new function in the same module.
3. *Add function to behaviour_info* – adds the function (either provided by the user or extracted in the previous step) to the list of callbacks to implement by *behaviour instances* (that is: the list defined by `behaviour_info/1` in the *behaviour declaration*).

4. *Move Function to Another Module* – moves the implementation of the function to the *behaviour instance*.

After calling either of these refactorings, a new *behaviour instance* has been created (if it did not already exist) and the current module has become a *behaviour definition*. These refactorings can be executed multiple times, using other functions and expressions as target, in order to add a series of functions as callbacks to the *behaviour*.

But there is still one desirable thing to do: the calls of the behaviours generated by previous refactorings still need to be generalised somehow, for example: a call at this point still looks like `server:start()` where `server` is the name of a concrete *behaviour instance*. If we want the new behaviours to use several instances, we must be able to choose which behaviour instance to use at a given time, for example: we may want calls to be of the form `Instance:start()`, where `Instance` is a variable that contains the name of the *behaviour instance* to use. This can be fixed in several ways (see *Adjustments for generalisation* in Section 4.3.1.4).

We decided not to automate the generalisation of calls since there are multiple possible alternatives whose suitability depends on the context.

4.3.1.3 Behaviour inlining

As part of this work, we have also developed a refactoring for carrying out the reverse process: behaviour inlining or unfolding. This refactoring combines the specific code in a *behaviour instance* with a copy of the generic code in its *behaviour definition*. A copy is made in order to ensure that other potential *behaviour instances* keep working.

The behaviour inlining process is supported by the composite refactoring *Unfold behaviour instance*. This refactoring takes a *behaviour instance* and the name of the output module to contain the inlined instance.

Internally, *Unfold behaviour instance* was implemented using the Wrangler DSL and calls the following individual refactorings and transformations in order:

1. *Copy module* – copies the *behaviour definition* to the output module updating only the references from the selected *behaviour instance*.

```
-module(operation).  
-export([get_op_result/2]).  
operation(A, B) -> A + B.  
get_op_result(A, B) -> {ok, operation(A, B)}.
```

Figure 7: Initial code

2. *Instantiate calls* (optional step) – finds the dynamic calls that have function names matching the ones in the list of callbacks (that is: the list defined by `behaviour_info/1`) and instantiates them to point statically to the *behaviour instance*.
3. *Move function* – moves the functions from the *behaviour instance* to the new copy of the *behaviour definition* (the output module).
4. *Remove behaviour declaration* – removes the `behaviour_info/1` function from the new copy of the *behaviour definition* (the output module).

We already mentioned that there are several ways of generalising function calls so that they can point to any *behaviour instance*. Step 2 (*Instantiate calls*) aims to reverse the process of generalising function calls, but it is approximate; it instantiates dynamic calls that have the same names than the behaviour callbacks, but there is nothing preventing a dynamic call from having the same name as a behaviour callback, being dynamic, and still not being a behaviour callback. If this was ever the case, our refactoring would produce incorrect results.

On the other hand, in some cases, step 2 will not be necessary at all, depending on which mechanism was used for generalisation. For both of these reasons, the step is left as optional.

4.3.1.4 Example

In this section, we show, through an example, how to apply the refactorings and transformations described in Sections 4.3.1.2 and 4.3.1.3 to the creation and inlining of Erlang behaviours.

We start the process with a module called `operation`, shown in Figure 7.

<pre> -module(operation). -export([get_op_result/2, behaviour_info/1]). behaviour_info(callbacks) -> [{operation, 2}]; behaviour_info(_Other) -> undefined. get_op_result(A, B) -> {ok, sum:operation(A,B)}. </pre>	<pre> -module(sum). -behaviour(operation). -export([operation/2]). operation(A, B) -> A + B. </pre>
---	---

Figure 8: Quasi-behaviour definition and instance after extraction

Creating a behaviour from scratch. In our example, the code that we want to place in a behaviour instance is already in a separate function, namely `operation/2`. Because of this, we selected `operation/2` and applied *Function to behaviour instance* refactoring.

If, instead, the function `get_op_result/2` was defined as follows:

```
get_op_result(A, B) -> {ok, A + B}.
```

We could achieve the same result by selecting `A + B`, applying the refactoring *Expression to behaviour instance*, and answering “`operation`” when asked about the name for the new callback.

In any case, the refactoring will ask us for the name of the “Destination module” which in our case we called `sum`.

The refactoring will make a *behaviour definition* (see Section 4.2.1 on page 56) out of the module `operation` by adding a `behaviour_info/1` function with the tuple `{operation, 2}` in it. A new module `sum` will be created to implement the `operation` behaviour with the function or expression selected as the callback `operation/2`. The result of the refactoring is shown in Figure 8.

Adjustments for generalisation. Arguably, the “behaviour” created up to now may not be actually considered a “proper” behaviour since the call inside `get_op_result/2` to function `operation/2` has the module `sum` hard-coded, which makes it not reusable (whereas behaviours are supposed to be reusable). We cannot use the generated *behaviour definition* together with new *behaviour instances*, for example, a module `division`: we would not be able to make `get_op_result/2` use the callback `operation/2` of that module.

<pre>-module(operation). -export([behaviour_info/1]). -export([get_op_result/3]). behaviour_info(callbacks) -> [{operation, 2}]; behaviour_info(_Other) -> undefined. get_op_result(A, B, Op) -> {ok, Op:operation(A, B)}.</pre>	<pre>-module(sum). -behaviour(operation). -export([operation/2]). -export([get_op_result/2]). operation(A, B) -> A + B. get_op_result(A,B) -> operation:get_op_result(A,B,sum).</pre>
---	---

Figure 9: Behaviour definition and instance after adjustments

One way of addressing this limitation starts by applying *Generalise Function Definition* refactoring to the atom `sum` in `get_op_result/2`, this will create a new function `get_op_result/3`, which takes the target *behaviour instance* module name as a parameter. In order to make the *behaviour definition* completely generic, it is usually desirable to also apply *Move Function to Another Module* refactoring twice: once to move `get_op_result/2` to module `sum`, and once to bring `get_op_result/3` back to module `operation`. This procedure will leave us with the generalised version shown in Figure 9.

Note that, by doing adjustments in this way, we have moved part of the concrete interface, that was originally available in the module `operation`, to the instance `sum`. The behaviour is preserved because the other places in the code of the hypothetical project, where the original function `operation:get_op_result/2` was used, will now point to `sum:get_op_result/2`. In module `operation`, we now have a generic version `get_op_result/3`, that takes the name of the specific instance to use as an extra parameter.

Alternative adjustments for generalisation. Adding a parameter is not the only way of generalising *behaviour instance* calls.

If the behaviour to use can be decided at compilation time, we may generalise it through the use of a macro (see Figure 10). If the behaviour represents a server, like is the case of `gen_server`, the name of the *behaviour instance* module can be stored in the state of the server or in the process dictionary (see Figure 11).

Both alternatives avoid the need to introduce a new parameter. In exchange,

```
-module(operation).  
  
-define(INSTANCE, sum).  
  
-export([get_op_result/2,behaviour_info/1]).  
  
behaviour_info(callbacks) -> [{operation, 2}];  
behaviour_info(_Other) -> undefined.  
  
get_op_result(A, B) -> {ok, ?INSTANCE:operation(A, B)}.
```

Figure 10: Alternative adjustments for generalisation 1

```
-module(operation).  
  
-export([get_op_result/2,behaviour_info/1]).  
  
-export([set_instance/1]).  
  
behaviour_info(callbacks) -> [{operation, 2}];  
behaviour_info(_Other) -> undefined.  
  
set_instance(Instance) -> put(instance_name, Instance).  
  
get_op_result(A, B) -> {ok, (get(instance_name)):operation(A,B)}.
```

Figure 11: Alternative adjustments for generalisation 2

```

-module(opsum).

-export([get_op_result/3]).
-export([get_op_result/2]).
-export([operation/2]).

get_op_result(A,B,Op) -> {ok, operation(A, B)}.

get_op_result(A,B) -> get_op_result(A,B,sum).

operation(A, B) -> A + B.

```

Figure 12: Result of inlining

the first approach requires the decision to be taken at compilation time (which is not always possible), and the second approach takes advantage of side effects (which may lead to an increased difficulty in error debugging).

These scenarios illustrate the fact that there is no one “correct” workflow for behaviour introduction. What we have done in this chapter is to provide a minimal workflow: new functionality can be created easily through Wrangler’s extension mechanisms (see Section 4.2.3.3 on page 59); in particular, through Wrangler’s DSL, existing refactorings can be combined in order to create new functionalities that are appropriate for each context (as the *interactive* refactorings themselves demonstrate).

Inlining a behaviour instance. We can revert the process of behaviour introduction by calling the refactoring *Unfold behaviour instance* from a *behaviour instance*. The refactoring prompts the user for a name to give to the output module (in our example we inline the module `sum` from Figure 9 and use the non-existent module `opsum` as output). We are then asked about whether we want to point all dynamically qualified callbacks to `opsum` (in our case we answered *yes*, in order to revert the adjustments for generalisation). The result of executing the refactoring was the module `opsum` shown in Figure 12.

Removing spurious parameters. We can see that both `get_op_result/2` and `get_op_result/3` in Figure 12 behave as the original code, but the function `get_op_result/3` has a spurious parameter (that is: `Op`). If we compile `opsum` we will get a warning for this reason.

```
-module(opsum).  
  
-export([get_op_result/2, operation/2]).  
  
get_op_result(A,B) -> {ok, operation(A, B)}.  
  
operation(A, B) -> A + B.
```

Figure 13: Result of removing spurious parameters

In order to obtain cleaner code, we can apply Wrangler’s *Unfold function application* refactoring to inline the application of `get_op_result/3` into its only usage in `get_op_result/2` and then we can remove `get_op_result/3` (together with its export declaration).

By doing this we will obtain the code shown in Figure 13, which is almost equivalent to the original one from Figure 7 (on page 67).

4.3.2 Integrated refactoring

The integrated refactoring assumes a slightly different scenario: one where we already have replication and we want to introduce abstraction to eliminate it. At this point, we must make an additional effort to find the commonalities between the two pieces and unify them, which can be a hard and error prone process (Tsantalis, Mazinanian and Krishnan 2015).

By having a mechanism that allows us to abstract the commonalities between two modules automatically, in addition to reducing the likelihood of introducing mistakes when we find ourselves in this situation, we allow for an easier approach for creating abstractions that exploit existing heuristics; in particular: the heuristic that tells us that it is easier to work with examples than with abstractions (Holmes and Langford 1976), and the heuristic that tells us that it is easier to modify than to create from scratch (Miller 1991). For example, we can create one example, make a copy of it, modify the copy, and then use an automated refactoring to abstract out the replication.

In this section, we therefore study a mechanism that we called the *integrated approach*, that works precisely in this way. It takes two similar modules and a partial mapping. The partial mapping must identify pairs of nodes, one in each AST, which are syntactically equal; we define what we mean by syntactically

equal nodes in Section 4.3.2.1. Given these two similar modules and partial mapping, our algorithm abstracts commonalities automatically by using the template pattern (see Section 2.6.1 on page 25) implemented as an Erlang Behaviour (see Section 4.2.1 on page 56). The algorithm is completely automated, it does not require any human intervention.

This way, the commonalities (given by the mapping) are moved to the *behaviour definition*, while the specific parts of the two modules (that is: parts that have no correspondence in the other module) remain in their original modules, and the two original modules become *behaviour instances* that implement the newly created *behaviour definition*.

Note that the approach is the opposite to that of the interactive approach in that we move away the generic parts instead of the specific ones (in any case the result is having the generic parts separated from the specific ones). The integrated approach also differs from the interactive approach in that, while the latter is implemented in terms of simpler refactorings, the former is implemented at a low level. It would probably be possible to express some of the aspects of the integrated refactoring in terms of simpler ones, but we chose to use a more “holistic” approach because the integrated refactoring affects the whole structure of the module and makes use of several complex and ad-hoc data structures for the combined processing of the different parts.

In addition to the general objective of abstracting out generic parts, the refactoring satisfies several properties (assuming there are no bugs in the implementation or errors in the design):

- The *behaviour instances* generated must comply with a common interface (in other words: both implement the same number of callbacks, and the callbacks must have the same name and number of arguments in both instances)
- The external interface and the behaviour of the original modules, as observed from that interface, must remain the same after the refactoring (other external modules that call the original modules should not be able to notice the refactoring).
- No conditional statements must be introduced. Control flow must be preserved through the insertion of functions and function calls.

<pre>-module(sum). -export([sum/2]). sum(A, B) -> {ok, A + B}.</pre>	<pre>-module(division). -export([division/2]). division(_A, 0) -> error; division(A, B) -> {ok, A / B}.</pre>
--	--

Figure 14: Example input for the integrated refactoring

For example, if we run the integrated refactoring on the modules from Figure 14 with a mapping that maps the atom `ok` and the tuple in one version to the atom `ok` and the tuple in the other, we obtain the modules in Figure 15. The example in Figure 15 was actually generated automatically from the modules in Figure 14 by our implementation of the integrated refactoring available as part of Wrangler (see Section 4.2.3 on page 59).

The algorithm behind the refactoring takes two similar modules and a tree mapping (which we describe in more detail in Section 4.3.2.1) and has two main phases: *cluster construction*, that groups the contiguous commonalities and moves them to the new *behaviour definition* module (Section 4.3.2.2), and *cluster linking*, that restores the control flow by linking the pieces together with function calls in a way that the original behaviour is preserved (Section 4.3.2.3).

4.3.2.1 Tree mapping

The algorithm takes as input two modules and one partial mapping that identifies syntactically equal nodes in their two ASTs. The work in this thesis does not study how to find a good partial mapping between the ASTs of the input modules. A good option would be to use a mapping derived from the minimal edit script including “move” operations (“move” operations and untouched nodes can be translated to mappings between the two ASTs, and additions and deletions can be left unmapped); but this has been claimed to be an NP-complete problem (Falleri et al. 2014).

For our experiments, examples, and for our implementation distributed as part of Wrangler, we use the tree matching algorithm from Al-Ekram, Adma and Baysal (2005) and, while this algorithm and our refactoring combined provide a useful tool already (we validate its usability in Section 4.3.4 on page 91), it would be

<pre> -module(sum). -export([sum/2]). -export([callback_1/2]). -behaviour(operation). sum(A, B) -> operation:common_1(?MODULE, A, B). callback_1(A, B) -> A + B. </pre>	<pre> -module(division). -export([division/2]). -export([callback_1/2]). -behaviour(operation). division(_A, 0) -> error; division(A, B) -> operation:common_1(?MODULE, A, B). callback_1(A, B) -> A / B. </pre>
<pre> -module(operation). -export([behaviour_info/1, common_1/3]). behaviour_info(callbacks) -> [{callback_1, 2}]; behaviour_info(_Other) -> undefined. common_1(Module, A, B) -> {ok, Module:callback_1(A, B)}. </pre>	

Figure 15: Result of applying the integrated refactoring to Figure 14

possible to choose and develop other mechanisms for finding a mapping that may be more appropriate in certain circumstances. In particular, we have found some more recent and heavily tuned algorithms that might have been more appropriate for our approach (Falleri et al. (2014); Fluri et al. (2007)), but we have not used them because, due to their complexity, reimplementing them would have been much more costly, and in the cases where there exist public implementations of them, they are written in different languages, which would have made integrating these implementations more difficult and cumbersome.

Syntactically equal AST nodes Independently of the mechanism we use to obtain a mapping, we must define what we mean by two AST nodes being syntactically equal.

We use three simple criteria:

- If we compare a pair of *leaf nodes*, we just compare the literal representation of the nodes after removing irrelevant information like comments and layout. This way, two variables or two atoms are considered syntactically equal if and only if their names match.
- If we compare a pair of *parent nodes*, we simply check that they have the same type and the same number of children. This way, two list constructors of the same length are considered syntactically equal independently of their elements, but a tuple and a list of the same length, or two lists of different lengths, are not.
- If we compare a *leaf node* and a *parent node* we can directly conclude that they are not syntactically equal.

We can represent the mapping as a set of arrows between both trees. In Figure 17a (on page 78) we show what a possible tree mapping representation would look like, where nodes with the same name are considered syntactically equal.

Another consideration about the mapping used as input is that, even though it is not a precondition of the integrated refactoring, results benefit considerably from the mapping supplied being as close to an isomorphism as possible (in other words: the relative position of a node in one tree should be similar to the position of its image in the other tree and there should be as few discontinuities in the

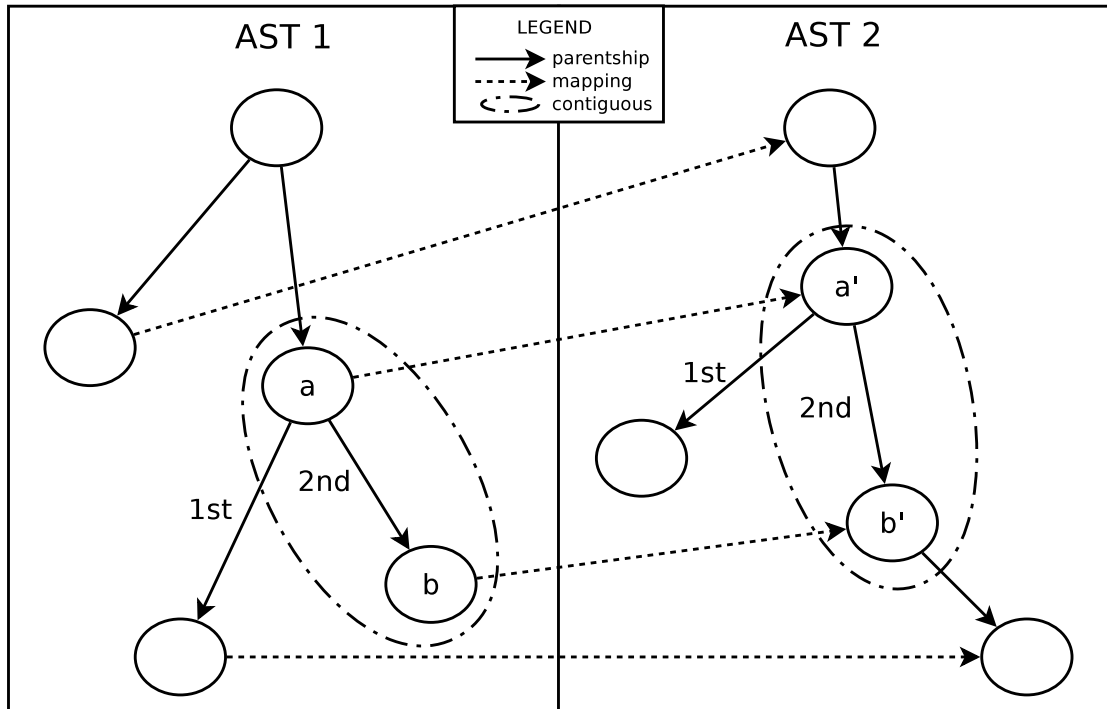


Figure 16: Example of pair of nodes with contiguous mapping

mapping as possible), the more discontinuities in the mapping, the higher the number of callbacks it will be necessary to introduce.

4.3.2.2 Cluster construction

The first step is to create groups of subtrees or clusters that we can use undivided. We do this in two stages: first we try to create maximal common clusters, and then we iteratively reduce them so that frontiers between clusters can be represented as terms or Erlang function calls, (in other words: so that clusters can be extracted as functions).

Creating common clusters. The tree mapping relates the common (syntactically equal) nodes of the ASTs of both input modules. In order to find common subtrees, we only have to traverse one of the ASTs and group all those nodes that have a contiguous mapping.

Given a pair of nodes a and b from the same tree where b is the n^{th} child of a , we say a and b are contiguous if and only if: the parent of the projection of b

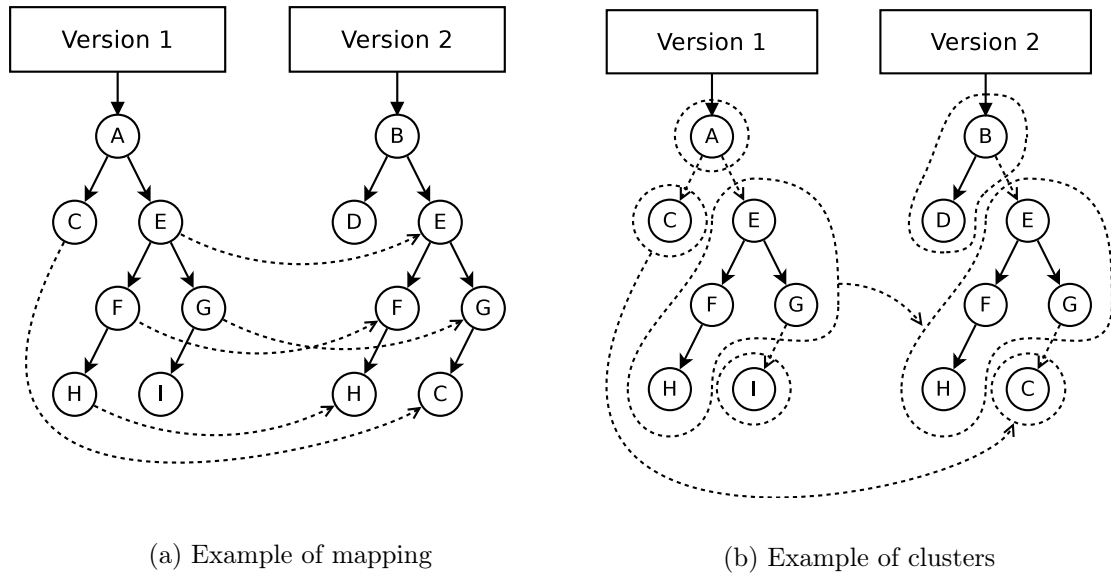


Figure 17: Tree matching and cluster construction

is identical to the projection of a and the projection of b is the n^{th} child of the projection of a , or if both a and b have no projection. In Figure 16 we show an example to illustrate the concept of nodes with contiguous mapping.

We call the relationship between a parent node and a child node in the same tree a *link*. After defining contiguous mapping we can define a *frontier link* to be any link between two nodes of which, at least one has a mapping, but which are not contiguous in term of their mappings. *Frontier links* are important because they define where clusters begin and end. In the same way, the set of *frontier links* that have one of their nodes in a given cluster forms its *frontier*.

In Figure 17b, we show how the mapping from Figure 17a could be clustered, where *frontier links* (links that are not contiguous in terms of their mapping) and *cluster mappings* (mappings between clusters that are syntactically equal) are both represented using dashed arrows.

Readjusting the frontier links. Before moving the common clusters to the new module, we must make sure that it makes sense from the point of view of the target language, in other words, we must check that the frontier links are valid candidates for function extraction. There are several reasons why this may not be the case, the most common ones are:

- The target cluster of a frontier link may not represent an expression. Other types of Erlang terms (for example: patterns) cannot be the body of a function.
- It may be syntactically incorrect to introduce a function call at that position (in the parent cluster). For example, we cannot introduce a function call in the header of a function.
- It may occur that variables defined in the child cluster are used from outside of it. Because creating a function will also create a new scope, the variables defined inside cannot be used outside (they are not visible). Nevertheless, there are ways around this limitation, for example, it is possible to “export” variables out of the scope by passing them as result of the function and using pattern-matching around the function call.

In principle, if we try to extract functions in places where frontiers are not valid candidates, the result will either produce compilation errors or behave differently from the original code (both of them undesirable properties for refactorings). But we can add exceptions to the validation as long as we can create a post-processing mechanism that fixes the problems (we do this to solve some readability problems, see Section 4.3.2.4 on page 83). On the other hand, we can add unnecessary extra rules if that helps us increase readability of the final code.

If we find that some frontier links are invalid, we can move them by removing the mapping of nodes at the borders of common clusters and recalculating. The trade-off of this procedure is that, when used, it prevents replicated code from being generalised. In the worst case scenario, we will fallback to having no common clusters and that would leave the input modules as they were originally.

We take as basic principle that, if necessary, we can have code common to both instances replicated in both the *behaviour instances*, but we cannot have code unique to one of the *behaviour instances* in the *behaviour definition*. We could do so by adding conditional expressions, but that would make future extension of the behaviour (the creation of new instances for the generated *behaviour definition*) more difficult, and it may add unnecessary complexity to the resulting code.

Common clusters and unmatched clusters. Once we have decided which nodes will form the common clusters, we move them to the new module as top-level

functions, and we remove their nodes from the ASTs of the input modules.

The remaining contiguous nodes form what we call the “unmatched clusters”.

At this point, we have two kinds of cluster:

- The *common clusters* represent code that was present in both of the original ASTs. Each common cluster will translate into a function in the output *behaviour definition*. We call these functions `common` functions.
- The *unmatched clusters* represent code that was only in one of the original ASTs (or was common to both but there was no mapping between both versions). Each unmatched cluster will become a function in one of the output *behaviour instances*. Those of the functions that are not linked to the root of one of the original modules will become `callbacks`, and they will be part of the list of functions to be implemented by the *behaviour instances*.

4.3.2.3 Cluster linking

At this point, we know which code goes in which module, but we probably have several pieces of code (clusters) that are disconnected from each other. The linking process is in charge of restoring the control flow (at the frontiers of the clusters) so that the *behaviour instances* behave like the original input modules. This is achieved through the introduction of function calls and, in order to do this we must assign names to each cluster.

In Figure 18 we show how the example from Figure 17 on page 78 would be divided into modules and linked. The dashed node I in Version 2 is the rendering of an indirection cluster. Indirection clusters are explained later in this section.

As we mentioned before, we need to ensure that the list of callbacks for both nodes is the same. With this aim, we start by naming pairs of clusters that have a common parent cluster. We find them by visiting every lower frontier link of the common clusters. We can find the child clusters by following the hierarchy of their leaf nodes in their original ASTs.

We know that, for each lower frontier link in a common cluster (that is: for each link that starts in the cluster and goes to a different cluster) we necessarily have two alternative child clusters (since nodes in the common clusters are the result of merging two nodes from two different ASTs). We also know that these two child clusters are not the same cluster (since otherwise they would have been

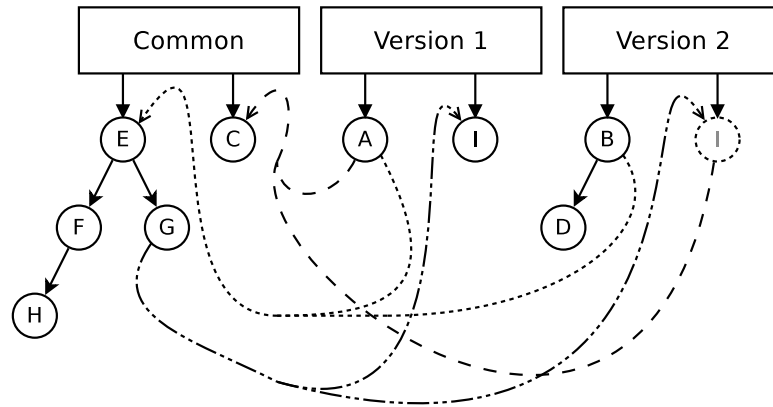


Figure 18: Example of linking

merged with the parent cluster). But we cannot assume that both child clusters are unmatched; it is possible that one or both of the child clusters are also common. This is because the frontier between clusters may be due to a discontinuity in the mapping, rather than to unmatched code.

For example, a node a and its parent b may both have an image (or pre-image) in the mapping and still belong to different clusters if the image of b is not the parent of the image of a , or if the image of a is in a different position in the child list of the image of b . We can observe this in Figure 17b (on page 78), where node G is parent of node C (in Version 2), and both have mappings; but they still belong to different common clusters.

Indirection clusters. Considering that, potentially, the input modules may have very different structures, it is possible that at some point we will find that we cannot give both alternative children (corresponding to each of the instances) the same name, since we may have already named one or both of the children differently. If that is the case, we create “indirection clusters” in either or both sides that we cannot name as desired. Indirection clusters are just top-level nodes that are rendered as a function with a function call as its only body. Of course, we want to minimise the use of indirection clusters, since they add complexity to the code.

For example, in the case of the cluster that contains the node I from Figure 17b (on page 78), we would ideally want to name its function the same as the cluster containing the node C , since they are both children of G . We know the content of

the node **G** is going to be in a single place (the *behaviour definition*, since it has a mapping); thus, we want the function call that links **G** with **C** and **I** to be the same, and for that we need **C** and **I** to have the same function name (otherwise we would need two function calls). But because the cluster containing **C** is a common cluster, this is not possible. That is why, in Version 2 of Figure 18, we have created an indirection cluster that has the same name (**I**), with the only purpose of redirecting the execution to the cluster containing **C** in common.

We are assuming, in the diagram in Figure 18, that each *subtree* made of round nodes in each module (or version) represents a single function that has the name of the root node. The ASTs of the output modules will obviously be normal trees with a single root, but we consider the AST of each separate function for simplicity, and we combine the different trees in each module into a single one at the end of the process.

When creating indirection functions there is also the non-trivial problem of finding appropriate names for the arguments, since the original function may use patterns and may have different names for each variable in each clause. We addressed this problem by reusing the solution implemented in the Erlang program documentation generator EDoc (Carlsson 2009). EDoc provides an elaborate algorithm with a series of heuristics that allows it to determine a name for the arguments of each function. In addition to looking at the names of variables when they exist, EDoc also checks the `spec` declarations when available, and also analyses composite patterns of several types in arguments, like match expressions, records, lists, and infix operators. In some cases, EDoc may fail to find anything that looks like a name for the argument and, in those cases (as a last resort), EDoc will generate a unique name starting with the letter “X” and ending with a number. This can be observed in the example of the Frequency Server pilot study (Section 4.3.4.1 on page 92).

Ensuring common arities. In Erlang, a call is considered different from another one if it has a different name or different number of arguments. Through indirection clusters we can make sure that alternative calls have the same name, but we also need to make sure that they have the same number of parameters in both *behaviour instances*. We do this by combining the parameter sequences of both alternatives. We add common parameters only once, and then we add

unmatched parameters of both sides. This gives us functions that have arity at most the sum of the arities of the functions combined, but the number of extra parameters depends on the extent to which the names of the parameters of the functions merged match each other.

Since the unmatched parameters are only used by one of the alternatives, in the opposite side we will use dummy values for those parameters, and we will prefix them with an underscore to avoid unused warnings. These dummy values will not cause a difference in the behaviour because they will not be used, and they are easy to generate thanks to the dynamic typing of Erlang, (we can simply use an atom like `none` or `undefined`).

We can see an example of this later in Figure 24 on page 101: the `callback_4/3` function has three parameters but only the first is used in the left version, and only the last two are used in right version; on the other hand the common function `common_1/4` is called using the atom `none` for the last two parameters from `prop_dets/0`, and using the atom `none` for the second parameter from `prop_parallel/0`.

4.3.2.4 Extra considerations

The integrated behaviour extraction refactoring subsumes several simpler refactorings and transformations, and, thus, some considerations applied to these simpler refactorings are also applicable to the integrated behaviour extraction. The main simpler refactorings implied by the integrated behaviour extraction refactoring are: function extraction and function module migration (that is: moving one function from one module to another). But, there are some considerations that are specific to this refactoring as well. In this section, we comment on some of these issues (both general and particular).

Combining sibling functions. The naïve approach for dividing ASTs translates horizontal borders between clusters into sequences of calls to functions with one expression each. By horizontal borders we mean frontier links that depart from one or more sibling nodes that belong to the same cluster but go to nodes that belong to other clusters. These calls can usually be merged into a single one, removing the need to “export” and pass variables among them.

In order to solve this problem without affecting the main part of the algorithm,

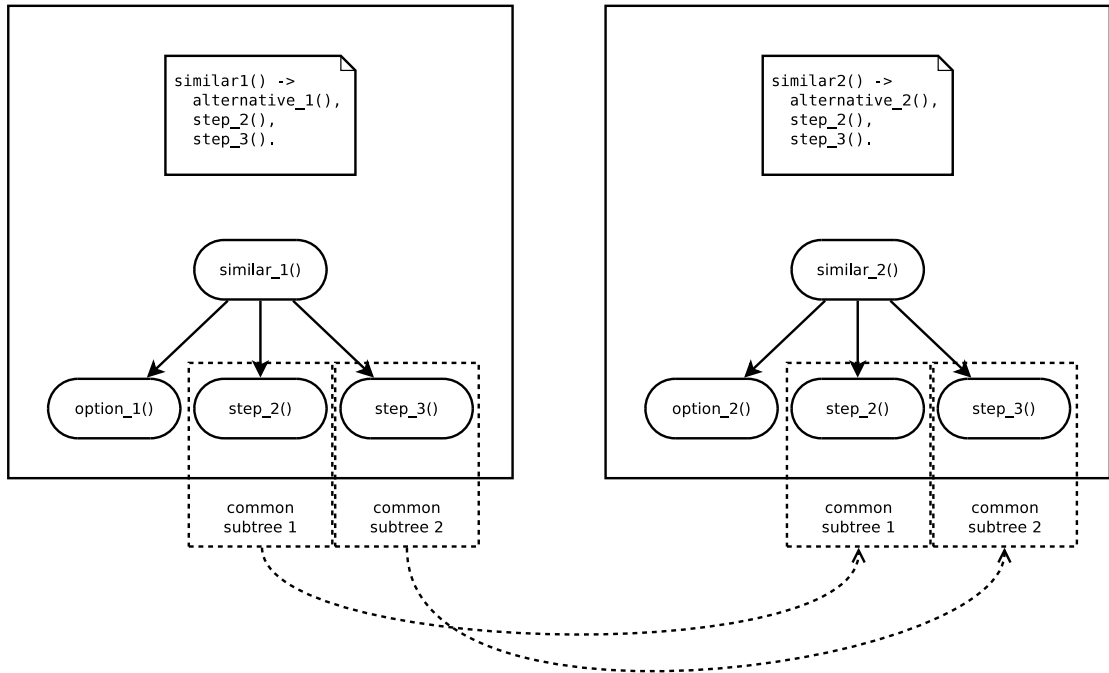


Figure 19: Example of fragmentation by horizontal border

we do a preliminary pass for detecting the places in the ASTs where horizontal division happens and we introduce blocks `begin ... end` (see Section 2.1.2 on page 13) to create a common parent for all of them in the AST. At the end of the process, we just remove the blocks we created since they are no longer necessary.

For example, the two snippets in Figure 19 would require the creation of two separate common clusters which would in turn translate into two different common functions which would have to be called independently (one for `step2/0` and one for `step3/0`). By introducing a `begin ... end` block as in Figure 20 we would only obtain one common cluster.

This artefact ensures that the frontier occurs in a single node and avoids unnecessary function fragmentation.

Exporting variables in block expressions. The rule for cluster adjustment about not allowing the creation of frontiers that would require “exporting” variables, as defined in Section 4.3.2.2 (on page 77), works well by default until we allow the combination of sibling functions.

For example, if we only have to apply function extraction to single expressions, whenever the expression is a match expression (for example: `Res = 1 + 1`), and

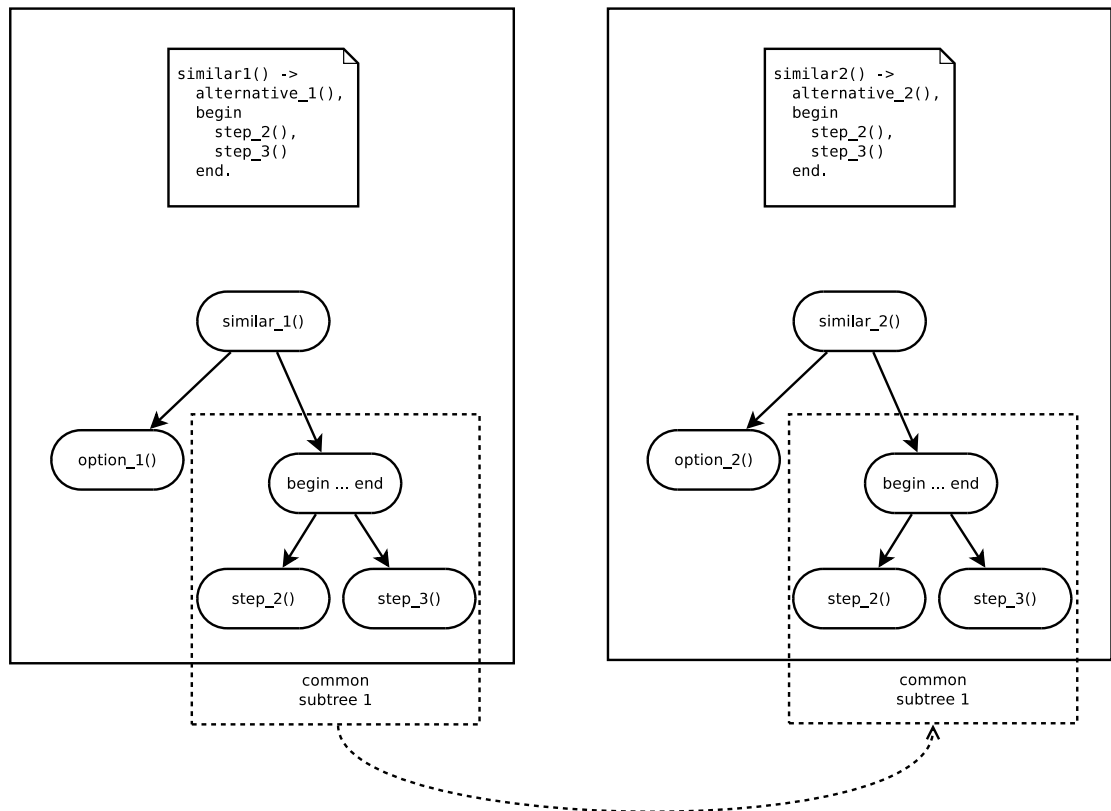


Figure 20: Example solution to fragmentation

if there are variables that would have to be “exported” (in our example it could be the variable `Res`), we can only apply function extraction to the right hand side of the match expression (in our example: `1 + 1`; which would leave us with something like `Res = common:common_1()`).

On the other hand, if we have a sequence of expressions, the assignment is no longer at the top of the subtree, thus, we cannot just rearrange the frontiers of the clusters without reintroducing the fragmentation. Thus, we must “export” the required variables in some other way.

Our solution consists of:

- adding an expression, to the end of the function extracted, for returning the “exported” variables in a tuple, and
- adding a match expression surrounding the function calls, in order to extract the variables from the tuple.

One example of this is the function `out:common_1/0` in Figure 22 on page 88. In general, being able to reduce the number of functions generated gives better readability, since high fragmentation forces developers to jump through several functions to understand small bits of functionality, as opposed to looking at a single fragment which is written consecutively in the code. Arguably, the addition of extra artifacts also makes it harder to read the extracted functions, but we consider this effect to be less damaging than fragmentation.

Result and exported variables combined. In some unusual scenarios, it may happen that both the “exported” variables and the result of the function are used from outside the function (see the case expression in Figure 21). Since our solution modifies the result of the function, we would be affecting the behaviour of the original code.

In order to prevent this from happening, we only create `begin ... end` blocks around expressions that either do not include the last instruction of a block or do not “export” any variables. We do not have to worry about modifying the value of intermediate sentences in blocks, because we know that these values are also discarded by the original code.

For example, let us consider the code in Figure 21. Our refactoring extracts two common functions `common_1` and `common_2`, (see Figure 22). We could, in

<pre> -module(mod1). -export([test1/0]). test1(A) -> D = case A of error -> Error = 1, Value = 2, {3, Value} end, {Error, D}. </pre>	<pre> -module(mod2). -export([test2/0]). test2(B) -> C = case B of ok -> Error = 1, Value = 2, {3, Value} end, {C, Error}. </pre>
---	--

Figure 21: Example of exported variables and result combined

principle have extracted a single function since the code:

```

Error = 1,    // in common_1
Value = 2,   // in common_1
{3, Value}   // in common_2

```

is common to both the input versions. But there is a conflict. The whole block “exports” the variable `Error`, but it returns the tuple `{3, Value}`.

If we added the variable `Error` at the end of the function we would be modifying its result (that is: the value that will be stored in variables `D` and `C` in `mod1` and `mod2` respectively) and, thus, altering the behaviour of the code. This is because the value of a block of expressions is the value of the last expression. Nevertheless, we can modify the return value for `common_1` to return the tuple `{Error, Value}`.

This is the reason why we extract two functions instead of one. It would still be possible to use only one function if we, for example, pack both the variable `Error` and the result (`{3, Value}`) into a tuple, we unpack it outside of the function and then return just `{3, Value}` as the last expression of the clause of the case block. Nevertheless, we chose the solution described above because it is generic, it avoids the creation of new variables, and it is simpler to implement.

There may be more cases where variables are “exported” from nodes that are not at the root of a subtree. In order to avoid breaking these kinds of frontier links for which we cannot automatically fix the “exported” variables, we ensure that common clusters do not “export” any other kinds of variables, by adjusting them as described in Section 4.3.2.2 (on page 77) .

```
-module(mod1).  
  
-export([test1/0]).  
  
-behaviour(out).  
  
test1(A) ->  
    D = case A of  
        error -> {Error, Value} = out:common_1(),  
                out:common_2(Value)  
        end,  
        {Error, D}.
```

```
-module(mod2).  
  
-export([test2/0]).  
  
-behaviour(out).  
  
test2(B) ->  
    C = case B of  
        ok -> {Error, Value} = out:common_1(),  
              out:common_2(Value)  
        end,  
        {C, Error}.
```

```
-module(out).  
  
-export([behaviour_info/1, common_1/0, common_2/1]).  
  
behaviour_info(callbacks) -> [];  
behaviour_info(_Other) -> undefined.  
  
common_1() -> Error = 1,  
              Value = 2,  
              {Error, Value}.  
  
common_2(Value) -> {3, Value}.
```

Figure 22: Output of our refactoring for modules in Figure 21

With our technique, those unusual cases would introduce a bit of replication in the result, but in exchange we will still be able to ensure the preservation of the original behaviour.

Macros and preprocessor directives. Moving a function from one module to another changes its context, it becomes affected by different directives. These directives can be conflicting with the directives of the destination module, they can include conditional directives (`ifdef` or `ifndef`), and they can even inline other files (`include` or `include_lib`) that, in turn, may contain additional directives too.

There is no easy universal solution for adjusting the directives so that the functions moved do not change behaviour. We opted for a compromise solution that will probably work most of the time: we copy all the individual directives on which the code of the function that we move depends. But because moving individual directives that are contained inside a conditional directive block may alter their behaviour, if the directives on which the function depends are inside a conditional directive block we copy the whole block instead.

Function migration artefact. The header of a function is mainly composed of patterns and guards (which do not accept function calls), because of this it is complicated to split the header of a function in parts, even if it has common patterns or even whole common clauses.

In cases where a function that exists in both versions has the same number of clauses and the same header for every clause in both versions, we can always move the whole function to the common module. Then, in order to keep the interface of the instance module unchanged we just insert an “indirection function”, that is: a function that has the same name and arguments as the original function, but instead of the same body it just has a call to the original function in its new location.

Function arity collision. Another consideration when moving functions is that there must not exist any function with the same name and arity in the destination module. In our case, this is mitigated by the fact that our destination module is new and, as such, there are no previous existing functions.

But we need to add one extra argument to some functions because, `common` functions, at some point, may need to call the unmatched modules back. Because the target module is only known at execution time, we pass the name of the module as a parameter (this can be seen in function `common_1` in Figure 15 on page 75).

This is not a problem for generated `common` functions, since every `common` function is generated with a unique suffix number. But it is a potential problem for functions we move as part of the *Function migration artefact*, they may also need an extra argument, and they may clash with another function moved by the *Function migration artefact* that:

- has the same name,
- has the same number of arguments plus one, and
- does not need an extra argument (because it does not call any other functions in the *behaviour instance*).

When this is the case, we need to generate, for one of the clashing functions, a new name that is unique (this can be done, for example, by adding a number at the end of the name), and to rename the function and all its references accordingly.

4.3.3 Wrangler usage

The work we have presented throughout Section 4.3 (on page 61) can be understood, to a large extent, in terms of smaller existing refactorings, as illustrated by the steps that form the interactive refactoring. Because of that, we have been able to reuse a lot of functionality that was already available in Wrangler. In particular, the refactorings that assist the interactive manipulation of Erlang behaviours have been implemented by using Wrangler's DSL for composite refactorings.

Several of the primitive refactorings, used by the new composite refactorings, already existed prior to this work. Some new refactorings and transformations were created through the use of the API for user-defined refactorings. For this reason, the work described in this chapter validates the effectiveness of Wrangler's DSL for extending the applicability of Wrangler.

But even some new primitive refactorings created as internal extensions to Wrangler during the course of this work were based on existing ones (in particular,

the new *copy module* refactoring was based on the previously existing *rename module* refactoring).

The refactoring for integrated behaviour extraction was implemented in its entirety as an internal extension to Wrangler, but both the internal infrastructure and the refactoring API were used extensively. This support allowed the new implementations to inherit properties like code layout and comment preservation.

Low-level mechanisms provided by Wrangler, like the categorisation of AST nodes, or the module information extraction, saved a lot of effort in the implementation of the integrated refactoring.

Finally, Wrangler refactorings have been extensively tested in production settings. Reusing these mechanisms gives our implementation an increased level of safety and reliability that would be very costly to achieve for an implementation from scratch.

4.3.4 Case study

In order to evaluate the refactorings implemented as part of this work, we have tried both behaviour extraction techniques (interactive and integrated) on two QuickCheck models with two variants each:

- models for testing the Frequency server (see Section 2.1.5 on page 15); and
- models for testing `ets` and `dets` term storage libraries (see Section 4.2.2 on page 58).

This case study aim at validating the proof of concept, as well as being realistic enough to highlight some limitations in our approach in practice. The reader can easily rerun these experiments since the full source code of the examples and the results of the case study are available in the `examples` folder of the GitHub repository of Wrangler at <https://github.com/RefactoringTools/wrangler>. This repository also contains the source of Wrangler, which includes the implementations of all the refactorings described in this chapter as of release 1.2.

The only runtime issue we found during the experiments was an error when applying the interactive technique to extract a small piece of code that contained the macro `?BUG_INSERT_NEW`, which is defined inside a conditional block defined

through preprocessor directives in `ets_eqc.erl`. We solved the problem by manually copying the macro definition to the destination module (the behaviour instance), and then the interactive refactoring worked as expected. The integrated refactoring did not require to manually copy the macro, it automatically copied the whole conditional macro block.

4.3.4.1 Frequency server

For the Frequency server, the integrated approach worked as expected since the difference between both variants is very small and localised. All the process, including parametrisation, was done automatically, but some boilerplate code was introduced.

The interactive approach allowed us to keep the interface of both modules in the common side, we just had to take one of the modules as starting point and move the different parts to an instance by using the refactorings. But we had to find the differences and commonalities manually, and we had to manually parametrise the name of the instance to use (which we did by introducing a macro). As we explained in Section 4.3.1.4 (on page 67), other solutions are possible, but by using a macro we did not need to modify the interface in the *behaviour definition*. We also realised that having the interface in the *behaviour definition* side (as obtained through the *interactive* approach) gives in this case a much more concise solution, because moving it to the *behaviour instance* would have implied introducing a lot of boilerplate code (as the solution produced by the *integrated* refactoring shows).

In both cases, we only created one small callback.

```
callback_1(Alloc, Freq) -> not lists:member(Freq,Alloc).
===
callback_1(_Alloc, _Freq) -> false.
```

However, the integrated approach needed to create 16 indirection functions for each instance, since, despite it unified most of the source code, it still needs to preserve the interface of the original modules. On the other hand, the version generated by the integrated refactoring is parametrised dynamically (as opposed to statically with a macro). For example, the function `precondition/4` was moved to the common module (`full_fsm`), but an indirection function was inserted:

```
precondition(From, To, S, Op) ->
    full_fsm:precondition(?MODULE, From, To, S, Op).
```

And we can see that the name of the module was added to the arguments of the precondition function so that it knows the callback of which module to call:

```
. . . .
precondition(_Module, {state, N}, {state, M}, {_Free, Alloc},
    {call, frequency, deallocate, [Freq]}) when M < N ->
    lists:member(Freq, Alloc);
precondition(Module, {state, _N}, {state, _M}, {_Free, Alloc},
    {call, frequency, deallocate, [Freq]}) ->
    Module:callback_1(Alloc, Freq);
precondition(_Module, {state, N}, {state, M}, {[ ], _Alloc},
    {call, frequency, allocate, [ ]}) when M > N ->
    false;
. . . .
```

4.3.4.2 ets and dets tables

For the variants of `ets` and `dets` tables, the integrated approach automatically abstracts an important part of the common code, and the result seemed to behave in the same way as the original code when running the tests. But the first attempt showed that some prior clone removal was convenient in order to reduce the number of callbacks created by the refactoring. For example, we created a function to store the name of the table used during the tests (of course, a macro could have been used instead):

```
table_name() -> ets_table.
===
table_name() -> dets_table.
```

However, even after clone removal, some of the callbacks generated by the integrated approach – despite being correct – may arguably be hard to understand for humans, since they do not carry any clear meaning, and they do not have meaningful names.

One clear example is the following snippet extracted from the result of the initial run of the integrated refactoring during the `ets` and `dets` case study.

```
common_17(Module) -> [{Module:callback_36(),  
                        Module:callback_37()}].
```

The integrated algorithm found that the list with a tuple was common to both `ets` and `dets`, but none of the elements of the tuple is common.

The example was extracted from when we ran the integrated refactoring configured to discard common clusters that have 3 or fewer nodes, because an AST of that size is very unlikely to be a meaningful match. But we can increase this limit if we get too many callbacks: in the case of the `ets` and `dets` models we tried several values, and in the end set the value to 7, since it got rid of most meaningless callbacks and produced an output that we could understand more or less easily (without having to alternate between the three modules many times to see what had happened). Increasing the limit further would progressively remove common functions (and, as a consequence, callbacks) until the result of the refactoring become the same as the input and no redundancy would be abstracted out.

For a later run of the integrated refactoring (whose results are published in the examples directory of the Wrangler distribution), we also made some ad-hoc modifications to the matching algorithm in order to obtain a better mapping, since the mapping is not part of the contributions of this section, nor is it a target of the case study. Note that the ad-hoc modifications are not part of the implementation that has been published as part of Wrangler (since they may be detrimental in the general case, and since they make the matching algorithm quite inefficient).

In order to give an idea of the effect of the minimum common cluster size, we have executed the integrated refactoring in the `ets` and `dets` example for minimum common cluster sizes from 1 to 20 nodes (this time using the published implementation of the matching algorithm). The statistics about the results can be seen in Table 1. Note that if the mapping used had been the same, we would have expected the number of common functions to decrease, and the number of callbacks to be increasing; but the reason that this is not the case is that the matching algorithm was implemented in a non-deterministic way.

Minimum common cluster size in nodes	Number of common functions	Number of callbacks per instance
1	26	36
2	19	32
3	17	23
4	16	22
5	13	25
6	14	21
7	11	23
8	10	20
9	11	24
10	11	22
11	7	18
12	6	21
13	6	16
14	4	15
15	5	17
16	2	13
17	2	13
18	2	13
19	2	13
20	3	15

Table 1: Influence of the minimum common cluster size parameter

Using the ad-hoc matching algorithm and discarding common clusters of 7 or fewer nodes, a better result was generated for the `precondition/4` function:

```
precondition(From,_To,S,{call,_,open_file,[_,[{type,T}]]}) ->
    lists:member(S#state.type,[undefined,T])
    andalso From==init_state;
precondition(_From,_To,_S,{call,_,insert_new,_}) ->
    ?BUG_INSERT_NEW;
precondition(_From,_To,_S,{call,_,_,_}) ->
    true.
===
precondition(From,_To,S,{call,_,open_file,[_,[{type,T}]]}) ->
    lists:member(S#state.type,[undefined,T])
    andalso (From==init_state orelse ?BUG_OPEN_FILE);
precondition(_From,_To,_S,{call,_,insert_new,_}) ->
    ?BUG_INSERT_NEW;
precondition(_From,_To,_S,{call,_,_,_}) ->
    true.
```

The whole function was moved to the common module except for the conflicting part:

```
precondition(Module,From,_To,S,{call,_,open_file,[_,[{type,T}]]}) ->
    lists:member(S#state.type,[undefined,T])
    andalso
        Module:callback_13(From);
precondition(_Module,_From,_To,_S,{call,_,insert_new,_}) ->
    ?BUG_INSERT_NEW;
precondition(_Module,_From,_To,_S,{call,_,_,_}) ->
    true.
```

And an indirection function and the conflicting part was kept as a callback in the original modules:

```
precondition(From, To, S, X4) ->
    common:precondition(?MODULE, From, To, S, X4).
```

```

callback_13(From) -> From == init_state.
===
precondition(From, To, S, X4) ->
    common:precondition(?MODULE, From, To, S, X4).

callback_13(From) -> From == init_state or else ?BUG_OPEN_FILE.

```

Different heuristics could be used to discard meaningless common functions, clone detection and elimination functionality of Wrangler already allows the user to parametrise certain metrics to decide which clones to remove. But this choice may not be either a matter of the number of nodes in the common functions or the size of the code generated, but of the meaning of the code in the particular context, which is harder to measure. We may decide to not combine two atoms, despite their being syntactically equivalent, because they may mean different things. For example, we may have in both input modules the number 32 and in one module it may mean kilograms and in other centimetres; or we may have two atoms with the same name refer both to a function and to a module name. In both these examples, it would probably not make sense to combine both equal instances into the same function. Since these aspects are difficult to measure, we suggest reengineering the system to allow users to make the choice.

Another issue we found is that the matching algorithm creates crossed mappings (in other words: it maps two nodes to what should be each other's image in the opposite AST) between methods that are similar. Crossed mapping occurs when two parts of the ASTs are structurally similar within each of the modules; we usually prefer to abstract out functionality that has the same or similar functionality and intention, often this corresponds to code that is structurally similar, but not always. Fortunately, these crossed mappings are quite easy to spot in the result after labelling the callbacks with their calling functions.

One effective solution for crossed mappings is to remove the structural similarities before applying the integrated refactoring. They can also be used as warning signs of replication existing in the code we are trying to synchronise. In Section 4.3.5 (on page 98), we show an example of two pieces of code that caused cross mapping (extracted from the pilot study on `ets` and `dets`), and we describe how using the integrated refactoring on the conflicting bits (as it is currently

implemented) can help us find and isolate instances of structural replication.

The integrated refactoring (when configured to discard common clusters of 6 or fewer nodes) moved 9 functions to the behaviour definition, created 14 callbacks and 12 common functions, but also created 9 indirection functions to preserve the interface of the original modules. When using the iterative approach, we started with all the functions of one version in the common module, and we moved approximately 11 functions to the instance. Later we had to manually compare the instance to the other version in order to create the other instance.

When choosing the clusters manually, we often defaulted to moving complete functions unless the changes were very localised, in those cases we created functions for individual expressions. Again, the results of the manual version can be more concise because we unified both interfaces and parametrised them with a macro.

4.3.5 Integrated approach as aid in unification

We described in the pilot study with `ets` and `dets` that replication within each of the input modules for the integrated approach makes it difficult for the mapping algorithm to obtain the right mapping. But the integrated refactoring has proven useful to help remove this replication, and after doing that, the mapping algorithm does a better job.

In this section, we illustrate how our refactoring helps us merge two similar functions. Let us consider the two functions extracted from the `dets` model in Figure 23. We realised that these two functions had many commonalities because the integrated refactoring mismatched them when trying to pair them with their syntactically equal counterparts in the `ets` model.

It is easy to see that there are many commonalities between both, but their structures are quite different. Abstracting out their commonalities is not trivial, and could easily lead to making a mistake. Thus, we applied the integrated refactoring to try to obtain a template for their combined function. In practice, in order to achieve this we had to move each function to a different module (this suggests that it would be a good idea to implement this refactoring for pairs of functions in addition to pairs of modules).

```

prop_dets() ->
  ?FORALL(Cmds, more_commands(3, commands(?MODULE)),
    ?TRAPEXIT(
      begin
        dets:close(dets_table),
        file_delete(dets_table),
        {H,S,Res} = run_commands(?MODULE, Cmds),
        ?WHENFAIL(
          io:format("History:~p\nState:~p\nRes:~p\n", [H,S,Res]),
          aggregate(command_names(Cmds), Res == ok))
        end)).

```

```

prop_parallel() ->
  fails(
    ?FORALL(Attempts, ?SHRINK(1, [100]),
      ?FORALL({Seq,Par}, parallel_commands(?MODULE),
        ?ALWAYS(Attempts,
          ?TIMEOUT(1000,
            begin
              [dets:close(dets_table)
                || _ <- "abcdefghijkl"],
              file_delete(dets_table),
              {H,ParH,Res} = run_parallel_commands(?MODULE, {Seq,Par}),
              ?WHENFAIL(
                io:format("History:~p\nParallel:~p\n\nRes:~p\n",
                  [H,ParH,Res]),
                collect(
                  length(Par),
                  aggregate([length(P) || P <- Par],
                    collect(
                      length([ok
                        || P <- Par,
                          {set,_,{call,_,open_file,_} <- P}),
                        Res == ok))))))
            end))))).

```

Figure 23: Original version of similar functions to be unified (after removing the comments and some redundant parenthesis).

On the first run, the integrated approach produces (among others) the following pair of callbacks:

<code>callback_1(S, _ParH) -> S.</code>
<code>callback_1(_S, ParH) -> ParH.</code>

Which tells us that `S` and `ParH` may be two variables with different names that are used in the same way (and indeed they are, so we rename `ParH` to `S`). This is not necessarily always the case, because two variables that are used interchangeably in one place may not be used interchangeably in another place.

We also observed that there is one subtle difference in the style of the code that surrounds the call to the `common_1` function preventing it from also including that code (`common_1` unifies the calls to the function `dets:close(dets_table)`, see below), `common_1` is inside a list comprehension that makes it execute several times in one side, and it is by itself on the other:

<code>common:common_1(),</code>
<code>file_delete(dets_table),</code>
<code>[common:common_1() <- "abcdefghijkl"],</code>
<code>file_delete(dets_table),</code>

We can, for example, replace the first occurrence to have the same structure without altering its behaviour by writing it like:

<code>[dets:close(dets_table) <- "a"],</code>

In Figure 24, we can see the result of running the refactoring on the two functions after making these two modifications. We probably do not want to leave the functions like this; for functions that have small body we may want to pass them as parameters instead. But we now have a starting point where the hard task of finding the differences has been done for us.

The phenomenon we have seen here: that, in order for a substantial transformation to be applied, there needs to be a degree of manual support before and after the main transformation; is common to many refactoring scenarios. We have also seen this phenomenon, for example, in clone detection Li and Thompson (2009).

<pre> prop_dets() -> ?FORALL(Cmds, more_commands(3, commands(?MODULE)), ?TRAPEXIT(common:common_1(?MODULE, Cmds, none, none))). callback_1(Cmds, Res, _Par) -> aggregate(command_names(Cmds), common:common_2(Res)). callback_2() -> "History:~p\nState:~p\nRes:~p\n". callback_3() -> "a". callback_4(Cmds, _Par, _Seq) -> run_commands(?MODULE, Cmds). </pre>	<pre> prop_parallel() -> fails(?FORALL(Attempts, ?SHRINK(1, [100]), ?FORALL({Seq, Par}, parallel_commands(?MODULE), ?ALWAYS(Attempts, ?TIMEOUT(1000, (common:common_1(?MODULE, none, Par, Seq)))))). callback_1(_Cmds, Res, Par) -> collect(length(Par), aggregate([length(P) P <- Par], collect(length([ok P <- Par, {set, -, {call, -, open_file, _}} <- P]), common:common_2(Res)))). callback_2() -> "History:~p\nParallel:~p\nRes:~p\n". callback_3() -> "abcdefghijk1". callback_4(_Cmds, Par, Seq) -> run_parallel_commands(?MODULE, {Seq, Par}). </pre>
<pre> common_1(Module, Cmds, Par, Seq) -> [dets:close(dets_table) _ <- Module:callback_3()], Module:file_delete(dets_table), {H, S, Res} = Module:callback_4(Cmds, Par, Seq), ?WHENFAIL(io:format(Module:callback_2(), [H, S, Res]), Module:callback_1(Cmds, Res, Par)). common_2(Res) -> Res == ok. </pre>	

Figure 24: Result of unifying the functions from Figure 23 (after removing redundant parenthesis and comments).

4.3.6 Discussion

The last example of the pilot study suggests that the final decision of what common code would be convenient to abstract should be left to the user. Currently the implementation of the integrated refactoring tries to extract all the pieces of code that have more than one node in their AST.

Better heuristics could be used based on the size of the code before and after the abstraction, or on the number of children the subtree has. But, heuristics are probably not enough, since what must be abstracted out seems to depend mainly on the semantics.

For example, it may be interesting to abstract a single common atom if whenever we want to change that atom it will only make sense to change it in all the instances, in other words, the atom has a general meaning. The inverse is also true, we may have a big replicated piece of code that just happens to be common to both approaches but is just a coincidence.

We can see this clearly if we look at etymology: the words “cypress” and “spree” both have the substring “pre”, but it is, as far as we know, a coincidence, abstracting “pre” in this case would only make things confusing. On the other hand, “preface” and “preview” also share the substring “pre”, but this time we know it is not a coincidence; in both words it means “before”. In code, this occurs often when we have values that happen to have a similar structure, as we have seen with `common_17/1` shown in Section 4.3.4.2 (on page 93).

In conclusion, the interactive approach allows for a more customised and clear division of matched and unmatched parts, but requires more manual work; whereas the automated version requires almost no work, but does not always do what is desired.

One idea for future work would be to add a visual “mapping editor” to the integrated solution, in order to give the choice of what to abstract to the user while keeping the rest of the process fully automated. Automatic parametrisation could be applied to the interactive approach, but that would require a more global approach instead of an iterative one.

4.4 Model parametrisation

In Section 4.3, we have studied how to model the differences and similarities between implementations by looking at their code. But two systems do not need to be implemented similarly in order to behave similarly, and we may not have access to their source-code (as is usually the case with web services, for example). In addition, source parametrisation does not allow us to compare implementations when their source code is very different, since it is based on finding commonalities.

Having a way to understand the differences in behaviour between different systems can help us make better choices when selecting between several alternatives, and it can help users and developers understand the effect of different configurations of the system.

If we want to model differences in behaviour we must have a way of representing (a model) that is independent of the implementation. Ideally the representation will be the same for any two implementations that exhibit the same behaviour when presented with the same inputs.

In this section, we present a series of experiments that study the effect of different configurations of a system in their behaviour as represented by a finite-state machine (FSM) model.

The experiments described in this section were carried out using the existing PLTSDiff algorithm (see Section 4.2.4.1 on page 60) and the “Automatic Inference of Erlang Module Behaviour” technique (Taylor, Bogdanov and Derrick 2013), as implemented in the combination of the regular inference tool Statechum (see Section 4.2.4 on page 60) and the Synapse interface for Erlang (see Section 4.2.4.2 on page 61). These tools take a working Erlang implementation and automatically exercise its API and generate an FSM model of its behaviour; alternatively, they take two FSMs and produce a “diff FSM” that shows the differences between both. The contribution presented here is not the tools (which existed previously to our work), but their application to the Frequency server. In particular, we compare four different configurations of the Frequency server example (see Section 2.1.5 on page 15).

4.4.1 Variants and configurations

We use four different configurations of the Frequency server as target for the experiments. The source-code for the base configuration (that is: the configuration we use as reference for comparison) is provided in Appendix A. The three other variants are produced by introducing small modifications to the source of the base configuration in Appendix A. For clarity, we present these variants in the form of small patches distributed throughout the rest of the chapter.

In Section 2.1.5 (on page 15), we have already given a brief specification of the expected behaviour of the Frequency server. But this specification is not detailed enough to describe the exact behaviour of the system; in fact all the variants used here conform to the specification. For generating the variants we have considered the following three aspects:

1. *Number of initial frequencies*: How many frequencies can be allocated by clients? We could consider there are infinite frequencies, even though that is probably not the case in real life. In this work, in order to keep models small, we consider two possibilities with few frequencies:
 - (a) 2 initial frequencies: represented with numerals 10 and 11.
 - (b) 3 initial frequencies: represented with numerals 10, 11 and 12.
2. *Behaviour in case of illegal deallocate*: What should be the behaviour of the server when a client tries to deallocate a frequency that is not allocated?
 - (a) **cannot**: the server throws an exception.
 - (b) **noop**: the server does nothing, but acts as with a normal deallocation.
3. *Order of allocation*: If there are several frequencies available, which frequency should be allocated first?
 - (a) **smallf**: the server always allocates the frequency represented by the smallest number from the ones available.
 - (b) **lifo**: the server stores the free frequencies in a stack, the frequency at the top of the stack is always allocated first. At the beginning, the frequencies with smaller numerals are closer to the top of the stack.

For simplicity, we represent combinations of these configurations as an Erlang tuple with the form `{NumFrequencies, DeallocationMode, AllocationMode}`, for example: `{3, cannot, lifo}`. The base implementation we use in this work (source-code provided in Appendix A) implements the configuration `{2, cannot, smallf}`.

4.4.2 Experiment structure

Through the rest of the chapter, we show the result of three different experiments (one per configuration). In each of the experiments we compare two variants, that is: the base variant (`{2, cannot, smallf}`), and an alternative variant, which is different from the base variant in exactly one of the configuration aspects described in Section 4.4.1 (on page 104).

For each configuration, we ran StateChum through Synapse for inferring an FSM. And, for each experiment, we show:

- the patch that shows the modifications we did to the base code in order to obtain the alternative configuration,
- the state-machines generated by StateChum for both of the configurations being compared, and
- the diff state-machine generated by PLTSDiff, that highlights the differences between both state-machines.

4.4.3 Deallocation behaviour

Let us first compare the two deallocation behaviours. The base implementation (`{2, cannot, smallf}`) throws an exception when we try to deallocate a frequency that is not allocated. Figure 25 shows how to modify the base implementation to ignore invalid deallocate calls instead (configuration `{2, noop, smallf}`).

In Figure 26 (on page 107), we can see the FSMs inferred by StateChum for both alternatives of deallocation behaviour. The labels on the transitions contain the function that produces the transition, the list of arguments the function takes, and the result, separated by commas.

```

deallocate({Free, Allocated}, Freq) ->
  {value, {Freq, Pid}} = lists:keysearch(Freq, 1, Allocated),
  unlink(Pid),
  NewAllocated = lists:keydelete(Freq, 1, Allocated),
  {[Freq|Free], NewAllocated}.
case lists:keysearch(Freq, 1, Allocated) of
  {value, {Freq, Pid}} ->
    unlink(Pid),
    NewAllocated = lists:keydelete(Freq, 1, Allocated),
    {[Freq|Free], NewAllocated};
  _ -> {Free, Allocated}
end.

```

Figure 25: Modifications for `noop` deallocation

The states in the models are labelled with numbers preceded by the letter 'P' (which stands for positive), the negative states and the transitions that would lead to a negative state (transitions that produce an exception) are omitted from the diagrams for clarity.

The state in the middle, shaped like a star, is the initial state. We can see that, in all diagrams, every call to `stop/0` ends on the initial state. The other four states represent all the possible combinations of available frequencies that can occur (considering we can only have a maximum of two allocated frequencies at any given time).

If we look carefully, we can identify the state that has no allocated frequencies at the end of the `start/0` transition (labelled P1001), we can also identify the state with no free frequencies as the one that has the self-transition with `allocate/0` that has return value `{error, no_frequencies}`.

The diagrams in Figure 26 are difficult to compare because the equivalent states are placed differently. But in the diagram produced by PLTSDiff, shown in Figure 27, we can clearly see that the difference between both diagrams is simply that the `noop` configuration has self-transitions for the deallocations that would fail (deallocations of frequencies which are not allocated), whereas the `cannot` configuration does not have any because they produce an exception (and negative states are omitted).

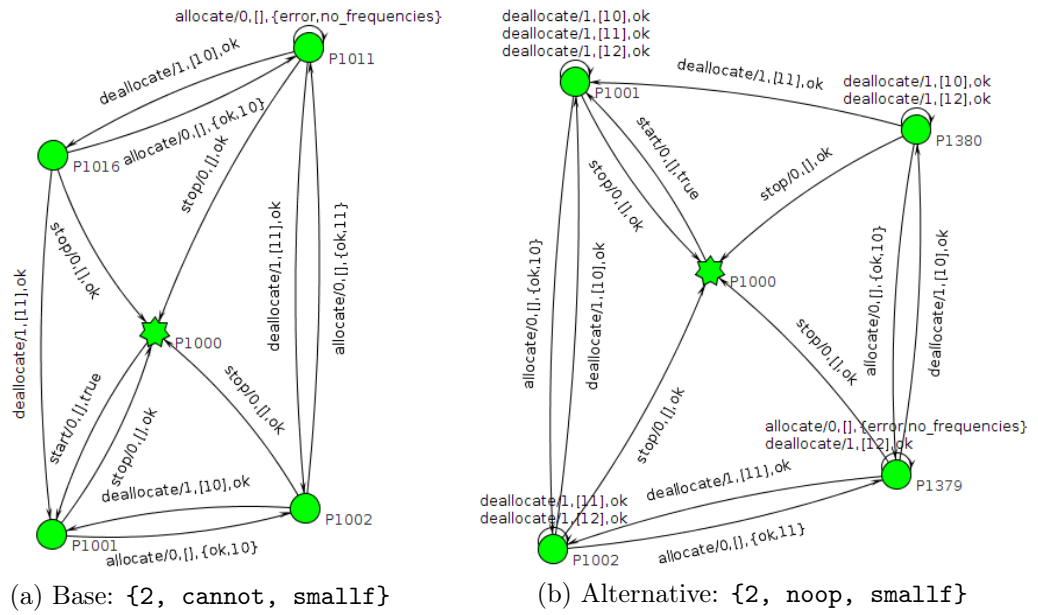


Figure 26: Both deallocation behaviours side by side

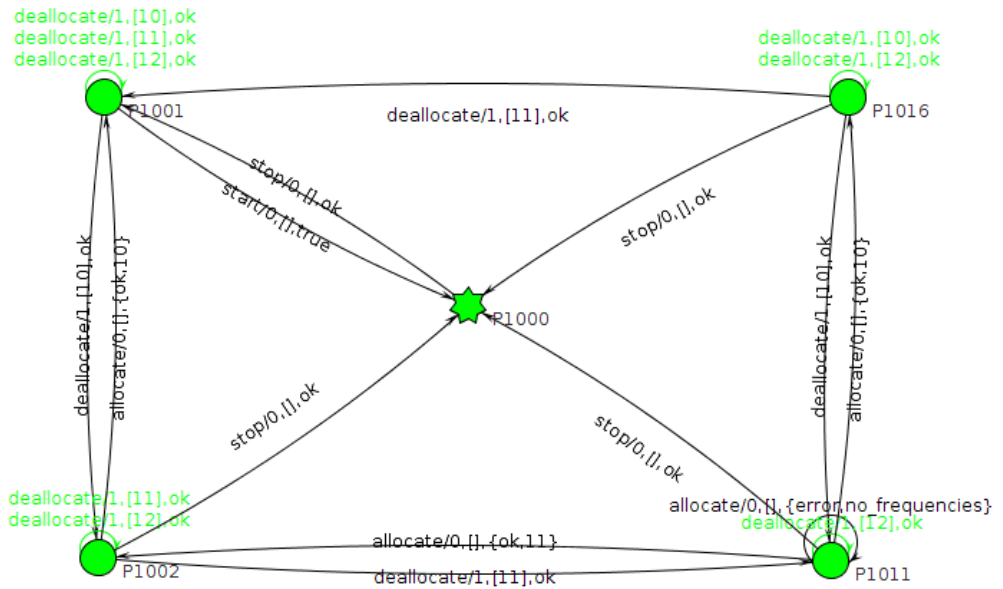


Figure 27: Differences between cannot and noop deallocation


```

loop(Frequencies) ->
  receive
    {request, Pid, allocate} ->
      {NewFrequencies, Reply} = allocate(sortfreqs(Frequencies),
        Pid),
      {NewFrequencies, Reply} = allocate(Frequencies, Pid),
      ...
sortfreqs({Freqs, Allocated}) ->
  {lists:sort(Freqs), Allocated}.

```

Figure 28: Modifications for `lifo` allocation

4.4.4 Allocation behaviour

Let us now compare the two different allocation implementations. We use the base configuration `{2, cannot, smallf}` as an example of a configuration that allocates smaller frequencies first, and `{2, cannot, lifo}` as an example of an implementation of last in first out allocation. The patch applied to the base configuration for obtaining the `lifo` configuration is shown in Figure 28. And Figure 29 shows the FSMs inferred by StateChum for both implementations displayed side by side.

We have already discussed the base configuration in Section 4.4.3. We can see that the `lifo` configuration has five states (instead of four) in addition to the initial state. The extra state (P1024) corresponds to the extra ordering of the frequencies in the stack when both are free, that is: 11 on top of 10. We can see that P1024 and P1001 are equivalent except in that, from P1024, `allocate/0` produces 11 instead of 10.

The diagram produced by PLTSDiff (in Figure 30) shows it more clearly, the state P1024 is an alternative to P1001 that occurs when we deallocate both of the frequencies in ascending order (first 10 and then 11).

4.4.5 Number of initial frequencies

Finally, let us see the effect on the FSM model of increasing the number of available frequencies. We use the base configuration `{2, cannot, smallf}` as an example with two available frequencies, and we use the same configuration with three available frequencies `{3, cannot, smallf}` as alternative. The patch for

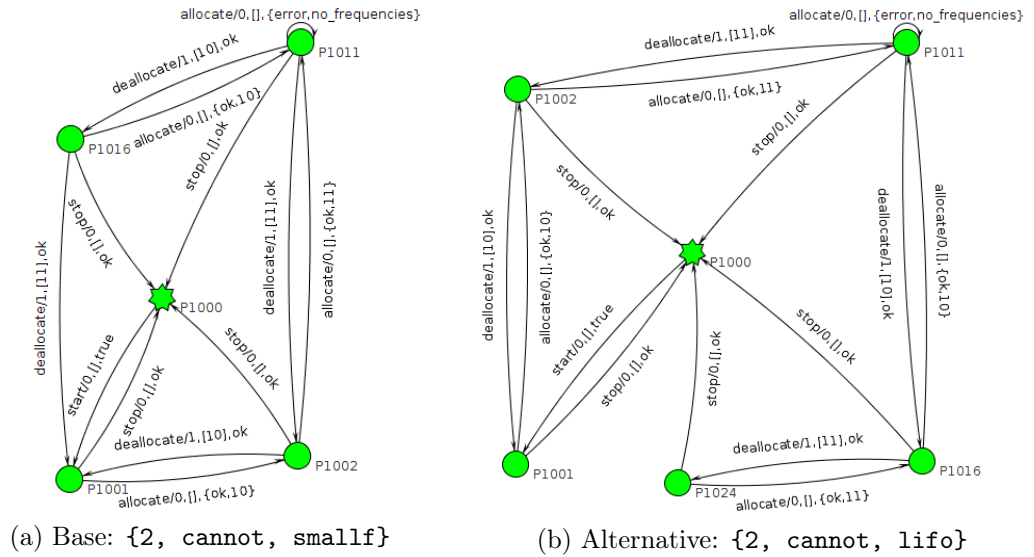


Figure 29: Both allocation behaviours side by side

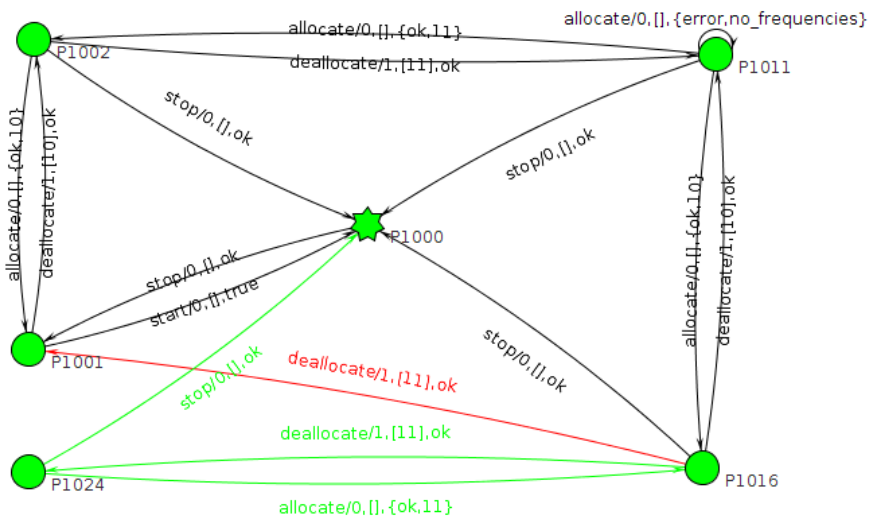


Figure 30: Differences between `smallf` and `lifo` allocation

```

Frequencies = {get_frequencies(), []},
loop(Frequencies).

% Hard Coded
get_frequencies() -> [10,11,12].
get_frequencies() -> [10,11].

%% The client Functions

stop()          -> call(stop).
allocate()      -> call(allocate).
deallocate(Freq) -> call({deallocate, Freq}).

```

Figure 31: Modifications for configuration with 3 available frequencies

generating the configuration with three frequencies is shown in Figure 31.

We can guess that even though the order of deallocations does not matter, since we use `smallf` allocation in both cases, we will have many more states because of the added frequency.

Indeed, as it can be seen in Figure 32 the FSM for the configuration `{3, cannot, smallf}` has 9 states: $2^3 = 8$ states to represent all the combinations of allocation and deallocation of the three frequencies, and the initial state. This time we have omitted the FSM of the base configuration for clarity.

We can see that the FSM for the configuration `{3, cannot, smallf}` is already a bit less practical and more difficult to read, even if it can still be understood if analysed carefully. But we can see that the increase in the number of states is exponential to the number of available frequencies, and we can foresee that FSMs generated would quickly become impractical if we keep increasing the number.

The diagram generated by PLTSDiff (shown in Figure 33) does however give us some interesting insights. On the one hand, we can see directly that the five states in the middle are almost unchanged (we know because their transitions are black), except for the `start/0` transition, and the transitions that take or return the number 12.

What is not shown so clearly by the diff diagram is that the outer four states are almost isomorphic to the original ones too (except for the transitions that take or return the number 12).

Furthermore, the transitions that allocate and deallocate the new frequency (12) are used to move between the inner and the outer four states.

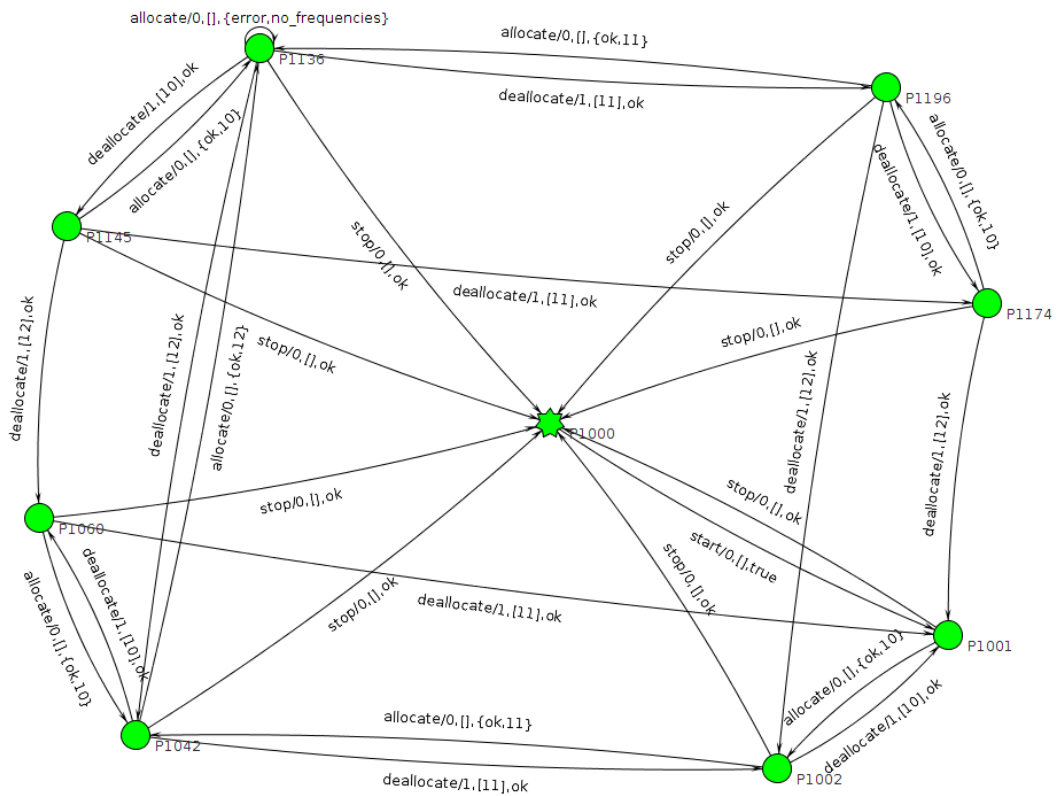


Figure 32: Behaviour for 3 frequencies: {3, cannot, lifo}

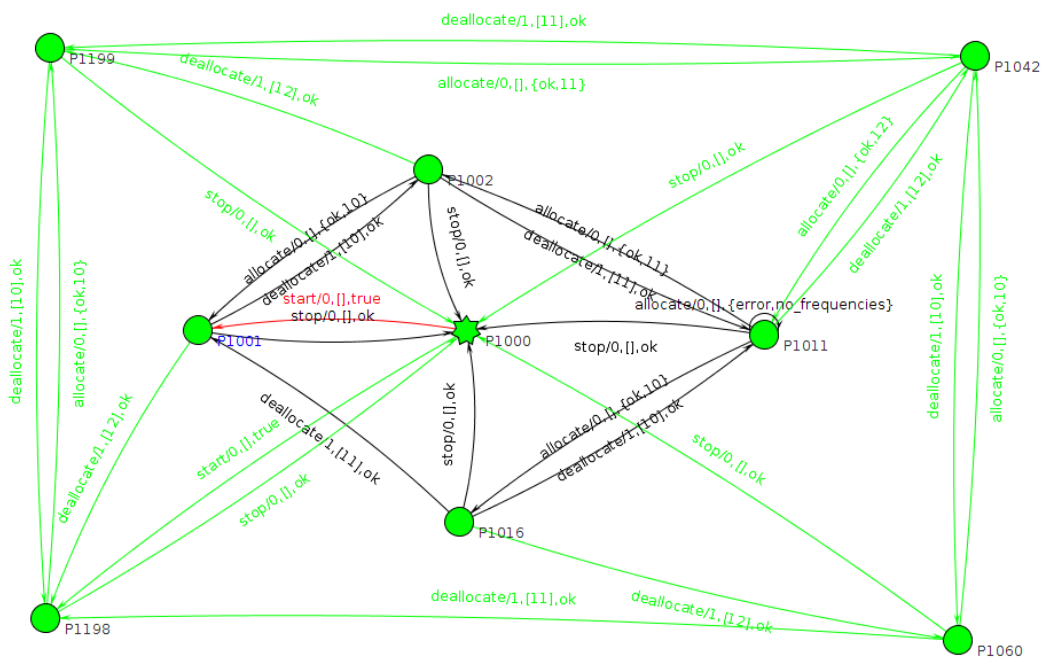


Figure 33: Differences between 2 and 3 available frequencies

These patterns show us the source of the state explosion: the introduction of a new frequency produces a new dimension that a finite-state machine is only able to represent by duplicating part of the state space. This patterns suggest a solution: we must find a different way of reflecting an increase on the number of frequencies that is independent of the behaviour of each frequency.

4.5 Chapter conclusions

In this chapter, we have studied automatic mechanisms for generalising from examples. The main insight from the experiments in this chapter is that generalisation follows from the abstraction of commonalities: we can generalise code by looking at patterns in the syntax, we can generalise state machines by merging states that are equal (or similar), and in the next chapters we will see how we can further generalise state machines by finding patterns in data usage, not only in control.

Source parametrisation. In Section 4.3 (on page 61), we have studied the automation of source code parametrisation. We have seen that deciding what to abstract depends on the semantics of the code and, thus, benefits from some level of user interaction. But we have also seen that the process of modifying the code can be automated to a large extent, that the automation of the process consists of refactorings and transformations that, in their majority, already exist, and we have obtained some insight that could be used in the future for the development of new tools and refactorings.

Nevertheless, source parametrisation is only applicable to the generalisation of versions of software that are implemented similarly and in the same language. We can indeed use the work presented to aid the development of parametrised test models but, in order to better understand the evolution of systems and their behaviour, we must look at implementation-independent models rather than their source code.

Model parametrisation. The finite state models studied in Section 4.4 (on page 103) are implementation-independent and can be used directly for test generation. But in Section 4.4.5 (on page 108), we have seen that the combination of

independent stateful subsystems (independent frequencies) can lead to an exponential increase in the number of states of FSMs if we model them as one single system.

We have also seen that, in some cases, the state explosion phenomenon presents itself as near-replication of “sub-state-machines”. The existence of this near-replication or parallelism between “sub-state-machines” gives us a clue of what we can do to adapt this way of representing behaviour (state machines) so that we avoid state explosion for at least some scenarios.

In Chapters 5 and 6, we study alternative ways of representing systems that attenuate this kind of state explosion by introducing explicit information about data flow in our models, with the ultimate aim of obtaining a representation mechanism that is more scalable and applicable to practical, real-life examples.

Chapter 5

Combining control and data

In Chapters 3 and 4 we have studied several ways of reducing the effort necessary to obtain models for use in testing. In both chapters, we have used FSM models to represent the behaviour of a simple system (that is: the Frequency server).

Reusing and parametrising can indeed reduce the effort required for testing systems. But manually writing test suites is still costly and only targets a finite set of scenarios (the ones described by the test suite).

Property-based testing (see Section 1.5 on page 5) allows the generation of an arbitrary number of tests, thus, this technique potentially allows to achieve higher coverage by writing tests that contain less code. Nevertheless, writing general properties and models is often more challenging than writing individual examples.

In this chapter, we show how existing unit tests can be leveraged to provide more testing value through inferring a model for the system from the tests, and by using the inferred model to generate new tests. There has been effort in the past for aiding testers and developers to use property-based testing by automating the process of inferring arbitrary properties from working implementations (Claessen, Smallbone and Hughes 2010). In this chapter, we analyse how to automate the transition from unit testing to property-based testing of stateful systems (in particular web services), by instrumenting the execution of existing test suites (in particular JUnit test suites), by using the information obtained to generate models for the tested system, and by using such models for automatically generating new tests.

These three processes combined allow us to assist testers and developers in

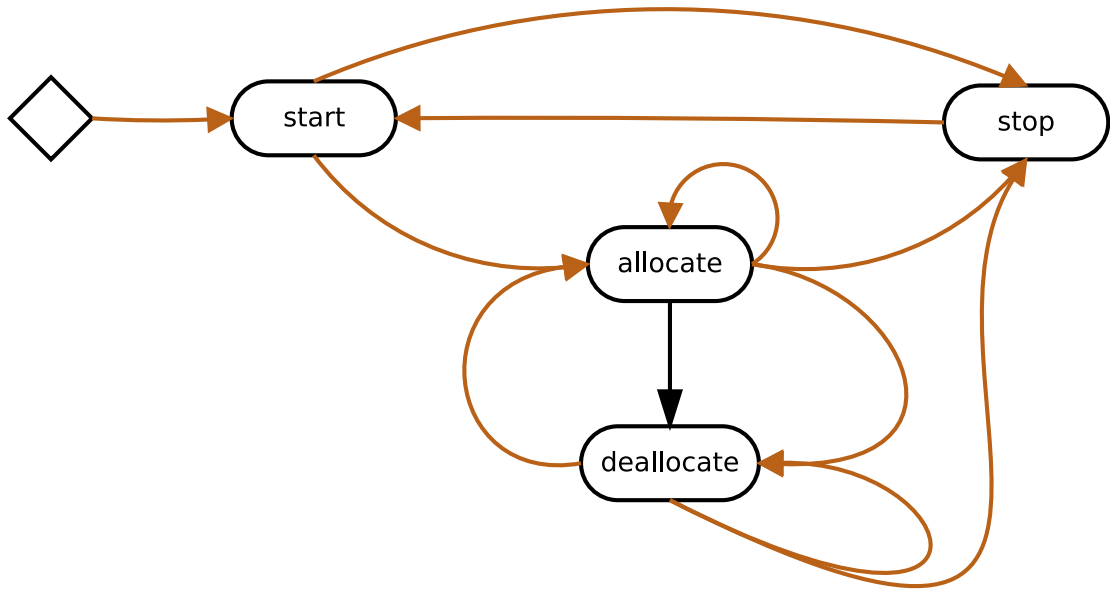


Figure 34: Frequency server state machine with data and control flow

the creation of tests, by automatically extending existing test suites. This can potentially augment the number of scenarios covered and reduce the effort required to create an arbitrarily large number of tests without having to design abstract properties manually. Because the procedure described in this chapter generates new tests by creating and using an FSM QuickCheck model (see Section 2.4.2 on page 23), this technique can also be potentially used as an aid for the migration from unit tests to property-based test models, since the models generated can be used as a template.

Through the mechanisms described in this chapter, we also try to address the problem of state explosion when we try to model systems that are composed of several subsystems. We achieve this by combining control flow and data flow in the same model, in contrast to most other work in this area.

For example, we already saw in Section 4.4.5 (on page 108) how increasing the number of frequencies near-doubled the number of states of an FSM inferred from the Frequency server. By adding data flow we can create a diagram that is independent of the number of frequencies (see Figure 34). We have coloured the traditional control transitions found in FSMs in brown, and we have added black arrows to represent data flow. In the example, data flow says that the output of allocate can be used as input to deallocate.

James The work described in this chapter has been implemented as the James tool, whose source code is publicly available (see reference: Lamela Seijas and Thompson 2014). James is a tool that aims at generating new tests for web services from existing JUnit tests. James transparently instruments the execution of existing JUnit tests and uses the information obtained to generate a visual model that can be rendered through the tool Graphviz (see Section 2.5 on page 24) and a QuickCheck FSM model (see Section 2.4.2 on page 23) that when executed produces new tests in the style of the tests provided as input.

The techniques described here and their implementation (the James tool) are both targeted to web services, since the nodes in the diagrams are grouped in subgraphs depending on the URL and method of the HTTP requests issued by the part of the tests they represent (see Section 5.2.2.2 on page 121). However, the main ideas presented here should be straightforward to apply to other types of API, (for example, the API of a dynamic library), as long as the system under test (SUT) is tested like a black-box and has a well defined interface. In fact, in Chapter 6 we study an inference algorithm, similar to the one used in this chapter, from a more formal perspective that does not make assumptions about the SUT being a web service.

Overview In the following sections, we enumerate the contributions of this chapter (Section 5.1), we explain our approach to instrumenting JUnit tests (Section 5.2), we describe the architecture of our implementation (Section 5.3), we provide a process for constructing models that combine data and control flow (Section 5.4) together with an example of its application to the Frequency server (Section 5.5), we overview how property-based models can be constructed to generate new tests (Section 5.6), we present the results of a pilot study that evaluates our approach and implementation (Section 5.7), we discuss the limitations of the approach (Section 5.8), we reflect on the lessons learned (Section 5.9), and in Section 5.10 we conclude.

5.1 Contributions

In this chapter, we show how existing unit tests can be leveraged to provide more testing value through the inference of a model from existing tests. We make four

specific contributions:

- We define a new approach to inferring an state machine model for a system, extended with data flow information, from an existing unit test suite and a working implementation of the system. The extended state machine combines both data flow and control flow information; existing approaches have tended to use just one of these.
- We show how to derive potential new test cases for the system under test automatically from the inferred model. New tests are generated from the model through the use of QuickCheck property-based testing (PBT) tool (see Section 2.4 on page 20).
- We provide a mechanism by which approximate QuickCheck models for Java systems can be inferred automatically, thus allowing the rapid development of PBT models from existing test suites. These models are approximate in the sense that they do not necessarily represent the precise behaviour of the system (not even for the scenarios provided as input to the inference algorithm), but they will usually present a similar or identical behaviour for some of the scenarios.
- We present a pilot study in which we apply our approach to generate new tests for an existing industrial system. Our work aims to extract models that represent both successful and failing behaviour of a target web service. The tests generated during the pilot study helped developers of the system find a previously unknown bug in the implementation.

The techniques described in this chapter have been implemented in the James tool. The source code of James is publicly available (Lamela Seijas and Thompson 2014).

The work in this chapter is based on the paper (Lamela Seijas, Thompson and Francisco 2016), the deliverable (Lamela Seijas, Thompson and Francisco 2012), Task 2.2 in (Arts et al. 2015), and a paper submitted for publication in a special issue of the Software Quality Journal. Part of the work described here has also been presented at UCAAT 2015 and at UKSCC 2015.

The pilot study in this chapter (including the second run) was carried out in collaboration with Miguel Ángel Francisco who also wrote the original report of

the pilot study in Task 2.2 of (Arts et al. 2015); excerpts of this report have been included in this chapter. In addition, both Miguel Ángel Francisco and Simon Thompson contributed with ideas, suggestions, guidance, advice, revisions, and editing. Laura M. Castro Souto also participated in the initial discussions about the design of James, and contributed with ideas and advice.

I contributed to this work with the code of the implementation of James, debugging and support during the experiments, most of the writing of the description of the system and techniques, and the summarisation and editing of the contents of the report about the pilot study.

5.1.1 Software contributions

As part of the work in related to this chapter:

- I implemented the whole of James, whose source is available at (Lamela Seijas and Thompson 2014). A web service version of the Frequency server is available in (Lamela Seijas 2014b) and partially in Appendix B.
- I wrote some examples of tests for the web service version of the Frequency server that were not used in the pilot study (Lamela Seijas 2014a). Note that there also exists a separate set of tests for the web service version of the Frequency server which were implemented by Interoud Innovation, whose source code is available in (Francisco 2014a) and also in Appendix C.
- I implemented a Java Erlang Bridge interface that was used to automatically classify the tests generated by James during the second run of the pilot study (Lamela Seijas 2014c).

5.2 Instrumentation

In this section, we study how our implementation (the James tool) gathers the information necessary to generate models from existing JUnit tests and a working implementation of the system under test.

5.2.1 Static and dynamic approaches

Most tools that extract information from existing software fall into one or both of two big categories: *static* and *dynamic*. In this section, we briefly discuss some of their advantages and disadvantages, and describe our preferred approach in this spectrum.

5.2.1.1 Static approaches

An approach is usually considered static if it analyses the software without executing it¹. One advantage of static approaches is that they usually gather information about software that is generic and relevant for any execution, since they do not rely on particular executions. On the other hand, there are some aspects of software that, for a static approach, are undecidable in the general case like, for example, termination (halting problem).

Static approaches usually analyse the source code of software, but in some cases they can target compiled versions of it instead (for example: bytecode or machine code). One advantage of targeting source code is that it tends to contain more information about the artefacts used by the developer, which may indicate high level intentions.

Unfortunately, the number of mature libraries that are available for Java source manipulation, and support the whole language, is small. The most popular ones focus on bytecode, which may already lose some information² about the structure of the code (due to optimisations), comments, and potentially variable names. Nevertheless, in the case of Java, little information is lost by compiling (specially if debugging information is enabled at compilation time).

5.2.1.2 Dynamic approaches

Dynamic approaches analyse the way in which a working piece of software executes by introducing logging mechanisms or by probing it during its execution.

Having to execute the code has the advantage of providing (and the disadvantage of requiring) real values for parameters and variables. In the case of

¹https://en.wikipedia.org/wiki/Static_program_analysis [last accessed 14-08-17]

²<https://stackoverflow.com/questions/30262635/is-any-information-lost-when-compiling-to-a-class-and-decompiling-back> [last accessed 14-08-17]

deterministic unit tests, a single execution will often reveal all the scenarios that are being tested. And these tests provide a valid starting point for the creation of new ones.

On the other hand, using a dynamic approach requires a working implementation. The requirement of having a working implementation in a test generation tool prevents its applicability from partial systems and test-driven development, since they require tests to be created before a working implementation exists. But this problem can be addressed through the use of mocking (Svenningsson et al. 2014).

5.2.1.3 Our approach

For this work, we have chosen a mainly dynamic approach that relies on the JVM Tool Interface (or JVMTI see Section 3.2.4 on page 31). JVMTI allows tools that use it to get information at execution time and even modify the execution. Nevertheless, some information about the source code can still be obtained through the API of JVMTI, like the names of methods and classes, and (by combining it with the use of reflection) the information about annotations (see Section 5.3.2 on page 122).

5.2.2 Data and control flow

The James tool extracts and combines both data and control flow information into a single model.

5.2.2.1 Data flow

Data flow (Rapps and Weyuker 1985) represents how variables are bound to values, and how these variables are to be used. Existing JUnit tests provide concrete examples of how data is used and reused in concrete scenarios. By modifying or generalising these scenarios, it is likely that we will find new meaningful aspects to test, some of which may not be tested by the existing test suite.

Even in cases when the modifications to data produce errors, these modifications may still be useful as “negative test cases”, since almost-valid input can help us find corner cases (Tsankov, Dashti and Basin 2013). For example, if, when serialising a request, the unit tests explicitly add quotes surrounding a value, then

the generalisation of the request generation may add quotes in places where they should not be. This kind of input would potentially help highlight problems like SQL injection and, thus, improve the security of systems. The application of specific knowledge about concrete vulnerabilities can be used to alter inputs in ways that increases the likelihood of finding particular security problems even further (Kieyzun et al. 2009).

James registers data flow by tracking all the objects that are referenced by the unit tests and by linking together the methods or functions that produce them as a result with the ones that take them as parameters. This way we obtain information about the way in which objects are constructed and used. To be precise, we must point out that, in addition to the objects that are directly referenced by the unit tests, we also need to track and include in the model those objects that are required to create objects that are referenced by the unit tests, and so on (that is: the transitive closure of references).

In the case of tests targeted at web services, requests are usually composed out of small pieces of information, like numeric values or dates, which are combined into bigger structures and then serialised, or directly embedded into templates.

In the same way, responses to requests may be unmarshalled, and the small pieces that compose them may be checked for correctness through the use of `xUnit assert` functions.

5.2.2.2 Control flow

Control flow represents how events affect the state of the system. In particular, we are interested in the state of the target web service and, thus, we track and model only the control flow of methods that produce HTTP requests, since they are the only ones that can possibly affect the state of the web service. Even though we do not include in the model methods that are not directly called from the tests, we analyse every method entry and exit, thus, methods that (directly or indirectly) call methods that produce HTTP requests are also detected.

In order to represent control flow, we link the events (the methods that produce HTTP requests) in execution order, and links are preserved during the merging process.

We explain how we identify methods that produce HTTP requests in Section 5.8.3 (on page 156).

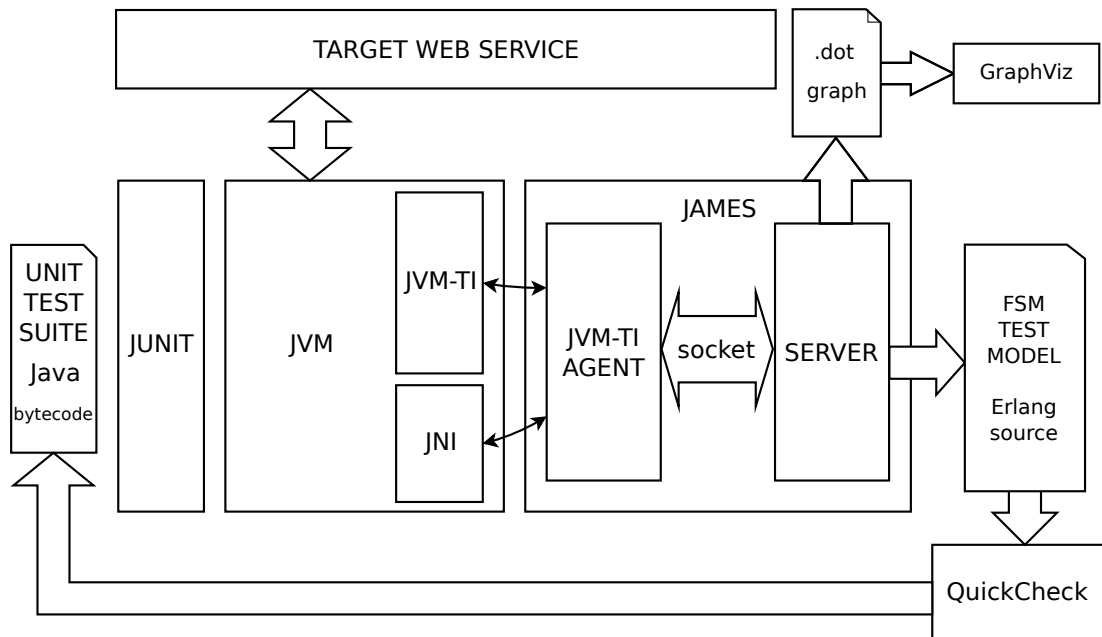


Figure 35: Architecture of James

5.3 Architecture of the approach

In Figure 35, we show the architecture of the James tool and the way it interacts with the rest of the systems during instrumentation and model creation.

5.3.1 JUnit and JVM

The system takes as input a set of JUnit tests. JUnit tests are Java code that uses the library JUnit and, like any other Java code, they are compiled into bytecode and executed by the Java Virtual Machine (JVM). JUnit tests taken as input are also expected to use a Java library (for example, a socket library or a web client library) to connect to the web service under test, issue the appropriate requests, and check that the responses are as expected.

5.3.2 JVMTI agent and JNI

James instruments the execution of the JUnit tests through the use of its JVMTI agent, which is a dynamic library written in C++. The JVMTI agent uses the JVMTI (JVM Tool Interface) and JNI (Java Native Interface) APIs to track the

method entry events, method exit events, and every object that is taken as a parameter or returned by the methods executed.

In order to avoid having to ask the user, James detects where the tests are automatically. This is achieved by inspecting methods called for JUnit annotations (that is: `@Before`, `@After`, `@Test`) through the use of reflection, accessed through the JNI. This information is also used for detecting whether methods are part of the set-up, the clean-up, or the actual tests. Since checking classes for annotations at each method call slows down the process considerably, we store in a cache the names of all those classes that do not contain JUnit annotations. In addition, the JVMTI agent also detects those methods that issue HTTP requests (see Section 5.8.3 on page 156).

All this information is sent through a socket to the James server, which is written in Erlang.

5.3.3 Erlang server

The process of instrumentation produces a long list of method calls, most of which usually do not belong to the tests themselves, but to frameworks (such as the Apache Ant library), or to the JVM itself.

The Erlang server filters most of the calls that do not belong to the tests, by using the information about JUnit annotations gathered by the JVMTI Agent. We also use the annotations to distinguish between method calls that belong to the *set-up* and *clean-up* procedures, and the actual test *body*.

Calls that produce objects used in the tests, even when these are not part of the tests themselves, must be tracked too, otherwise James may not have information to create those objects when the new tests are generated.

5.3.4 GraphViz and QuickCheck models

Once all necessary information from the execution has been gathered, the Erlang Server can use it to produce a model. This model can be rendered in two ways, graphically (through the use of dot, part of GraphViz, see Section 2.5 on page 24), or as a property-based test model (that uses the `eqc_fsm` module of QuickCheck, see Section 2.4.2 on page 23). Both models are generated similarly and the initial parts of the process of generation are identical. In fact, the current version of

James is able to generate simultaneously both models and to create hyperlinks between the GraphViz and the QuickCheck version.

5.3.4.1 GraphViz

The GraphViz model is a graph representation that can be visualised. It contains most of the information of the model and can be used for documentation and to check visually for problems in the tests, the system, or in the instrumentation process itself. One example of the GraphViz rendering of a model generated by James can be seen in Figure 42 on page 140.

5.3.4.2 QuickCheck FSM

The QuickCheck `eqc_fsm` models generated by James, when executed, will print out JUnit tests that are similar to the ones used as input. `eqc_fsm` models generated by our implementation do not issue requests towards the target system directly, we decided to print the tests generated instead since they are easier to debug this way, but they can potentially be used as a starting point for migration to property-based testing.

Part of one example of a QuickCheck `eqc_fsm` model generated by James from the Frequency server can be found in Appendix D.

5.3.5 Feedback

Since the QuickCheck `eqc_fsm` model produces new JUnit tests, we can add those (after manually reviewing them) to the initial set of tests used as input, and then we can restart the cycle and potentially obtain a refined version of the model.

This process is limited by the lack of soundness of our inference algorithm. It is possible that at some point, the algorithm is not able to learn any more. We study a way to restore soundness in Chapter 6.

5.4 Model construction

In this section, we describe how the information gathered is used to generate a model. We illustrate the explanation with examples of graphical diagram representations generated by James through GraphViz. In Section 5.5 (on page 136),

we analyse in detail a diagram generated for the Frequency server web service (source-code available in Lamela Seijas 2014b and partially in Appendix B) by using as input a set of tests provided by Interoud Innovation (available in Francisco 2014a and Appendix C).

Later in Section 5.6 (on page 139), we explain how test generation is achieved by relying on the models and ideas described in this section.

5.4.1 Common flow graph

James starts by generating a graph with the calls to methods that were executed directly from the tests provided as input, these calls are represented as square-boxed nodes (see Figure 42 on page 140). Later in this section we will explain how the nodes in the graph are connected.

In principle, we only represent the level of abstraction expressed by the tests; the model uses the calls that are issued directly from the body of the tests, the calls done elsewhere are not considered (as long as they do not provide values used by calls that are issued directly from the body of the tests).

For example, in the following snippet, the only method calls that would have nodes representing them in the model would be `allocateFrequencyResponse` and `checkNotRunningError`; `assertEquals` is also called but not directly from the tests, so it is hidden by the level of abstraction in which the tests are written:

```
@Test
public void testAllocateNotStarted() throws IOException {
    FreqServerResponse allocateFrequencyResponse = allocateFrequency();
    checkNotRunningError(allocateFrequencyResponse);
}

private void checkNotRunningError(FreqServerResponse response) {
    Assert.assertEquals(ERROR_RESPONSE, response.getState());
    Assert.assertEquals(1, response.getError().size());
    Assert.assertEquals(ERROR_TYPE_NOT_RUNNING,
        response.getError().get(0).getErrorType());
}
```

The previous code would produce the diagram in Figure 36, where we can see that `assertEquals` does not appear (the `stopServer` node comes from the clean-up code which we do not show). The advantage of this way of abstraction is that it

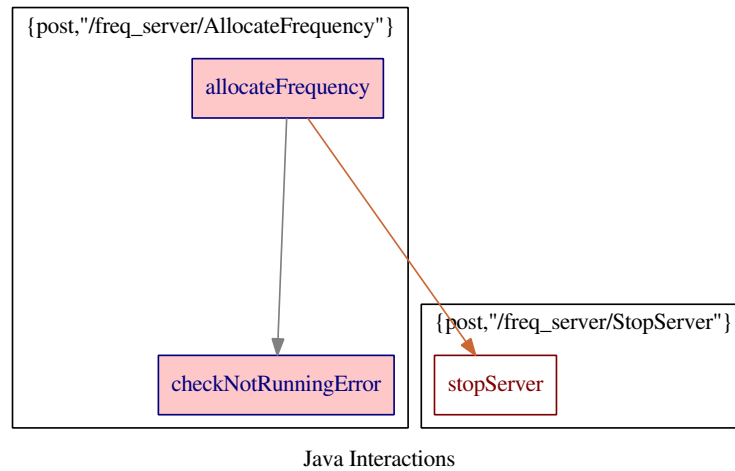


Figure 36: Example of abstracted out code

has already been manually chosen by the testers to represent the scenarios that are being tested (whoever wrote the tests used as input is indirectly choosing the level of abstraction for the diagrams). This feature also allows users to remove low-level details from the model by abstracting them out through function extraction.

But we still try to include calls that are necessary to provide values needed by other calls already included (independently of their level). For example, in the following snippet, the function `getFrequencyAllocated` is not called directly from the tests, but its result is made accessible and used by them, so the call that produces it (`getFrequencyAllocated`) is still included in the diagram as shown in Figure 37 (on page 128):

```
Integer freq;
```

```
@Test
```

```
public void testHypothetical() throws IOException {
    FreqServerResponse startServerResponse = startServer();
    checkNoErrors(startServerResponse);
    FreqServerResponse allocateFrequencyResponse = allocateFrequency();
    extractFrequency(allocateFrequencyResponse);
    FreqServerResponse deallocateFrequencyResponse = deallocateFrequency(freq);
    checkNoErrors(deallocateFrequencyResponse);
}
```

```
public void extractFrequency(FreqServerResponse f) throws IOException {
```

```
    freq = f.getResult().getFrequencyAllocated();  
}
```

In practice, in the implementation available online at the time of writing, and in the version of James used for the pilot, these hidden calls are not always included because James is configured to treat traces of depth greater or equal to two differently. This modification was done to improve speed of execution of instrumented tests, and to avoid problems when trying to get information of native Java methods, but it can be easily tweaked if needed. The diagrams in Figures 36 and 37 were indeed generated by James automatically from the code provided, after adjusting these parameters.

The nodes are connected using two types of arrows:

- For *data flow*, gray arrows connect the methods that produce a result (that is: those that return a value or an object) with those that take the result as a parameter, or those that use the result as a base object, that is: those methods that are called “on the object returned”, (the object that can be referenced through the keyword `this` from the code of the method). Base object usage is represented with dashed gray arrows.
- For *control flow*, brown arrows connect methods that issued HTTP requests, in the order they were called during the execution of the input tests.

A detailed legend of the different notations used in the GraphViz graphs generated by James can be found in Table 3 on page 137 and Table 4 on page 138.

5.4.2 Merging process

The process described so far already generates a model, but it is usually too dense to be useful, because it has too many nodes and arrows, which makes it difficult to understand; and it does not generalise the scenarios obtained, because it only represents the scenarios presented by the set of JUnit tests used as input.

The merging process tries to generalise and simplify the graph while keeping the aspects of the model that give actual information about the general behaviour of the target system (as opposed to specific behaviour in response to specific input). This generalisation is achieved by merging paths with the same topology, similarly to how it is done by the k-tails algorithm (see Section 2.3.2 on page 20).

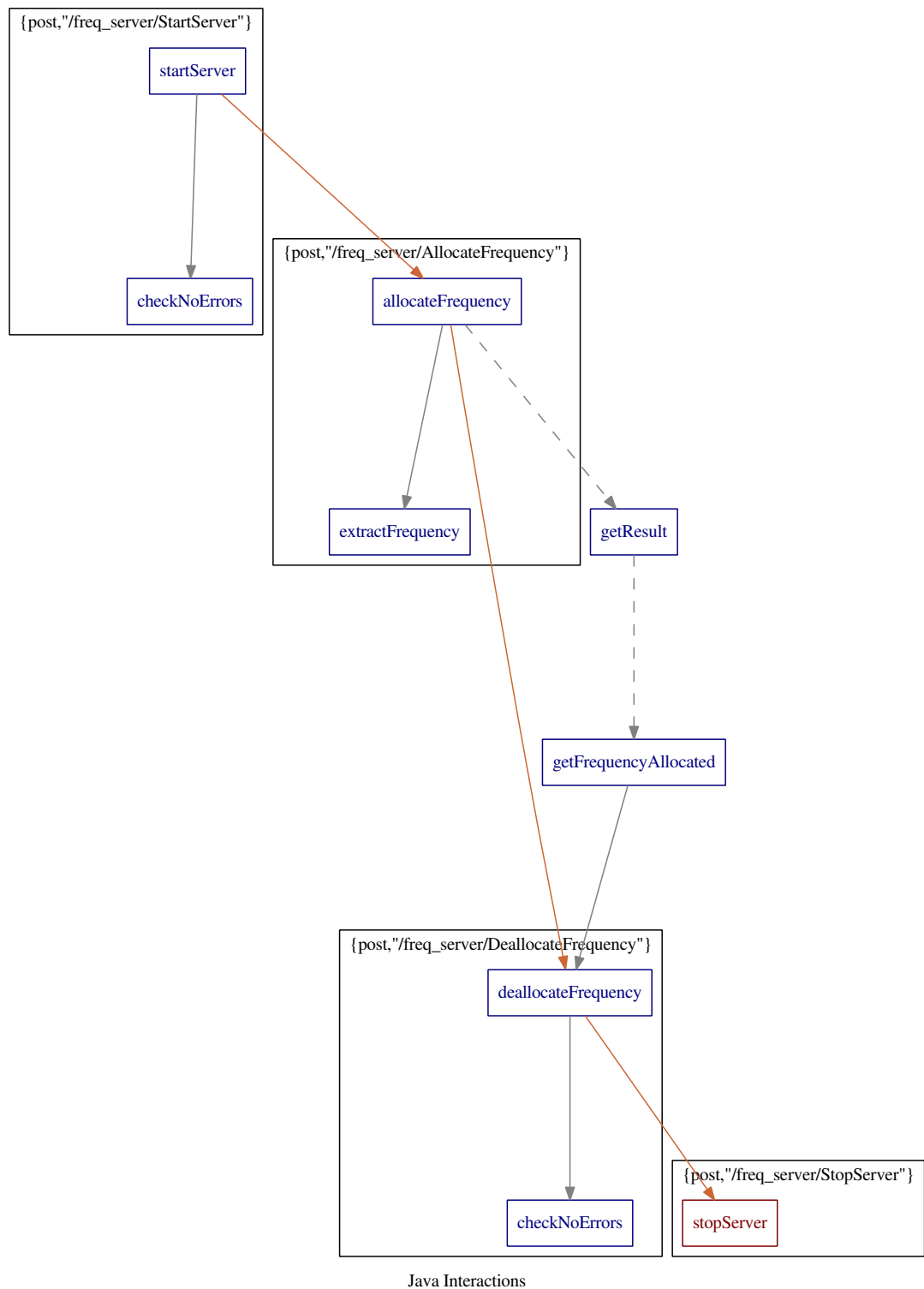


Figure 37: Example of external code included because of dependencies

James searches every subtree in the graph, alternately following the arrows directly and in reverse. Then it merges the subtrees that contain pairs of methods with the same topology both in data and in control flow.

Longest subtrees are merged first, down to a minimum length (upper K). All tails of the graph (leaf and root nodes) are allowed a lower bound (lower K); the intuition behind “lower K” is that if a pair of longest matching subtrees is delimited by the end of the graph (has leaf or root nodes), it may be that the lack of commonalities between both subtrees is due to their small sizes, rather than to the “states” that they represent being different.

The usage of a constant bound (parameter K) is inspired by the K-Tails algorithm (see Section 2.3.2 on page 20). If we imagine all the possible executions of a system as a tree, in which each possible input is a different branch, and where each node represents a different state of the system, we can consider two states to be equivalent (i.e: the same state) in the tree if their subtrees are isomorphic, in other words, if the possible executions of the system starting in both states are the same. In practice, it is not feasible to explore all possible executions of a system every time we want to decide whether we want to merge two nodes; it is also impossible if the number of possible executions is infinite. The parameter K decides how far in the search space we will check before concluding that two states are equivalent. If we set K to zero, every node will be merged; the bigger the value of K is, the more conservative the merging process is.

Our merging algorithm uses the idea of bounding the search length with a parameter to decide which nodes to merge too; but there are three main differences with the original K-Tails algorithm. First, we do not start with a tree but with a graph, we have more than one “root node”; in our early experiments, we found that comparing only the descendants of each node did not produce good results, thus, our algorithm checks the equality of descendants and ascendants alternately. Second, we do not have only control flow, but also data flow, and we use the same parameters lower K and upper K to limit the exploration of both. Lastly, our nodes have labels, they do not represent states but “method calls”, thus, even for a lower and upper K of 0, two nodes will never be merged if they have labels (e.g: they call a method with a different signature). The default values for K in James (at the time of writing) are: 4 for upper K, and 1 for lower K.

In order to give a sense of the effect of K, in Table 2 (on page 131), we present

statistics of the number of nodes and arcs resulting from running James using different values of lower and upper K using the data obtained in the second run of the pilot study. In principle, as we said, increasing either of the lower and upper K parameters leads to a denser diagram, but there are a series of filters to the process that may skew the distribution depending on the resulting topology of the graph. For example, merging a pair of nodes may actually result in more nodes since extra `oneOf` nodes may have to be created (as described in the legend for `oneOf` nodes in Table 3 on page 137).

In Figures 38, 39, 40 and 41 (on pages 132, 133, 133 and 134 respectively), we present an approximate pseudocode description of the merging algorithm.

The `merging_algorithm` function in Figure 38 (on page 132) shows how we first search for pairs of subtrees longer than upper K, and if we fail we search for subtrees longer than lower K but with the requirement that both subtrees in the pair must be maximal (represented by the boolean taken as last parameter by `find_best_pair` and `find_best_pair_rec`). The actual merging of isomorphic subtrees (carried out in the pseudo-code by the function `merge_pair`) is basically done by taking each pair of one node in one side of the isomorphism and its image, moving all the incoming and outgoing arrows from one of the nodes to the other and deleting the orphan node. Nevertheless, `oneOf` nodes may have to be created to group the data or control flow arrows corresponding to the different parameters (as described in the legend for `oneOf` nodes in Table 3 on page 137).

In Figure 40 (on page 133) we present a possible way of storing the abstract data type `tree`. The data type `tree` represents a subtree in the model graph that we explore by using breath first search; because the graph may have loops, we will not explore those nodes that are already in a higher (closer to the root) level of the subtree. Each subtree also has a direction (either “upwards” or “downwards”, which specifies whether they will follow the arrows directly or inversely when expanded). Expanding a subtree will take all the leaf nodes (the ones in the last level) and follow the arrows that are incoming or outgoing (depending on the direction), the nodes at the other end of the arrows will become the new last level of the subtree (except those that are in the subtree already).

The `find_best_pair` function in Figure 39 (on page 133) implements the initialisation phase of the algorithm for finding candidate trees to merge. First, it creates two singleton subtrees for every node (one with direction “upwards” and

Upper K Lower K	1	2	3	4	5	6	7	8	9	10
1	145/264	149/274	150/281	151/285	152/284	152/284	152/284	151/285	151/285	151/285
2	-	172/322	175/341	171/341	176/350	176/350	176/350	171/340	171/340	171/340
3	-	-	394/775	430/854	435/879	435/879	435/879	434/876	434/876	434/876
4	-	-	-	502/952	507/985	507/985	507/985	510/992	510/992	510/992
5	-	-	-	-	510/992	510/992	510/992	510/992	510/992	510/992
6	-	-	-	-	-	497/979	497/979	497/979	497/979	497/979
7	-	-	-	-	-	-	496/980	496/980	496/980	496/980
8	-	-	-	-	-	-	-	483/970	483/970	483/970
9	-	-	-	-	-	-	-	-	483/970	483/970
10	-	-	-	-	-	-	-	-	-	483/970

Table 2: Effect of lower and upper K on 2nd run of pilot study (nodes/arcs)


```

void merging_algorithm(Int bigK, Int smallK, Graph model) {
  while (true) {
    Maybe<Pair<Tree, Tree>> best_pair :=
      find_best_pair(bigK, model, false)
    if (best_pair == Nothing) {
      best_pair := find_best_pair(smallK, model, true)
      if (best_pair == Nothing) {
        return
      } else {
        model.merge_pair(best_pair.getContents())
      }
    } else {
      model.merge_pair(best_pair.getContents())
    }
  }
}

```

Figure 38: Merging algorithm base function pseudo-code

one with direction “downwards”). Then we search for the longest pair of equivalent subtrees and (if they are at least `min_tree_depth` deep) we return them for the `merging_algorithm` function to merge them.

The `find_best_pair_rec` function in Figure 41 (on page 134) implements the algorithm that finds the best candidates to merge. It iteratively expands the subtrees and filters the unique ones, until there are no more subtrees. When this happens we recover the last batch of surviving subtrees and choose one of the deepest. When in `strict_mode` we also need to ensure that the remaining subtrees are maximal.

Inspired by the algorithm Blue-Fringe (that relies on the classification of traces into positive and negative for the detection of equivalent states, see Section 2.3.1 on page 18), we classify methods that issue HTTP requests into “normal” and “erroneous” according to whether they produce values used by method calls whose name contains the keywords `error` or `fail`.

We combined ideas from Blue-Fringe and k-tails because we start with a graph instead of a tree (like a PTA or APTA), so it was not clear in principle that Blue-Fringe would prevent the graph from over-merging (over-generalising). Because we cannot merge nodes with different names (to be precise nodes that have methods with different signatures), it turns out that often it is not necessary to specify a minimum bound (`K` parameters) in order to obtain a useful diagram.

```

Maybe<Pair<Tree, Tree>> find_best_pair(Int min_tree_depth,
                                       Graph model,
                                       Boolean strict_mode) {

  List<Node> node_list := model.get_nodes()
  List<Tree> tree_list := [];
  for each node in node_list {
    tree_list.add(create_tree_starting_in(node, "upwards"))
    tree_list.add(create_tree_starting_in(node, "downwards"))
  }
  Maybe<Pair<Tree, Tree>> best_pair :=
    find_best_pair_rec(tree_list, model, strict_mode)
  if (best_pair == Nothing) {
    return best_pair
  } else if (best_pair.getContents().first().tree_depth >= min_tree_depth) {
    return best_pair
  } else {
    return Nothing
  }
}

```

Figure 39: Equivalent subtree initialisation pseudo-code

```

struct Tree {
  Node[] [] nodes_in_each_level, // except loops
  Node nodes_reached,           // union of nodes_in_each_level
  Int tree_depth,                // number of levels of the tree
  Direction direction            // one of "upwards" or "downwards"
}

```

Figure 40: Example subtree data type pseudo-code

```

Maybe<Pair<Tree, Tree>> find_best_pair_rec(List<Tree> tree_candidates,
                                           Graph model,
                                           Boolean strict_mode) {

    List<List<Tree>> grouped_tree_list :=
        group_isomorphic_trees(tree_candidates);
    List<List<Tree>> repeated_tree_list :=
        filter_out_unique_trees(grouped_tree_list);
    if (repeated_tree_list IS EMPTY) {
        return Nothing
    } else {
        List<Pair<Tree, Tree>> next_level = [];
        for each tree_list in repeated_tree_list {
            List<Tree> tree_list_copy = clone(tree_list)
            for each tree in tree_list_copy {
                expandTree(tree)
            }
            Maybe<Pair<Tree, Tree>> best_pair :=
                find_best_pair_rec(tree_list_copy, model, strict_mode)
            if (best_pair != Nothing) {
                next_level.add(best_pair.getContents())
            }
        }
        if (next_level IS EMPTY) {
            if (strict_mode) {
                // We remove those trees that can be expanded
                remove_trees_not_maximal(repeated_tree_list)
            }
            if (repeated_tree_list IS EMPTY) {
                return Nothing
            } else {
                return Something(get_first_pair_of_trees(
                    get_sublist_with_deepest_trees(
                        repeated_tree_list)))
            }
        } else {
            return maybe_deepest_tree_pair(next_level)
        }
    }
}

```

Figure 41: Equivalent subtree search pseudo-code

One difference between models generated by James and FSMs is that the algorithm in James stores events (in our case method calls) in the nodes, as opposed to normal regular state machines that often store events in transitions and nodes represent states (this is also one difference with the work in Chapter 6, where we store events in transitions, and nodes represent states). Models produced by James do not have transitions as such, they are implicit in the control flow, methods in the nodes can be seen as producing a transition in the system when they are executed. This is similar to the difference in the way Mealy and Moore machines encode output data: Mealy machines output is associated to inputs and states and visual representations usually include output as part of the transitions, whereas Moore machines encode output in terms of the state only, and visual representations usually include the output as part of the state; see (Mealy 1955) and (Moore 1956).

We decide whether a node that issues an HTTP request is expected to produce an error by looking at the nodes that receive its data flow because usually these nodes contain assertions that check whether the response is an error message, so the type of assertion determines the type of the method (this can be observed in the example shown in Figure 42 on page 140). If we used a proxy, we could detect errors by looking at the status code of responses to HTTP requests (since responses whose status code starts with 4 or 5 represent client and server errors respectively). We could potentially obtain this information through the JVMTI agent but, like when obtaining the target URLs for requests, the solution would probably be dependent on the specific approach used by the tests to connect to the web service (the particular library used).

In addition to the constraints described before, nodes labelled as “normal” are never merged with nodes labelled as “erroneous”, data arrows are never merged with control arrows, and data arrows are never merged with other data arrows that provide values for a different parameter. Two parameters are considered different if they have different positions (in the list of parameters/arguments) or because they have different Java type.

Since we merge only subgraphs of a minimum depth, it is likely that all the sequences merged have the same or similar semantics. The merging process produces new connections and loops between nodes, both in the data and the control flow.

In order to make the diagram clearer, we group together, in the same subgraph, methods that issue HTTP requests to the same URL and with the same HTTP method. Nodes that hang from these nodes and do not receive values from methods that produce different HTTP requests are also included in the same subgraph. We include these nodes too because, in our experience, they tend to be related (they are the ones that unmarshal the result or check that the results are as expected).

5.5 Example

In this section, we discuss in detail the result of applying the model inference process of James to a toy example test suite: a set of JUnit tests targeted at a web service version of the Frequency server (see Section 2.1.5 on page 15). I implemented the web service version of the Frequency server in Java; the full source-code is available at (Lamela Seijas 2014b) and partially in Appendix B. The JUnit tests used as input were developed by an independent party (Interoud Innovation), and can be found in Appendix C and online at (Francisco 2014a).

The full graphical diagram extracted by James is presented in Figure 42 (on page 140), and part of the QuickCheck `eqc_fsm` model extracted by James is provided in Appendix D.

5.5.1 Interpreting the model

With a quick look to the model (Figure 42 on page 140), we can see that there are four different types of requests (grouped in four subgraphs): `allocateFrequency`, `startServer`, `deallocateFrequency`, and `stopServer`. These correspond directly to the four operations that the Frequency server accepts (see Section 2.1.5 on page 15).

We can see that all of them have their normal version (white background) and their failing version (pink background). Additionally, both `deallocateFrequency` and `stopServer` have a version with red outline, since they are used in the clean-up procedure.

Even though this is not displayed in the diagram, we can deduce that the normal `startServer` method is initial (it can be the first to be called), because it



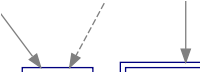



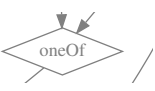
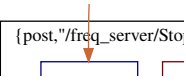

	<p>Negative instance classes</p> <p>Calls with keywords like “error” or “fail”, and calls that produce values used by them are considered erroneous, and painted pink.</p>
	<p>Methods, @Test, @Before, and @After</p> <p>The outline of rectangular nodes represents the places where the command was found, see Table 4 on the next page. A single node may appear in several places since it may be the result of merging several nodes.</p>
	<p>Arrows</p> <p>Data flow is represented with grey arrows. Arrows connect the methods that produce an object as result with methods that take it as a parameter.</p>
	<p>Dashed Arrows</p> <p>Whenever an object produced as result of a method is used as base object of another method, (that is: the <code>this</code> object of the method), the data flow relationship is represented with a dashed grey arrow.</p>
	<p>Brown Arrows</p> <p>Control flow is represented through brown arrows. These are created following the order in which the methods were originally executed in the input unit tests.</p>
	<p>Loop highlighting</p> <p>Loops in control flow are represented with thicker arrows.</p>
	<p>oneOf diamonds</p> <p><code>oneOf</code> nodes group together several converging arrows of the same type that provide alternative data or control flow (that can be used interchangeably). Because a method may take several arguments, we use <code>oneOf</code> nodes to distinguish between the data flow arrows for each different argument.</p>
	<p>HTTP request grouping (subgraphs)</p> <p>Methods that are related to an HTTP request targeted at a same URL and that use the same HTTP method are grouped in subgraphs surrounded by a black rectangle. The tuple in the rectangle denotes the method and URL used.</p>
	<p>Double outline</p> <p>Static methods are denoted with double outline. Thus, methods with double outline should not have incoming dashed arrows.</p>

Table 3: Diagram symbol legend









@Before	@Test	@After	Outline colour
No	No	No	Grey 
Yes	No	No	Green 
No	Yes	No	Blue 
No	No	Yes	Red 
Yes	Yes	No	Teal 
No	Yes	Yes	Purple 
Yes	No	Yes	Yellow 
Yes	Yes	Yes	Black 

Table 4: Colour legend for method nodes outline

does not have incoming brown arrows (control flow), but this is not a necessary condition, there may be other initial states (internally, information about initial states is stored, but this is not displayed in the diagrams).

We can also see that there is data flow from the normal `allocateFrequency` method to the `deallocateFrequency` methods, this represents that the result of `allocateFrequency` can be used as a parameter to `deallocateFrequency`. This can be seen more clearly in the detail show in Figure 43 (on page 141).

In Figure 43 (on page 141), we can see that it is possible to extract the result of the call `allocateFrequency` by calling `getResult` and then `getFrequencyAllocated`, and this value can be passed directly as a parameter to `deallocateFrequency`. But if we pass the frequency to `deallocateFrequency` a second time it will produce an error (highlighted by the pink background of `deallocateFrequency`). The model also tells us that another way of producing a deallocation error is by using the integer 0 as parameter for `deallocateFrequency`, this is always true because the integers used internally to represent frequencies are bigger than 10.

At this point, the reader may wonder how does the model know whether the frequency used in the first `deallocateFrequency` is the same as the one used in the second `deallocateFrequency` if `allocateFrequency` was called several times. Unfortunately, the models produced by James cannot represent this difference, which is one of the reasons why they lack soundness, we study a possible solution to this problem in Chapter 6.

5.5.2 The model is more general

One example of how the model is more general than the input tests is made obvious by the fact that our web service implementation of the Frequency server has a limit on the number of frequencies that can be allocated at the same time, but this limit was not explored by the existing unit tests. An implementation that allowed an arbitrary number of frequencies to be allocated would still pass the tests, despite its behaviour being different.

Nevertheless, a random test generator (see Section 5.6) that would randomly traverse the control flow of our model (shown in Figure 42) could try to allocate enough frequencies to do so, since there exists a control loop around the allocation command. At some point the server would return an error and the assertion `checkNoErrors` would fail.

5.6 Test generation

Using the approach presented in Section 5.4 (on page 124), we are able to build a comprehensive model of a system from a set of JUnit tests. Assuming that the tests make a sensible exploration of the SUT, then it is possible, not only to construct a graphical model of the system (as shown in Section 5.5 on page 136), but also to construct a QuickCheck FSM model for the SUT (see Section 2.4.2 on page 23) that will generate new tests for the system when executed.

In Section 5.5, we have seen examples of how some possible sequences of method calls are represented in the graphical model. The QuickCheck FSM models generated by James are analogous to the graphical ones, but represent the information programmatically. In this section, we describe how the elements of the model (and the graphical representation) correspond to the elements of QuickCheck FSMs generated by James.

5.6.1 Building a `eqc_fsm` model

A QuickCheck `eqc_fsm` model can be built by translating the different elements of the model or diagram:

1. The state transitions of the QuickCheck `eqc_fsm` model can be defined to

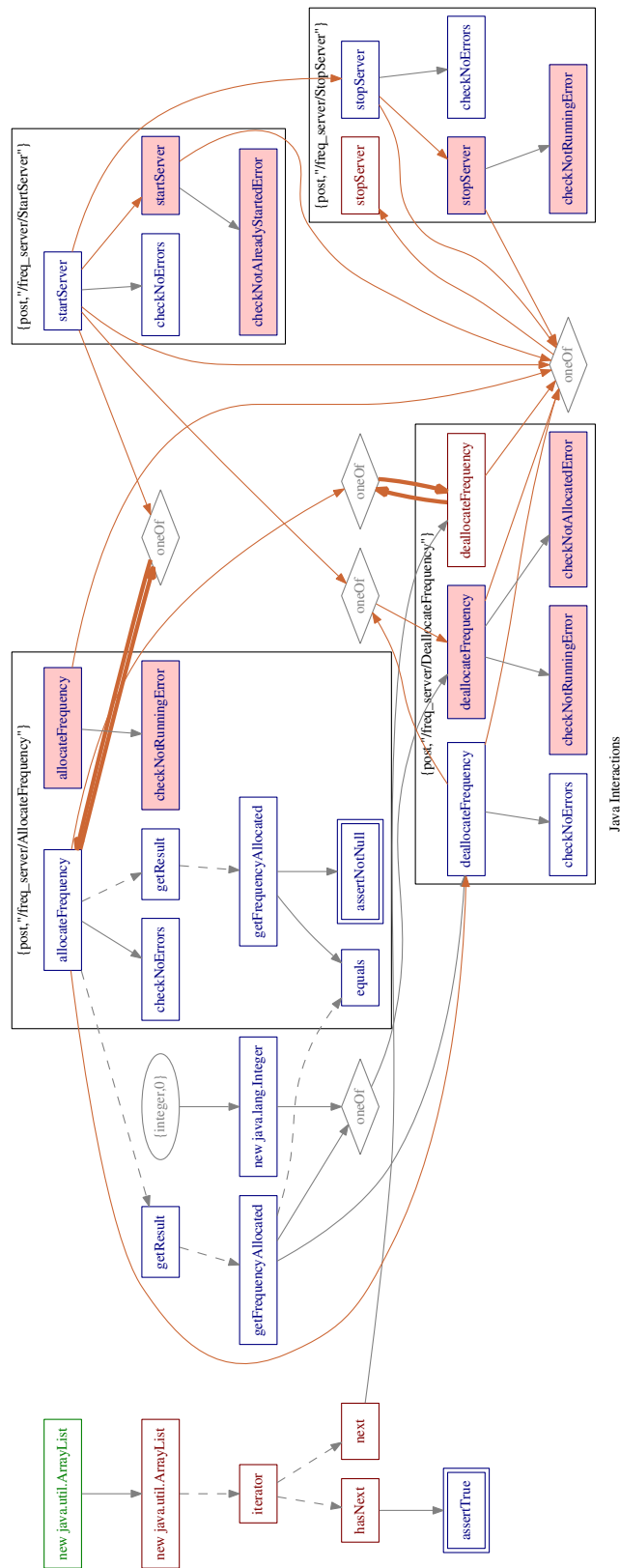


Figure 42: Diagram extracted by James from the Frequency server

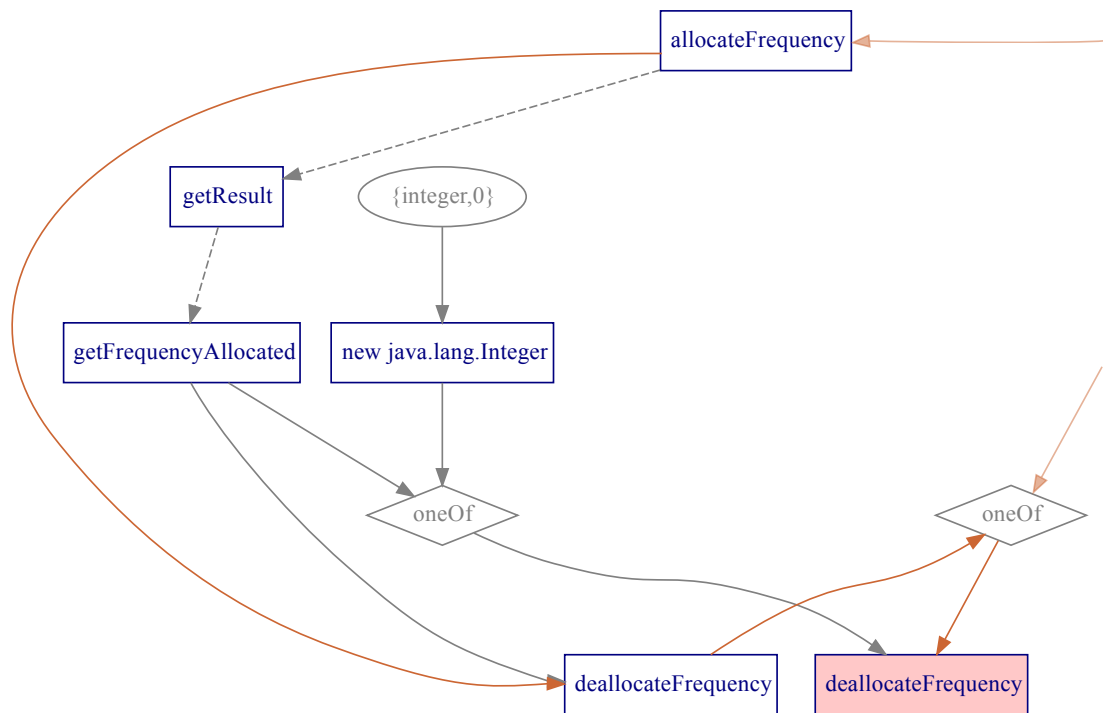


Figure 43: Slice of diagram displaying exceptional behaviour

match the control flow (including looping behaviour), given by the brown links in the visualisations.

2. Data flow indicates how we can create generators that call each other (possibly with recursion), and how the values produced by these generators can be used as parameters for calls in the control flow.
3. The combination of data flow and control flow gives an indication of the values that need to be stored as part of the state data of the QuickCheck `eqc_fsm` model. For example, Figure 43 shows how the result of an invocation of the method `allocateFrequency` can be stored in state data in order to be reused as a parameter for a posterior invocation of the method `deallocateFrequency`.

Our implementation preemptively stores results for each method call previously executed, and it obtains the values required by each method by randomly choosing between reusing the results already generated or by generating new ones by issuing the required calls to the appropriate methods.

4. Similarly to the way values required are generated, we include generators for postconditions that follow data flow directly within each subgraph. These will produce the postconditions in terms of the result of the method executions from the control flow.

In order to guarantee termination of the generators, we must bind their recursion with a strictly decreasing number. This can be done by computing, for each node, the minimum depth (distance to the top of the graph), and by ensuring that we eventually force the data generation process to follow a path with a strictly decreasing depth.

In methods with several parameters, the depth must include the minimum depths for all parameters (since a path that decreases the depth of the data flow path required to generate of one parameter may increase the depth of the data flow path required to generate another parameter).

For example, in Figure 44, we can see that `produceHeat` is a constructor and, thus, has depth 0; we could then conclude, if we calculate depth as the minimum, that `eggHatches` has depth 1, and that `chickenLaysEgg` has depth 2, whereas `protoChickenLaysEgg` has obviously depth 3. If we use this way of computing depths to guide test generation, we will arrive to the wrong conclusion that if we want to find a finite way of constructing an “egg”, the choice marked by the `oneOf` node should resolve to `chickenLaysEgg`, which would cause an infinite loop in the process. This problem illustrates that the depth of both `chickenLaysEgg` and `eggHatches` nodes should actually be established as infinite.

It is worth noting that the problem may not have a solution if we consider an arbitrary model. But the fact that the model was extracted from an actual execution guarantees that there exists a solution, since the values were indeed created somehow in the unit tests provided as input (this is one advantage of using a dynamic approach).

For this particular example, James created the following data generator for the `oneOf` diamond:

```
args_for(Size, _WhatToReturn, State, "diamond13o15") ->
  ?LAZY((oneof([args_for_op(Size, return, State, "11")] ++
               [args_for_op(Size - 1, return, State, "17")
                || Size > 0])));
```

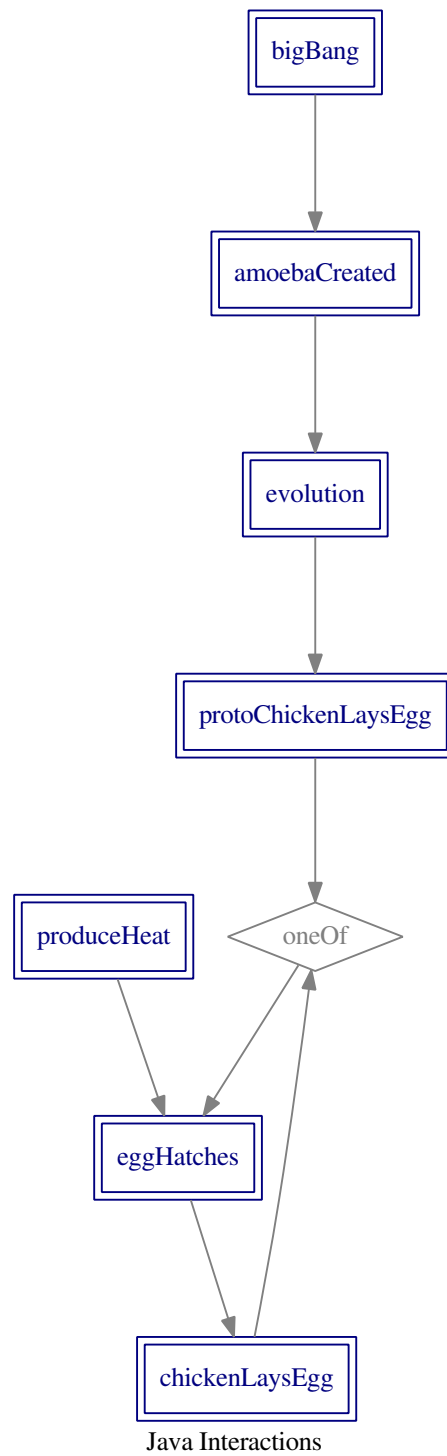


Figure 44: Chicken and egg problem in data generation

The `oneof` function is a QuickCheck primitive that randomly chooses one of the elements of the list provided. `args_for_op` function is just a wrapper function for `args_for` (in the same way `args_for` is a generator for nodes like the diamond node), the call with the label "11" corresponds to the node `protoChickenLaysEgg`, the call with the label "17" corresponds to the node `chickenLaysEgg`, and the label "diamond13o15" corresponds to the `oneOf` diamond. The `Size` parameter is used by QuickCheck to determine the size of the elements generated, a higher `Size` value represents a bigger value. We can see that the created generator decreases the value of `Size` every time we choose the method `chickenLaysEgg`, and it will not include the option `chickenLaysEgg` when `Size` is 0, thus guaranteeing termination.

Appendix D shows how the function `args_for/4` fits with the rest of the `eqc_fsm` model.

5.6.2 Generation of tests

The QuickCheck `eqc_fsm` models generated as described in Section 5.6.1 are analogous to the diagrams that we can visualise. In Figure 45 we can see the visual representation of part of the internal structure used to generate a QuickCheck `eqc_fsm` model for the Frequency server web service, and overlaid in black and gray we see the traversal QuickCheck did to generate the test in Figure 46.

Tests can be generated through the following steps:

1. The graph is traversed randomly through the control path, from the entry star through the brown arrows, with optional looping behaviour. Each node in this path (hereafter *step*) represents a call (HTTP request) to the API.
2. For each *step*, we generate the parameters required by following data flow arrows in reverse (possibly reusing values from previous steps), as shown (in Figure 45) by the green arrows.
3. Optionally, for each *step*, we generate postconditions by traversing the data flow in the direction of the arrows within the subgraph.

Since the model represents the behaviour of the target web service, it should not raise any (intended) exceptions. The results returned by the web service can be

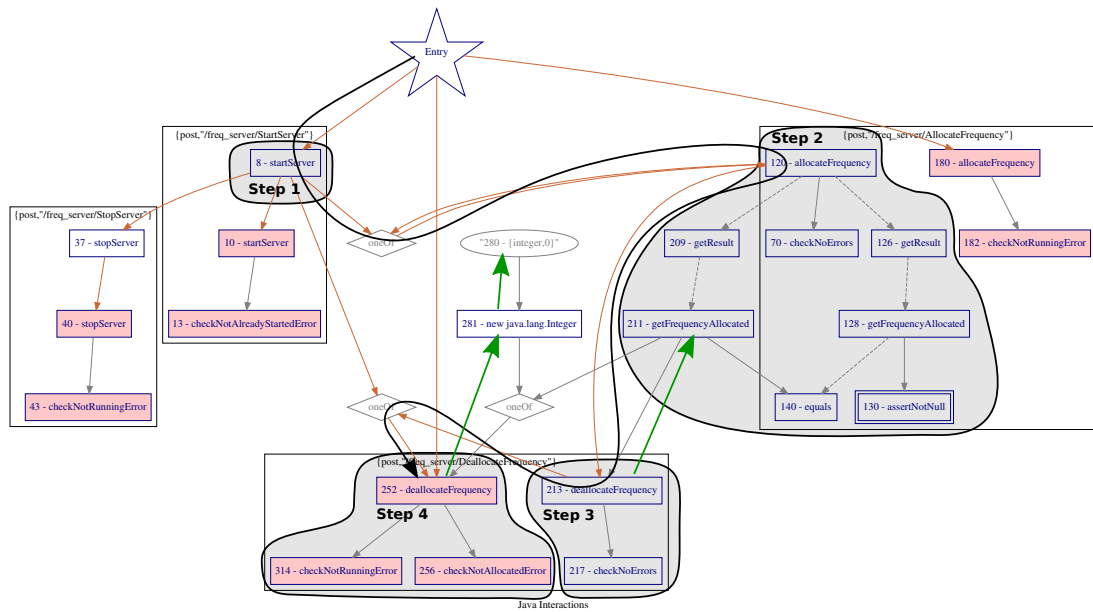


Figure 45: Diagram representation of test generated by James

classified as positive or negative depending on whether they represent an error or a normal result.

Generated QuickCheck `eqc_fsm` models, when run, print new JUnit test cases that can, after manual review, be added to the original suite. Manual review is necessary because some of the tests generated may have wrong postconditions, execute invalid commands, or test irrelevant aspects of the system; a test generated may fail because the test is wrong, not necessarily because the system is wrong; ultimately, it is up to the user (or up to some other authoritative oracle) to decide which behaviour is right (this is called the oracle problem Harman et al. 2013).

Once the generated tests have been suitably reviewed, it is possible to rerun the extraction process on the extended test suite and, thus, potentially generate a refined model of the system. This iterative refinement approach has been used in the past for model inference (Walkinshaw, Derrick and Guo 2009), and is similar to the CEGAR approach used in model checking (Clarke et al. 2000).

Unfortunately, tests generated may not necessarily be correct, not even for scenarios that have already been classified (see the discussion on soundness in the introduction of Chapter 6). The example test generated provided in Figure 46 is actually incorrect, some of the postcondition like:

```
// Postcondition: 2
```

```
FreqServerResponse var1 = this.startServer();
FreqServerResponse var2 = this.allocateFrequency();
// Postcondition: 1
this.checkNoErrors(var2);
// Postcondition: 2
Result var4 = var2.getResult();
Integer var5 = var4.getFrequencyAllocated();
Result var6 = var2.getResult();
Integer var7 = var6.getFrequencyAllocated();
boolean var8 = var5.equals(var7);
// Postcondition: 3
Result var9 = var2.getResult();
Integer var10 = var9.getFrequencyAllocated();
junit.framework.Assert.assertNotNull(var10);
// End of postconditions
Integer var12 = var6.getFrequencyAllocated();
FreqServerResponse var13 = this.deallocateFrequency(var12);
// Postcondition: 1
this.checkNoErrors(var13);
// End of postconditions
int var15 = 0;
Integer var16 = new Integer(var15);
FreqServerResponse var17 = this.deallocateFrequency(var16);
// Postcondition: 1
this.checkNotAllocatedError(var17);
// Postcondition: 2
this.checkNotRunningError(var17);
// End of postconditions
```

Figure 46: Example of test generated by James (without package qualifiers)

```
this.checkNotRunningError(var17);
```

will fail, and this issue needs to be solved manually.

5.7 Pilot study

In order to validate our approach, we have carried out a pilot study where we applied James to the administration web service of the system VoDKATV, developed by Interoud Innovation (Francisco 2014b). VoDKATV is an IPTV/OTT middleware that provides end-users with multimedia services through different devices such as a TV (through a set-top-box), a PC, a tablet, etc. The system is composed of several components that are integrated through the use of web services.

In particular, VoDKATV provides a web service used by administration applications to configure the VoDKATV platform; it allows applications to create users, set prices, configure channel lists, etc. This web service is invoked via HTTP or HTTPS (depending on the specific deployment), and returns data in XML format. Previous to the pilot study, there existed some manually written JUnit test cases targeted at this web service, and some of those were used as input for the James tool during the pilot.

By using the approach explained in this chapter, we were able to generate a diagram representing the execution of a JUnit test suite. New test cases were generated automatically from that execution, and the generated tests allowed the developers of the platform to find a previously unknown bug in the implementation of the web service. In the rest of this section, we present the results of the pilot study; the original report can be found in Arts et al. (2015).

Previous to the pilot study, four research questions were agreed by the whole team, and six measurements were established to try to answer the four questions.

The four questions were:

- A Is it technically feasible to augment Java test suites by inferring new tests from existing test suites?

- B Is the process of inferring new tests from Java test suites feasible from a business point of view: can the process be accomplished at a cost appropriate to the improvement in tests?

- C Do the inferred tests effectively augment the existing tests? This can be shown by discovering new faults in the system under tests?
- D Is the method assessed as accurate, quality enhancing and useful by the developers involved in the pilot study?

The six measurements were:

1. Number of tests in a unit test suite needed for the automatic extraction of useful test models [answers question A].
2. Number of additional (i.e. previously non-existent) tests cases needed for the automatic extraction of useful models (i.e. manually added by developers so that the extraction process can generate a useful test model) [answers questions A and B].
3. Number of new test cases added to test suite by means of the extracted models (automatic test suite enhancement) [answers question A].
4. Number of bugs revealed by means of the extracted models [answers question C].
5. Time and computational resources needed to infer and generate the models (scalability) [answers question B].
6. Developers' rating (0-10) of accuracy, quality and usefulness of the extracted models, compared to previously existing unit test suites [answers question D].

5.7.1 Results of the pilot study

During the pilot study, the James tool and, by extension, the approach described in this chapter, were tested for technical feasibility, viability from the business point of view, and for accuracy and effectiveness of the generated tests.

Using the 28 existing tests of VoDKATV, we used James to generate a model which is too big to include in this thesis (the model generated on one of the attempts consisted of 163 nodes, 311 arrows, and 12 subgraphs), and we generated a series of tests that were used to answer the questions of the pilot study

5.7.1.1 Number of tests required as input

One threat to validity of our approach was the potential number of existing tests required as input. A large number could translate into the need of additional testing effort (which is precisely one of the problems that the approach tries to address). In turn, the need for additional effort could translate into reduced viability from the business point of view. During the pilot, James was able to generate new tests even from a single input test case, even though in this case the tests generated were not very diverse.

For example, from a test that created and deleted a room, James was able to generate a negative test that deleted a non-existent room.

5.7.1.2 Additional effort required to obtain useful models

Similarly, even after confirming that the number of tests required for James to work is small, it still was a threat to validity the possibility that the effort required for the results to be useful would be disproportionate.

During the pilot, the initial set of 28 tests available was enough to produce a model considered “useful” for the 20 target operations tested. Thus, it was not necessary to add any extra tests. Nevertheless, it was necessary to adapt the existing JUnit test suite in order for James to produce the expected results (mainly due to the limitations described in Section 5.8 on page 154).

In particular:

- It was necessary to encapsulate some functions, by making the tests more high level.
- Results improved after rewriting methods to avoid side effects, by rewriting some parts to use a pure functional style.
- Developers unfolded one aspect of the set-up into the tests so that generated tests were more accurate.

These adaptations responded to problems either in the model or in the test generation. For example, because James issued errors when generating a model (for example, from tests that contained unsupported features), because James generated tests with methods that we did not want the generated tests to have (for

example, methods that generate debug information), or because we wanted certain procedures to be treated equally (merged) or treated differently (not merged) in the resulting model.

5.7.1.3 Number of new tests generated

In order for the approach to be useful, it must generate a significant number of new tests that explore new aspects that were not explored by the existing ones. During the pilot, James was able to generate thousands of JUnit test cases. Some of these tests were generated several times, but future experiments showed that replication could be addressed easily through the inclusion of the QuickCheck macro `?ONCEONLY` in the `eqc_fsm` model, which keeps track of generated tests and tells QuickCheck not to issue tests that have already been generated before.

Some tests were generated by James that exercised behaviours not considered in the original JUnit test suite, for example:

- Deleting existing and non-existing rooms in the same call to `deleteRooms`.
- Deleting duplicated rooms in the same call.
- Trying to update rooms that do not exist.
- Trying to create a device in a room that does not exist.
- Trying to create two devices with the same MAC address.

5.7.1.4 Number of bugs revealed

A good measure of the usefulness of the approach is the number of errors that it helps discover.

During the pilot, James helped developers to find one wrong behaviour. When the operation that deletes a device was invoked with an empty device identifier, it produced a `NullPointerException`, instead of returning a “required field” error as expected.

5.7.1.5 Time and computational resources required

The original suite took between 2.8 and 3.5 seconds to execute, whereas the instrumented test suite took between 70 and 100 seconds. The generation of the model took James an additional 20 to 25 seconds to complete.

5.7.1.6 Developer evaluation

The developer of the tests was asked to comment and rate James tool on a scale of 0 to 10 for accuracy, quality, and usefulness. In summary, the assessment was:

- Accuracy: 4
“[...] James generates thousands of new JUnit test cases, some of them test aspects that are not taken into account in the original JUnit test suite. However, there are many other test cases that are wrong because they try to test something in a wrong way (they make no sense and they fail even though the implementation of the SUT behaves as expected for the test scenario), [...]”
- Quality: 7
“The new test cases generated by James follow the same style and guidelines used in the original Java code [...] hence we consider that the quality of the new test cases in terms of source code quality is similar to the original [...] the variable names used in the new test cases [...] makes the new test cases harder to read.”
- Usefulness: 8
“Using James [...] helped to identify some situations that had not been tested before [...] the structure of the original JUnit test suite had to be modified slightly [...]”

5.7.2 Second run

In successive reviews of this work and thesis, we have been asked to provide more information that we did not collect during the first pilot study. For this reason, we recently reran (as faithfully as possible) the same experiments carried out during the first pilot study, but this time recorded more information and used the

All postconditions that pass	0	1	2	3	TOTAL TESTS
0	3	0	0	0	3
1	151	25	0	0	176
2	720	591	29	0	1340
3	157	584	161	31	933
4	0	0	18	16	34
5	0	0	251	244	495
6	0	0	118	107	225
TOTAL TESTS	1031	1200	577	398	3206

Table 5: Distribution of postconditions in tests generated

most recent version of James available instead. In this subsection, we present a summary of the extra data collected.

For the second run, we executed James on the same 28 tests (adapted to James from scratch), and generated a model.

This time, we manually connected the model to VoDKATV by using the JEB interface (Lamela Seijas 2014c). The modified model (at test generation time) checked the tests generated against VoDKATV before outputting them, and commented out those parts of the tests that produced exceptions. This mechanism ensured that tests output by the model passed.

Using this mechanism, the model generated by James generated 10,000 random tests of which 3,206 were unique and, thus, were output by the model.

By analysing the 3,206 output tests (including their commented out lines) we obtained the following data.

In Table 5 we show the distribution of postconditions generated in tests and the amount of passing postconditions in each case. Cells with grey background indicate tests that pass directly, without commenting out any instruction, 88 in total. In our experiments, except for the postconditions (assertions or calls to methods that execute one or more assertions), none of the normal instructions in the test generated raised any exceptions; thus, all generated failing tests were due to failing postconditions.

In Table 6 we show how many methods in each test issue HTTP requests in the tests generated, and the average number of methods, postconditions, and

Number of HTTP methods per test	Number of tests	Average number of methods per test	Average number of postconditions per test	Average number of instructions per test
0	3	1.0000	0.0000	1.0000
1	126	6.7143	1.6349	11.3571
2	568	9.8380	1.9701	14.8908
3	1417	12.8574	3.2519	19.8483
4	717	17.0181	3.3710	26.8745
5	263	21.5247	3.4144	34.2167
6	80	25.7375	3.3375	41.4625
7	31	30.2903	3.1613	48.9032
9	1	33.0000	3.0000	48.0000
TOTAL	3206	14.2077	2.9994	22.1978

Table 6: Distribution of methods that produce HTTP requests

Classification	Matching tests
1 interesting part	10
2 interesting parts	2
Both interesting and not	8
Only non-interesting parts	9
TOTAL	30

Table 7: Manual evaluation of interest for first 30 tests

Classification	Matching tests
Negative tests	13
Positive tests	5
Both positive and negative	3
Non-interesting tests	9
TOTAL	30

Table 8: Manual evaluation of positive or negative testing

instructions per test.

The first 30 tests generated during the second run were also classified by the developer of the original tests using the following criteria (see Table 7):

- Interesting parts of a test are those that have postconditions that check what they do.
- Uninteresting parts of a test are those that do things but do not check the result (or not correctly).
- A test with both means that it has both interesting and uninteresting parts.
- A test with neither means that the test does nothing to the SUT.

All this is done while ignoring those postconditions that produce exceptions, and, thus, if we consider that the SUT is correct, they all represent wrong postconditions (the proportion of these failing postconditions is already detailed in Table 5).

Additionally, the tests that had at least one interesting part were classified manually into the following categories (see Table 8):

- Positive tests: for example, a room is created and the postcondition checks that the creation was successful, or a room is deleted and the postcondition checks that the deletion was successful.
- Negative tests: for example, an attempt is made to delete a non-existent device or to create a room with incorrect data, and the postconditions checks that the result is an error.

5.8 Limitations

There are some limitations to our approach, some of them are derived from the technologies used (technical limitations) but could be circumvented if a large amount of development effort was applied; others are intrinsic to our approach (conceptual limitations) and would require considerable changes to it.

5.8.1 Technical limitations

In Java, some variables have primitive types (for example: `int`, `char`, `boolean`), all of these can be replaced by wrapper classes (for example: `Integer`, `Character`, `Boolean`), but using them is less efficient. One limitation derived from the shallow use of JVMTI is that primitives cannot be tracked. And the same problem exists for operators like `+` or `&&`, which are treated differently from normal methods by the JVMTI.

Our current implementation attempts to track primitives by linking observations of the same value. This works fine if all the primitives that are observed have different values, but this is quite unlikely to happen, specially because there are some primitives that are reused frequently with different intentions; primitives like `true`, `false`, and `0`. In addition, it is not possible for our implementation to trace the use of primitives through operators.

The solution suggested by the documentation of the JVMTI (JVM-TI 2006), is to use dynamic bytecode modification. It would be theoretically possible to use this mechanism to, for example, replace primitives with objects and operators with methods. But, because James was built as a prototype, we bypassed the problem by replacing primitives manually previous to the instrumentation process.

In addition, some artefacts used in Java code are translated into compiler-generated methods, and some methods are implemented natively. There are some kinds of information (like information about local variables) that JVMTI API does not always provide for some of these methods.

Even for normal methods, the amount of information that can be retrieved directly through the use of JVMTI API depends on whether the code was compiled with debug information.

In our aim to obtain a more versatile tool, we chose to use ways of extracting information that rely on JVMTI methods that, in our experiments, worked independently of whether Java code was compiled with debug information enabled.

5.8.2 Conceptual limitations

As we described in Section 5.2.2.2 (on page 121), in our approach, control flow is only tracked for methods that issue HTTP requests, since they are the ones that can modify the state of the target web service. This means that the model

will not consider the consequences of side-effects that are produced by the rest of methods. Those methods cannot modify the state of the web service, but they can modify and depend on the local state of the client that is running them.

If the tests used as input contain local side-effects, the models produced by James may be inaccurate. In Chapter 6, we study a way of redesigning the inference algorithm described in this chapter so that it acknowledges the possible side-effects caused by any method, while still representing, in a single model, the interactions between control and data flow.

Another problem, which is generic to dynamic approaches – already reported in previous research (Lo et al. 2011) – is the large number of traces produced by dynamic instrumentation, which causes the analysis of relatively small test suites to require a substantial amount of memory and, thus, slows down the process considerably. This problem is mitigated by a careful early filtering of the traces collected as mentioned in Section 5.3.3 (on page 123).

Finally, analysing the data flow can be a problem by itself when it is not explicit. While, in Section 5.8.1, we talked about the difficulty to track primitives as opposed to tracking objects, there is a semantic aspect to it: even if all the values were objects, we may still have two equal objects that refer to the same thing in the real world, for example: two objects for the number 10 may be created separately and still both be used to represent the frequency channel number 10. On the other hand, the two instances of number 10 may represent different things, like the floor number 10 of a building. In those cases, our algorithm has no easy way of knowing whether those equal values refer to the same thing, or maybe, for example, one refers to the floor on which the program is running.

Fortunately, it is usually considered a good practice to declare “objects” and “values” that refer to the same thing together as constants, so that information is usually present in programs. Nevertheless, it must probably be the user who provides (one way or another) the flow information, in Chapter 6 we directly assume the user provides it as part of the input.

5.8.3 Control tracking workaround

The task of identifying methods that issue HTTP requests could be carried out by ensuring that all traffic goes through a proxy and letting the proxy communicate

this information in a synchronised way to the JVMTI agent. Nevertheless, this approach would add extra complexity to the implementation and would potentially require a context switch between the JVMTI agent and the proxy for each method call, and this would introduce a delay that would slow down the whole process considerably.

Instead, we use the JVMTI agent directly to detect Java methods that we know produce HTTP requests, this is a small overhead since we already needed to track all the methods' entry and exit events to retrieve the information about control and data flow anyway. In our experiments, we have set the agent to detect the methods `openConnection` and `setRequestMethod` from the class `HttpURLConnection`. Other target systems may use different methods to produce HTTP requests, but James can be easily adjusted to detect those instead.

5.9 Lessons learned

In this section, we present some concrete insights from the pilot study about weaknesses, potential future improvements, and areas for future research.

5.9.1 Implicit relationships

During the pilot study we observed that there was a relationship between groups of instances that had an intrinsic relationship among them: they had to be used together in certain function calls in order for the calls to be correct.

For example, in one of the tests used as input during the pilot study we could find the following sequence of calls:

```
String inputXml1 = generateRoomXML(roomId1, description1,
                                   tagTreeNodeId1);
String resultCreate1 = createRoom(inputXml1);
checkThatRoomWasCreated(inputXml1, roomId1, description1,
                        tagTreeNodeId1, resultCreate1);
```

We can see that there is a relationship between the values stored in the variables `roomId1`, `description1`, and `tagTreeNodeId1`: they all correspond to the same physical room. Because of this relationship, it would most likely not make sense to change any of the three parameters in either of the function calls.

Unfortunately, James cannot infer this relationship, and randomly choosing the parameters for `checkThatRoomWasCreated` so that all of them correspond to the same room, even if we only have only created two rooms, would have a probability of one out of $2^4 = 16$. And this number only grows exponentially when we increase the number of calls in the tests generated.

This particular choice of parameters is due to the existence of what we could call an “implicit object”, the room, with a set of attributes that have a special meaning when used together. This kind of relationship is relatively simple and we could potentially come up with ways of detecting it, but we can imagine there could exist very complicated relationships between values (or objects) that are far from obvious to guess, for example: one method could have a different effect depending on whether it is passed a room with a prime number of devices associated.

In order to produce more meaningful tests, it would be necessary to detect these implicit relationships and consider them when generating new tests. Of course, it is still interesting to generate wrong implicit relationships for negative testing, but the proportion of positive cases should be comparable to the proportion of negative cases, and it would be desirable to be able to predict the classification of generated tests accurately.

Alternatively, we can delegate the task of managing implicit relations to testers and developers, by assuming that there exist abstractions that hide them, or by assuming they are all explicit (for example, by assuming there are unique explicit identifiers for everything).

5.9.2 Classification of traces

As mentioned in Section 5.4, James classifies methods into two categories:

- Positive – those that are expected to represent normal behaviour of the system.
- Negative – those that exercise unexpected behaviour of the system, for example: in the face of wrong or malformed input. Our implementation used the strings like “fail” and “error” as heuristic.

We assume that tests provided pass, thus, we do not consider execution time errors in the tests. Tests that fail could straightforwardly be considered as negative tests,

but this is not very useful in the case of web services, since we try to find errors in the target web service, not on the side of the client.

In the pilot and, for example, in the step 4 of Figure 45 on page 145, we could see that responses for different types of error were checked differently, with separate assertions. The current approach tries to merge all the failing nodes together and, because of this, tests generated may have several assertions that are contradictory.

As a consequence, we could already see that the example generated test shown in Figure 46 on page 146 checks the same result value for two contradictory assertions `checkNotAllocatedError` and `checkNotRunningError`. The same problem may potentially occur with positive assertions.

These results suggest that future work could classify tests into an arbitrary number of categories, instead of only two.

5.9.3 State inference for objects

The current version of James uses control flow to represent the state of the server. But this approach is not used for inferring the state of anything in the client. One possibility would be to also record the control flow of methods that do not issue HTTP requests, in order to model the state information of the client as well. But this would probably lead to intricate, hard to read diagrams, as we saw in Section 4.4 on page 103.

A better alternative could be to track state for objects or classes separately, instead of the system as a whole. Several different approaches for modelling the state of objects have been explored in the past (Henkel and Diwan 2003; Yuan and Xie 2005; Xie and Notkin 2004). Nevertheless, it would be necessary to coordinate the state of the objects in the model through the use of guards or messages.

Chapter 6 explores this idea by presenting a formalism inspired by the idea of representing the state of different objects separately, but synchronisation is done through the combination of the different models into a single one, similarly to how it is done in this chapter, but with an extra care to preserve soundness.

5.10 Chapter conclusions

In this chapter, we have presented a set of techniques to generate useful models that combine information from both data and control flow, and for using these models to generate new tests. These techniques have been implemented in the James tool and, even though further validation would be required in order to say whether the approach is effective in practice or whether other approaches would be preferable, we have seen an example of their application to an industrial system that both suggests:

- That the techniques described here may be useful to help find bugs.
- That there are both technical and theoretical limitations to them; in particular, we highlighted its lack of soundness, which limits its accuracy.

We have also illustrated the techniques with examples from specific executions of James and examples extracted from the pilot study.

But the experiments also show how by combining control and data flow we can obtain more expressive models, and open the door to a more modular perspective for modelling systems. In Chapter 6 we explore this perspective from a slightly more formal point of view, by paying more attention to the accuracy of the inference algorithm and abstracting away from the context of its application, that is: we will not discuss any more about web services, JUnit, JVMTI, or instrumentation; instead we will focus on the traces and the models.

Chapter 6

Recovering soundness

In Chapter 5, we have seen how combining control and data flow can increase the expressiveness of our models, by allowing smaller models to represent larger state spaces. But, the merging algorithm described in Chapter 5 does not guarantee that the model generated predicts the behaviour of the system correctly with respect to the examples given as input.

For example, we can provide the inference algorithm a test that fails, and the inference algorithm may produce a model that generates the exact same tests and still expects it to succeed. We say that an inference algorithm is sound if it is guaranteed to produce models that conform with all available information (if a set of traces is provided as input, the resulting model must be able to correctly classify at least all the traces in that set).

It is worth discussing this claim; it is not obvious that James is not sound:

- On the one hand, considering that James only records control flow for the target web service, it would not be hard to find an example in which the order of the methods called from the client affects the classification of the test (whether it is positive or negative). Since James does not record the order in which the methods are called, the model produced by such examples would be non-deterministic and thus unable to classify the test provided as input.
- On the other hand, let us assume that the methods called from the client are referentially transparent except for the issue of HTTP requests to the server. Then, we might be able to argue that James is sound given a big enough

value for both K parameters, since such a value will prevent any merging. But, in that case, for the inference algorithm to be useful, we would need to show that there exists a value of K for which James generalises enough (while at the same time preserving soundness). For k -tails, there exist proofs that show that the inferred machine is minimal under some assumptions. We have not proven that for the merging algorithm in James; and we have not proven that such a proof is impossible either.

- All these details make it very difficult to reason about the relationship between the ability to obtain a concise state machine and the ability to correctly classify inputs of the merging algorithm of James (regulated by the parameter K). For that reason, in this chapter, we describe a model and algorithm that does not depend on a parameter K , but rather, it tries to merge as many states and transitions as possible without losing soundness.

Soundness is important because its absence makes it harder to iteratively refine models:

- Incorrect tests that are generated are not guaranteed to be new, although this can potentially be prevented just by keeping track of the tests generated.
- The models inferred are, in principle, unable to represent some aspects of the system: that is, no amount of learning would keep the models from generating certain kinds of incorrect tests.

For these reasons, before starting to implement the inference algorithm presented in this chapter, we wrote a property that states that the model inferred from a set of traces correctly classifies all the strings from which it was inferred, given the proviso that inputs are valid according to the definition provided in Section 6.2.1 (on page 164), and we tested this property in QuickCheck for Haskell (see Section 2.4 on page 20), by evaluating it for sets of randomly generated traces. Throughout this chapter, when we talk about QuickCheck we mean QuickCheck for Haskell.

Of course, it can be argued that testing does not guarantee that the property is true for all inputs (there can still be bugs in the implementation), but it makes it much more likely that we will find out if the property is broken by a fundamental flaw in the design of the algorithm. Formal verification of the property is left to future work.

Glass and bottle. We will illustrate the definitions in this chapter by using traces of a hypothetical system. This is not a real system, but we did define a model that is included as part of the source code of the implementation in Haskell (Lamela Seijas and Thompson 2016b). It is not necessary to use the model to run the examples shown in this chapter, the traces provided are enough to generate the diagram; the model was just used for preparing the examples.

The hypothetical system just consists of five possible actions or operations:

1. `createBottle` - Produces a closed bottle.
2. `createGlass` - Produces an empty glass.
3. `openBottle` - Takes a closed bottle and produces an open bottle.
4. `closeBottle` - Takes an open bottle and produces a closed bottle.
5. `fillGlassUsingBottle` - Takes a glass and an open bottle.

For simplicity, we do not consider what would happen if we fill a glass twice, or if the bottle is empty; those aspects are irrelevant regarding the examples in this chapter. But we assume that an arbitrary number of glasses and bottles can be created, and that `fillGlassUsingBottle` cannot be applied to a closed bottle.

As a case study, we will use two versions of the Frequency Server, similar to the implementation described in Section 2.1.5 (on page 15).

6.1 Contributions

In this chapter, we present a formalised model similar to the one presented in Chapter 5, and we informally describe a sound inference algorithm to learn the model from examples of traces. We do this by extending the FSM model and the Blue-Fringe inference algorithm (see Section 2.3.1 on page 18) respectively.

In particular, we formally describe:

- *parametrised automaton*, as an extension of the regular automaton that allows symbols to include parameters.
- The format that input traces of *parametrised automata* must have in order to be considered valid.

- The algorithm that a *parametrised automaton* uses to run valid input traces and classify them.

Later, we informally present an inference algorithm for parametrised automata in terms of its differences with the Blue-Fringe inference algorithm for regular automata (see Section 2.3.1 on page 18). We illustrate the whole inference process by showing the inference from a set of four traces and diagrams of the model at each step.

The work presented in this chapter has not been published at the time of writing this thesis. I contributed with the design of the technique, the implementation, and most of the writing. The supervisor of this thesis, Simon Thompson contributed with ideas, suggestions, guidance, advice, revisions, and editing.

6.1.1 Software contributions

I have implemented prototypes in Haskell of the inference and classification algorithms described in this chapter, and their source code is available at (Lamela Seijas and Thompson 2016b).

6.2 Parametrised automaton formal definition

In this section, we describe a new formalism that we call *parametrised automaton*, which extends regular automata by modifying the concept of symbol to include parameters that reference previous symbols in the trace and can be used to describe the data flow when used for representing software systems. Accordingly, this modification forces us to define parametrised alternatives for the concepts of trace, automaton, and run.

6.2.1 Input format

A parametrised automaton takes as input a parametrised *trace* or *word*. In this section, we formally define parametrised *trace*.

6.2.1.1 Alphabet

An alphabet is a finite set of labels $\Sigma = \{l_1, l_2, \dots, l_n\}$.

6.2.1.2 Symbol

For any $d \in \mathbb{N}$, we call symbols of order d (written as sym_d) the set of all possible symbols that can be used in the position d of a *trace* or *word*. Formally, we define sym_d as the set of 2-tuples (l, p) where:

- l is a label of the alphabet: $l \in \Sigma$
- p is a parameter list defined by a finite sequence of natural numbers $p = \langle p_1, p_2, \dots, p_n \rangle$ such that:
 - $\forall i. p_i \in \mathbb{N}^0$, parameters represent the index of a previous symbol in the trace, starting with zero (which represents the first symbol of the trace).
 - $\forall i. p_i \geq 0 \wedge p_i < d - 1$, only symbols earlier in the trace can be represented.
 - $\forall i \forall j. i \neq j \Rightarrow p_i \neq p_j$, every parameter must reference a different previous symbol in the trace.

The reason why we do not allow several parameters in the same symbol to be equal is that we want to track the last place in which a value was used at any given point in the trace. We do so by storing in which parameter of which symbol it was last used, but if it was used in several parameters simultaneously we would have to track several last usages and it would considerably complicate the whole algorithm and representation. The convenience of having this constraint in the input will become clearer after looking at the dependency rewriting (Section 6.3.1 on page 177) and the parametrised automaton representation (Section 6.2.2) later in this chapter.

6.2.1.3 Parametrised trace

In order to be valid, a parametrised input *trace* or *word* is a finite sequence of symbols $\langle s_1, s_2, \dots, s_n \rangle$ such that:

$$\forall i \in [1, n]. s_i \in sym_i$$

For clarity, we represent the empty sequence $\langle \rangle$ as ε .

6.2.1.4 Example

Given the following alphabet:

$$\Sigma = \langle \text{createBottle}, \text{createGlass}, \text{openBottle}, \text{fillGlassUsingBottle} \rangle$$

A possible valid trace would be:

$$t_1 = \langle (\text{createBottle}, \varepsilon), (\text{createGlass}, \varepsilon), (\text{openBottle}, \langle 0 \rangle), \\ (\text{fillGlassUsingBottle}, \langle 1, 0 \rangle) \rangle$$

This would be an invalid trace:

$$t_2 = \langle (\text{createBottle}, \varepsilon), (\text{createGlass}, \varepsilon), (\text{openBottle}, \langle 2 \rangle) \rangle$$

Since the third symbol ($\text{openBottle}, \langle 2 \rangle$) does not belong to sym_3 (it belongs to sym_n where $n \geq 4$). This restriction is defined because the parameter “2” would refer to the result of the symbol ($\text{openBottle}, \langle 2 \rangle$) itself, whereas it should point to a symbol that is earlier in the trace.

Another invalid trace would be:

$$t_3 = \langle (\text{createBottle}, \varepsilon), (\text{createGlass}, \varepsilon), (\text{fillGlassUsingBottle}, \langle 0, 0 \rangle) \rangle$$

Since the third symbol ($\text{fillGlassUsingBottle}, \langle 0, 0 \rangle$) has two parameters that are equal in its parameter list, and it would require the model to incorporate the concept of “cloning” a value, which would foreseeably be more complicated. In the model, as it is defined currently, values are stored on a single place at all times.

6.2.2 Parametrised automata

In this section, we present a formal definition for parametrised automata.

6.2.2.1 Possible sources

We have seen that, on *traces*, *symbols* depend on previous symbols. This is represented by the numbers in their parameters. Analogously, we allow transitions in

parametrised automata to depend on other transitions.

We first start by defining what a transition can depend on, that is: what a *source* for a dependency can be. Thus, given a set of transitions T , we define the set of possible sources of T , represented as S_T , as follows:

$$S_T = \{(o, p) \mid o \in T \wedge p \subseteq \mathbb{N}^0 \wedge |p| \leq 1\}$$

The elements of S_T represent possible sources for parameters obtained from other transitions.

- o is the source transition.
- p is a list that may either be empty or have exactly one element:
 - If p is the empty list ($p = \emptyset$), the source represents the “result” of the source transition.
 - If p has exactly one element ($|p| = 1$), the only element of p represents the parameter number (the position in the sequence of parameters) to use from the source transition (where 0 represents the first parameter).
 - The model is invalid if p has more than one element ($|p| > 1$).

For example, $(2, \emptyset)$ represents a value produced by the transition with identifier 2. $(3, \{1\})$ represents a value used as a second parameter (parameter number 2) by transition with identifier 3.

6.2.2.2 Possible source combinations

A transition can depend on several sources. In fact, each parameter of a transition can depend on more than one source (it must depend on at least one). Thus, we define a *source combination* as a sequence of sets of sources, where the first set of the sequence defines the possible sources for the first parameter, the second set of the sequence defines the possible sources for the second parameter, and so on...

Formally, given a set of transitions T , the set of all possible source combinations of T , represented as SC_T , is defined as:

$$SC_T = \{\langle s_1, s_2, \dots, s_n \rangle \mid \forall i. s_i \subseteq S_T \wedge s_i \neq \emptyset\}$$

where S_T is the set of possible sources of T , as defined in Section 6.2.2.1.

6.2.2.3 Parametrised automata

Now we can define a parametrised automaton as a 6-tuple $(Q, T, \Sigma, \delta, q_0, N)$ where:

- Q is a finite set of states.
- T is a finite set of transition identifiers.
- Σ is an alphabet as defined in Section 6.2.1.1 (on page 164).
- δ is the transition function index:

$$\delta = T \rightarrow Q \times \Sigma \times SC_T \times Q$$

where the first element of the image of the δ function represents the source state of the transition, and the last element of the image of δ represents the destination state of the transition. SC_T is the set of possible source combinations of T , as defined in Section 6.2.2.2.

- q_0 is the initial state, $q_0 \in Q$
- N is the set of failing states, $N \subseteq Q$

6.2.2.4 Example

The parametrised automaton represented in Figure 47 on page 172 (legend in Table 9 on page 171) can be formalised as follows:

- $Q = \{S, E\}$ – These represent the two states of the automaton: S represents the initial state (represented as a yellow start in the diagram), and E represents the error state (represented as a red pentagon in the diagram).
- $T = \{1, 2, 3, 4, 5, 6\}$ – These are just identifiers for the different transition. Transition identifiers are not shown in the diagram.
- $\Sigma = \{\text{createBottle, createGlass, openBottle, closeBottle, fillGlassUsingBottle}\}$ – These are all the different labels that transitions can have. They are represented by brown round nodes in the diagram.

- δ is the following function:

T	$Q \times \Sigma \times SC_T \times Q$
1	$(S, \text{createBottle}, \varepsilon, S)$
2	$(S, \text{createGlass}, \varepsilon, S)$
3	$(S, \text{closeBottle}, \langle \{(5, \{0\})\} \rangle, S)$
4	$(S, \text{fillGlassUsingBottle}, \langle \{(2, \emptyset)\}, \{(1, \emptyset), (3, \{0\})\} \rangle, E)$
5	$(S, \text{openBottle}, \langle \{(1, \emptyset), (3, \{0\})\} \rangle, S)$
6	$(S, \text{fillGlassUsingBottle}, \langle \{(2, \emptyset)\}, \{(5, \{0\})\} \rangle, S)$

δ maps the transition identifiers to the transition information. Transitions are represented in the diagram as brown arrows that go between states and have a brown round node with the label in the middle; in the formal notation, the first and last elements of the image of δ function represent the source state and the target state of each transition respectively. The round labels in the middle of transitions also have incoming and outgoing black arrows that represent parameter dependencies between transitions; parameter dependencies are the ones represented between angle brackets in the formal notation. For example, the transition number 6 is the one at the top of the diagram, labelled `fillGlassUsingBottle`, that starts at the initial state, S , (star shaped) and goes to the same initial state, S . We can see in the formal definition that the transition takes two parameters:

- The first parameter (or parameter 0), $\{(2, \emptyset)\}$, comes from transition 2 (`createGlass`) and takes the result (\emptyset), this is represented by the black arrow that says “RetVal as param: 0”.
 - The second parameter (or parameter 1), $\{(5, \{0\})\}$, comes from transition 5 (`openBottle`) and takes the value used as its first parameter (or parameter 0), this is represented by the black arrow that says “Param-Num 0 as param: 1”.
- $q_0 = S$ – This indicates which of the states in Q represents the initial state, in our example it is the state S .

- $N = \{E\}$ – This indicates which subset of states in Q represent erroneous states, in our example it is just the state E .

Another example, we can see that:

$$\delta(4) = (S, \text{fillGlassUsingBottle}, \langle \{(2, \emptyset)\}, \{(1, \emptyset), (3, \{0\})\} \rangle, E)$$

this means that there is a transition with identifier 4 from state S (the initial state) to state E (the failing state) that can be run with a symbol labelled “`fillGlassUsingBottle`”, that has an unused result of `createGlass` as first parameter (from transition with identifier 2), and has as second parameter either an unused result from `createBottle` (from transition with identifier 1) or a value whose last use was as first parameter (parameter 0) of `closeBottle` (from transition with identifier 3).

6.2.3 Run

Given a valid trace t and a parametrised automaton a , there are four possible results of a running t : *accept*, *reject*, *undefined*, *indeterminate*. A parametrised automaton is considered *deterministic* iff there is no valid trace that, when run, will produce the result *indeterminate*.

6.2.3.1 Execution state

The result of the run is determined by iteratively parsing the input trace t and updating the execution state, starting with the initial execution state.

The execution state of a parametrised automaton $a = (Q, T, \Sigma, \delta, q_0, N)$ can be defined by a 4-tuple (q, e, v, r) where:

- q is the current state where $q \in Q$
- e is the current position in the trace starting with 0 (that is: the number of symbols that have been processed already) where $e \in \mathbb{N}^0$
- v is the collection of values that can be used as parameters where $v \subseteq S_T \times \mathbb{N}^0$ (it is represented by a set that contains the sources created and not used paired with the position in the trace where they were created or last used)



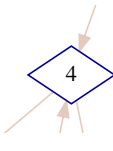

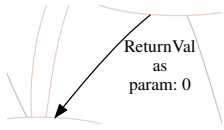
	<p style="text-align: center;">Initial state</p> <p>The initial state q_0 is represented with a yellow star labelled START. It is the state in which every run begins.</p>
	<p style="text-align: center;">Failing/erroneous state</p> <p>Failing or erroneous states N are represented with a red pentagon with double outline labelled ERROR!. There could be several failing states but we usually merge them so that only one is left.</p>
	<p style="text-align: center;">Normal state</p> <p>Every state in Q that is not the initial state or a failing/erroneous state is represented with a blue diamond labelled with a number.</p>
	<p style="text-align: center;">Transition</p> <p>Transitions δ are represented as brown arrows that go from one state to another. The arrows have an elliptical node in the middle which is labelled with the label of the symbol they accept and “(. . .)”, but this node is part of the transition (not to be confused with a state node).</p>
	<p style="text-align: center;">Parameter dependency</p> <p>Sources for parameters SC_T are represented through black arrows labelled with the type of source (ReturnVal or ParamNum X) and the parameter position in which the source is used (as param: X)</p>

Table 9: Legend for parametrised automaton diagrams

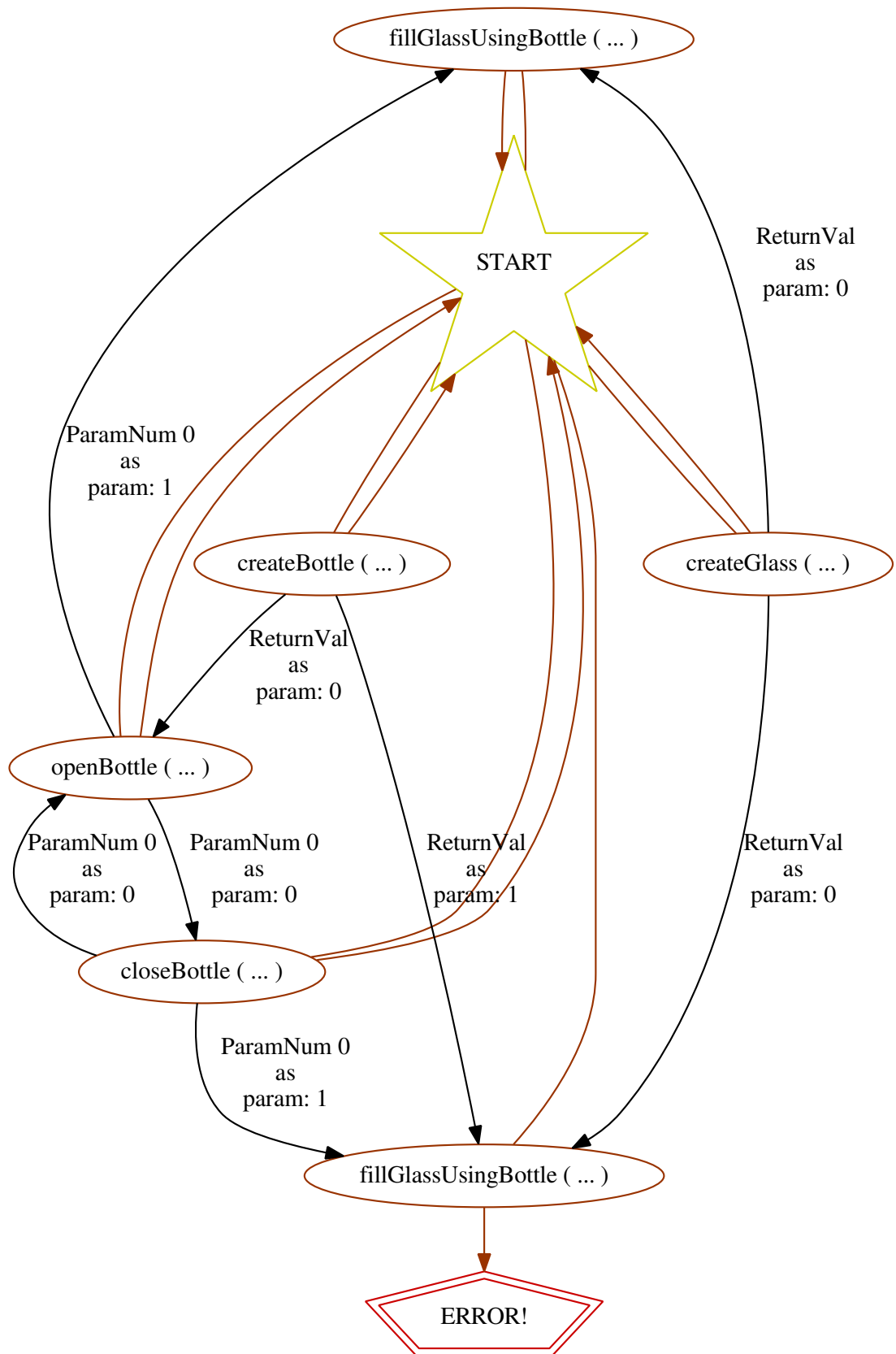


Figure 47: Representation of example parametrised automaton

- r is the unparsed part of the input trace t . r is represented as a suffix of the input trace sequence t , but it may not necessarily be a valid trace itself.

Initial execution state At the beginning of the process of the automaton $a = (Q, T, \Sigma, \delta, q_0, N)$ running the trace t , the execution state is initialised to:

$$(q_0, 0, \emptyset, t)$$

That is, the initial state of the automaton, the position 0 in the trace, an empty collection of values, and the whole trace remaining.

Rejection execution state If at any point during the run, the current state q becomes a failing state, that is: $q \in N$, then we say the result of the run is *reject*.

Acceptance execution state If the trace is not rejected and the unparsed part of the input trace r becomes empty, that is: $r = \varepsilon$, then we say the result of the run is *accept*.

6.2.3.2 Execution state transition algorithm

Given a trace or trace suffix r , which is a non-empty sequence, where $r = \langle \alpha, \beta, \dots \rangle$, in order to parse the first symbol α where $\alpha = (l, p)$ such that $\alpha \in sym_n$ for some n and where p is a list of parameter list where $p = \langle p_1, p_2, \dots, p_n \rangle$, we must find a triple (f, s, ls) where:

- f is a transition identifier $f \in T$
- ls is a possible source combination $ls \in SC_T$ where $ls = \langle ls_1, ls_2, \dots, ls_n \rangle$ such that:
 - $|ls| = |p|$ – the parameter list of the symbol to be processed and the sequence of possible sources must have equal length.
 - $\delta(f) = (q, l, ls, q')$ for some destination state q' – the list has to belong to a transition that has the same label as the symbol we are trying to parse (l), and has to start in the current state (q) but the target state (q') can be any state.

Note that ls is a sequence of sets, each set represents possible sources for a parameter, thus we still need to choose which element in each set to use (which source for each parameter), s represents that choice.

- s is a choice of sources that corresponds to the sequence ls , where $s = \langle (sp_1, o_1), (sp_2, o_2), \dots, (sp_n, o_n) \rangle$, such that:
 - $|s| = |ls|$ – the choices of sources and the sequence of sets of possible sources must have equal length (one element per parameter).
 - $\forall i. sp_i \in ls_i$ – the first element of each tuple in s is one of the sources in ls .
 - $\forall i. (sp_i, o_i) \in v$ – for each parameter, we pick a value o that must be stored in source sp , v is part of the execution state and establishes which values are in which sources at any moment of the execution.
 - $\forall i. p_i = o_i$ – the value o represents at which position in the trace the value was created, this identifies the value uniquely (there may be several values in the same source, but they were necessarily created at different points in the trace). This is the way the trace specifies which value is being used and must match.
 - $\forall i \forall j. i \neq j \Rightarrow (o_i \neq o_j)$ – we cannot use the same value more than once in the same symbol (because that would duplicate it).

Undefined transition If there is not such triple (f, s, ls) then we say the result of the run is *undefined*.

Non-determinism If there are several possible triples (f, s, ls) (independently of whether they lead to the same destination state q' or not) then we say the result of the run is *indeterminate*.

Normal transition If there is exactly one triple (f, s, ls) we update the execution state as follows:

- the current state is set to q'
- the current position in the trace is increased by one: $e' = e + 1$

- the values used (specified by the sequence s) are moved to the transition used, and a new result value is generated. We update the storage v to reflect this, where \tilde{s} is the set of elements of the sequence s , $(v \setminus \tilde{s})$ is the storage without the used values, $s_i = (sp_i, o_i)$ is the i^{th} element of the sequence s , $\{((f, \{i\}), o_i) \mid i \in [0, |\tilde{s}|]\}$ are the parameters used (which we store in the transition f), and $((f, \emptyset), e)$ is the new result value:

$$v' = (v \setminus \tilde{s}) \cup \{((f, \{i\}), o_i) \mid i \in [0, |\tilde{s}|]\} \cup \{((f, \emptyset), e)\}$$

- the first element is removed from the unparsed part $r' = \langle \beta, \dots \rangle$

6.2.3.3 Example

Let us consider a run of the following trace by the automaton described in Section 6.2.2.4 (on page 168):

$$t_1 = \langle (\text{createBottle}, \varepsilon), (\text{createGlass}, \varepsilon), (\text{openBottle}, \langle 0 \rangle), (\text{fillGlassUsingBottle}, \langle 1, 0 \rangle) \rangle$$

We first start with the initial execution state:

$$x_0 = (S, 0, \emptyset, t_1)$$

The execution would go through the following iterations:

q	e	v	r
S	0	\emptyset	$\langle (\text{createBottle}, \varepsilon), \dots \rangle$
S	1	$\{((1, \emptyset), 0)\}$	$\langle (\text{createGlass}, \varepsilon), \dots \rangle$
S	2	$\{((1, \emptyset), 0), ((2, \emptyset), 1)\}$	$\langle (\text{openBottle}, \langle 0 \rangle), \dots \rangle$
S	3	$\{((5, \{0\}), 0), ((2, \emptyset), 1), ((5, \emptyset), 2)\}$	$\langle (\text{fillGlassUsingBottle}, \langle 1, 0 \rangle) \rangle$
S	4	$\{((6, \{1\}), 0), ((6, \{0\}), 1), ((5, \emptyset), 2), ((6, \emptyset), 3)\}$	ε

Because q is not a failing state and r is empty we can conclude that the result of the run is *accept*.

But if we remove the third symbol from the trace:

$$t_2 = \langle\langle \text{createBottle}, \varepsilon \rangle, \langle \text{createGlass}, \varepsilon \rangle, \langle \text{fillGlassUsingBottle}, \langle 1, 0 \rangle \rangle\rangle$$

The execution would go through the following iterations:

q	e	v	r
S	0	\emptyset	$\langle\langle \text{createBottle}, \varepsilon \rangle, \dots \rangle$
S	1	$\{((1, \emptyset), 0)\}$	$\langle\langle \text{createGlass}, \varepsilon \rangle, \dots \rangle$
S	2	$\{((1, \emptyset), 0), ((2, \emptyset), 1)\}$	$\langle\langle \text{fillGlassUsingBottle}, \langle 1, 0 \rangle \rangle\rangle$
F	3	$\{((4, \{1\}), 0), ((4, \{0\}), 1), ((4, \emptyset), 2)\}$	ε

This time q is a failing state so we can conclude that the result of the run is *reject*.

6.3 Inference algorithm

In this section, we informally describe an inference algorithm that generalises Blue-Fringe algorithm (see Section 2.3.1 on page 18). Blue-Fringe is a particular case of the algorithm described in this section, in which every symbol has exactly zero parameters.

Blue-Fringe, and by extension our algorithm, takes as input a set of traces classified as positive or negative. These traces are generally used to represent valid and invalid behaviours of software systems respectively. For illustrative purposes, we will assume that symbols represent function calls in a program execution, and that parameter dependencies represent the use of the results of previous function calls as parameters (data flow), but traces could represent other kinds of events too.

Because a negative trace indicates erroneous behaviour, if a trace is negative then all extensions of that trace (other traces that have it as a prefix) must be negative as well. Therefore, we only consider minimal negative traces.

Our algorithm consists of four steps that must be carried out sequentially: dependency rewriting, APTA generation, state merging, and transition merging.

6.3.1 Dependency rewriting

The first step is to rewrite the parameter dependencies for the transitions. The rewritten notation presented in this section is more flexible than the input notation presented in Section 6.2.1 (on page 164), thus the input notation is more convenient to describe the constraints of valid input (it is canonical and its structure already makes some invalid inputs impossible to represent). The rewritten notation is more appropriate for the algorithm itself, since the algorithm obtains values from the place where they were last used, and the rewritten notation describes symbols in terms of their last usage. In fact, with the rewritten notation we do not need to modify the APTA generation part of the algorithm, we can use the same approach as blue-fringe.

6.3.1.1 Pointing to last usage

The rewriting consists in replacing the numbers that represent the parameter dependencies that have already been used earlier in the trace with pointers to the place where they were last used. If we consider again the trace:

$$t_1 = \langle (\text{createBottle}, \varepsilon), (\text{createGlass}, \varepsilon), (\text{openBottle}, \langle 0 \rangle), \\ (\text{fillGlassUsingBottle}, \langle 1, 0 \rangle) \rangle$$

After rewriting we would obtain:

```
[createBottle [], createGlass [], openBottle [RetVal 0],
 fillGlassUsingBottle [RetVal 1, ParamNum 0 of 2]]
```

where `RetVal N` represents the result value of the symbol in position `N` (would be written as (N, \emptyset) in source notation, see Section 6.2.2.1 on page 166) and `ParamNum N of M` represents the value last used as the parameter in position `N` of the symbol in position `M` (would be written as $(M, \{N\})$ in source notation).

This new notation reflects the effect of passing values as parameters to symbols, for example: the effect of using the result of `createBottle` as parameter of the symbol `fillGlassUsingBottle` (parameter 1) is different if the same result, as in the case, has previously been used as parameter of `openBottle` (the input format does not explicitly represent this dependency).

There is no way of representing alternative sources in this notation, because it will only be used to represent input traces; to represent parametrised automata we will use the formal notation defined in Section 6.2 (on page 164).

6.3.2 APTA generation

The next step is generating an APTA tree (or augmented prefix tree acceptor, see Section 2.3 on page 17). We construct a prefix tree where branches are annotated with the symbols of the rewritten input traces, including the parameters, in the same way it is done by the Blue-Fringe algorithm (see Section 2.3.1 on page 18). The last node of each negative trace will be marked as a failing state and all failing states will be merged together at the end of the APTA generation.

In order to illustrate the process, we will consider an input set with two positive traces (t_1^+ and t_2^+) and two negative traces (t_3^- and t_4^-), all written in input notation:

$$\begin{aligned}
 t_1^+ &= \langle (\text{createBottle}, \varepsilon), (\text{createGlass}, \varepsilon), (\text{openBottle}, \langle 0 \rangle), \\
 &\quad (\text{fillGlassUsingBottle}, \langle 1, 0 \rangle) \rangle \\
 t_2^+ &= \langle (\text{createBottle}, \varepsilon), (\text{createGlass}, \varepsilon), (\text{openBottle}, \langle 0 \rangle), (\text{closeBottle}, \langle 0 \rangle), \\
 &\quad (\text{openBottle}, \langle 0 \rangle), (\text{fillGlassUsingBottle}, \langle 1, 0 \rangle) \rangle \\
 t_3^- &= \langle (\text{createBottle}, \varepsilon), (\text{createBottle}, \varepsilon), (\text{createGlass}, \varepsilon), (\text{openBottle}, \langle 0 \rangle), \\
 &\quad (\text{fillGlassUsingBottle}, \langle 2, 1 \rangle) \rangle \\
 t_4^- &= \langle (\text{createBottle}, \varepsilon), (\text{createGlass}, \varepsilon), (\text{openBottle}, \langle 0 \rangle), (\text{closeBottle}, \langle 0 \rangle), \\
 &\quad (\text{fillGlassUsingBottle}, \langle 1, 0 \rangle) \rangle
 \end{aligned}$$

By applying the Blue-Fringe algorithm directly to the rewritten input traces we would get a diagram like the one in Figure 48.

Note that the numbering used for dependencies (in the parameters of the symbols) depends on the original traces. If we reordered the symbols or inserted new ones in the traces we would have to update the parameters to preserve their meaning. Introducing loops in our model will effectively allow us to insert new symbols in the middle of traces, thus, in order to avoid losing dependency information when introducing loops, we must give transitions a unique identifier and rewrite all parameters in terms of those.

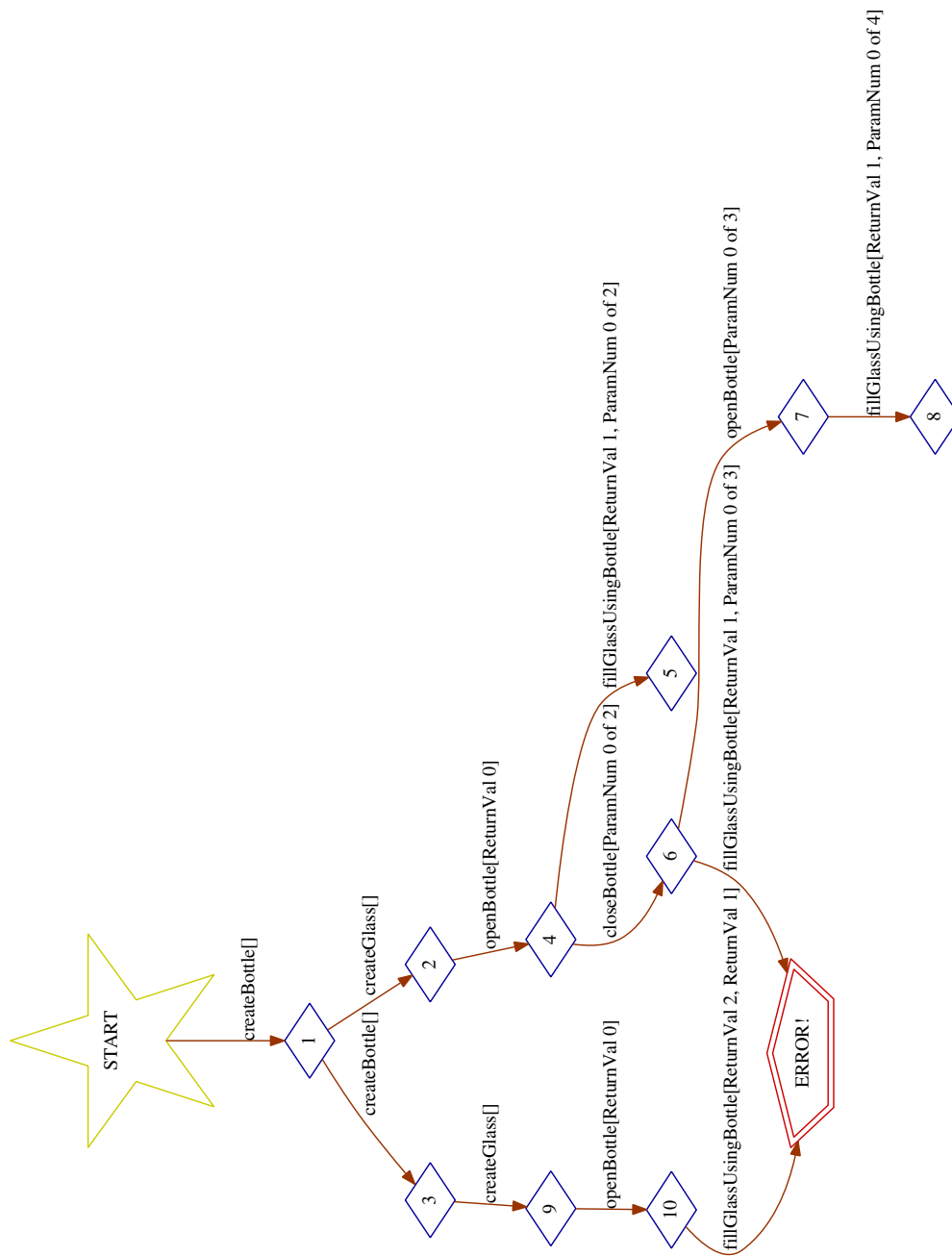


Figure 48: APTA tree as it would be generated by Blue-Fringe

To avoid having several numberings, from now on, in the graphical representations of parametrised state machines, we will illustrate parameter dependencies as black arrows between transitions; internally, in the implementation, we use one numbering for transitions (represented as T in our formal notation) and one numbering for states (represented as Q in our formal notation). By replacing the numbers in the parameters of the symbols with black arrows that point to the corresponding transitions, we obtain a diagram like the one in Figure 49.

In order to be able to draw arrows to and from transitions, we write the symbol name inside a brown elliptical node in the middle of the transition arrows (the brown arrows). These nodes do not represent states of the parametrised automaton, but are part of the transition representation itself (just a representation artefact).

To make this clearer, we draw both the transition nodes and the transitions in the same colour (brown), and the actual nodes that represent states are drawn in blue, are diamond shaped, and have a number on them (except the starting state which we represent as a yellow star with the label **START** on it, and the failing state, which we represent as a red pentagon with double outline and the label **ERROR!**).

6.3.3 State merging

Next step is to merge potentially equivalent states, similarly to how it is done by the Blue-Fringe algorithm (see Section 2.3.1 on page 18).

6.3.3.1 Difference from Blue-Fringe

The difference with the Blue-Fringe algorithm is that we need to slightly redefine what we mean when we say two symbols or transitions are different. We will consider two transitions to be different if any of the following conditions apply:

- They have different names (different symbol names or labels).
- They have a different number of parameters.
- For any particular parameter (for example: second parameter), each of the transitions depends on transitions that have different identifiers (in other words: they depend on different transitions).

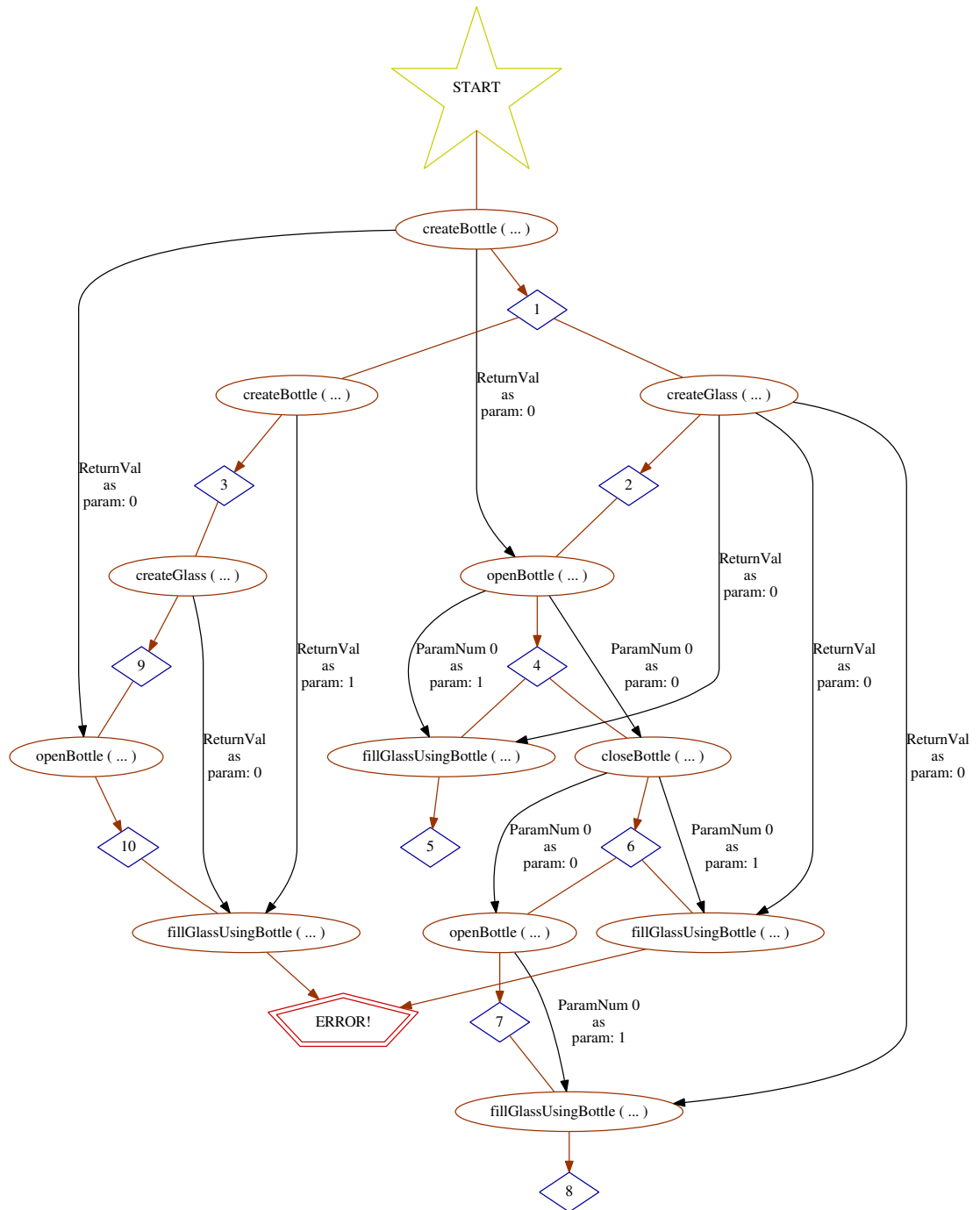


Figure 49: APTA tree as generated by the parametrised algorithm

6.3.3.2 Merging procedure

APTA generation is a conservative initial approach that will create lots of states, but in most cases several of these states will actually be equivalent and represent the same state of the system. We want states in the model that represent equivalent states of the system to be represented by the same state in the graph, and we achieve this by merging them.

The state merging process tries to reduce the number of states by iteratively merging the states that are indistinguishable by using the available information. We will only assume that two states are not equivalent if one of the following is true:

- One is the failing state and the other one is not.
- Both states are normal, but both are a source of equal (that is: not different) transitions that bring us to a pair of states that are not equivalent.

Note that we assume as an invariant that no transitions depart from a negative state since we only consider minimal negative traces and we never merge the negative state with a normal one. For simplicity, we start by merging all negative states into one, and we refer to it as “the negative state”.

For merging two nodes and choosing which nodes to merge first we use the same procedure from Blue-Fringe, described in Section 2.3.1 (on page 18), where transitions are considered different following the criteria defined in Section 6.3.3.1 (on page 180).

The process finishes when no more nodes can be merged, that is: all nodes remaining in the model are unmergeable according to the criteria defined in Section 2.3.1 (on page 18).

6.3.3.3 Solving non-determinism

We already defined the concept of *unmergeable* nodes when we talked about Blue-Fringe (see Section 2.3.1 on page 18). *Unmergeable* nodes are those that, if merged, would produce non-determinism that, in order to be solved would force us to merge a positive and a negative state.

There is non-determinism whenever two equal transitions that depart from the same state have the same dependencies (at this point still one per parameter), and

go to two different states.

The existence of non-determinism is undesirable since checking whether a trace is accepted by a non-deterministic state machine is more expensive in terms of both space and execution time. For this reason, whenever we introduce non-determinism, we immediately try to resolve it by merging the alternative destination states of the conflicting transitions.

This process can end in two ways, either:

1. We eventually remove all the instances of non-determinism.
2. At some point, we will be forced to merge a normal state with the failing state (which we cannot do since we know they are not equivalent). If this happens, we will roll-back the whole chain of mergers, and mark the pair of nodes that triggered the chain of mergers as *unmergeable*.

The state machine resulting from the process is guaranteed to be deterministic, since the APTA tree was deterministic, every chain of merges is initiated from a point where the state machine is deterministic, and we roll-back chains of merges whenever they introduce non-determinism.

6.3.3.4 Example

The result of applying the state merging process to the tree in Section 6.3.2 (on page 178) is shown in Figure 50 (on page 185).

We can see that the initial state (represented by a star), and the failing state (represented by a red pentagon) are the only ones remaining.

This is not always the case, but when it happens it means that the state of the values used as parameters (that can be seen as objects) is enough to explain the observed behaviour of the system, in other words: it is not necessary to consider a global state in order to understand the system. A global state would be necessary if there were, for example, global variables or implicit relationships between the different values (as in the case of implicit object, described in Section 5.9.1 on page 157).

Nevertheless, we can see that if we consider the data flow (the dependencies represented as black arrows) there are no loops, black arrows always go downwards in the graph.

Data flow can be seen as the representation of the “state” of the different “objects”. The initial “state” of an “object” would be represented by its *constructor transition*, where a *constructor transition* is one that does not have any incoming black arrows, like `createGlass` and `createBottle`.

If we start from a constructor transition, and we follow the data flow, we find the transitions that use the “objects” generated by the constructor. These transitions can be seen as forming embedded PTAs for each object, where each transition is an event that occurs to the object. These embedded PTAs overlap since there are transitions that affect several objects. For example, if we start from `createBottle`, we can get to `openBottle`, and then to `closeBottle`; or we can get directly to `fillGlassUsingBottle`. If we start from `createGlass`, the only transition we can obtain is `fillGlassUsingBottle`.

The logical step now would be to generalise these embedded PTAs so that we introduce loops in the data flow, and we obtain a more general behaviour. We can do this by merging equivalent transitions in a similar way to how we merged equivalent states during the state merging process (Section 6.3.3 on page 180). We can quickly see that there are some transitions that do the same thing and are, nevertheless, replicated. For example, if we look at the two transitions labelled `openBottle`, we can see that one takes the result of `createBottle` and one takes the result of `closeBottle`. But both have the same effect and both go from the start state to itself; there does not seem to be any reason why they could not be represented as one single transition with two alternative sources. Indeed, we will see that the result of merging transitions produces a model with only one `openBottle` transition.

In Section 6.3.4 we describe in more detail how to decide whether two transitions are equivalent and how to merge them when they are.

6.3.4 Transition merging

The reason for both state merging and transition merging is generalisation: APTA trees accept all the positive traces and reject all the negative traces provided as input, but they will not classify any positive traces that are not prefixes of the traces taken as input (the result of running them will be *indeterminate*). The APTA tree by itself just stores the provided traces. For there to be actual learning,

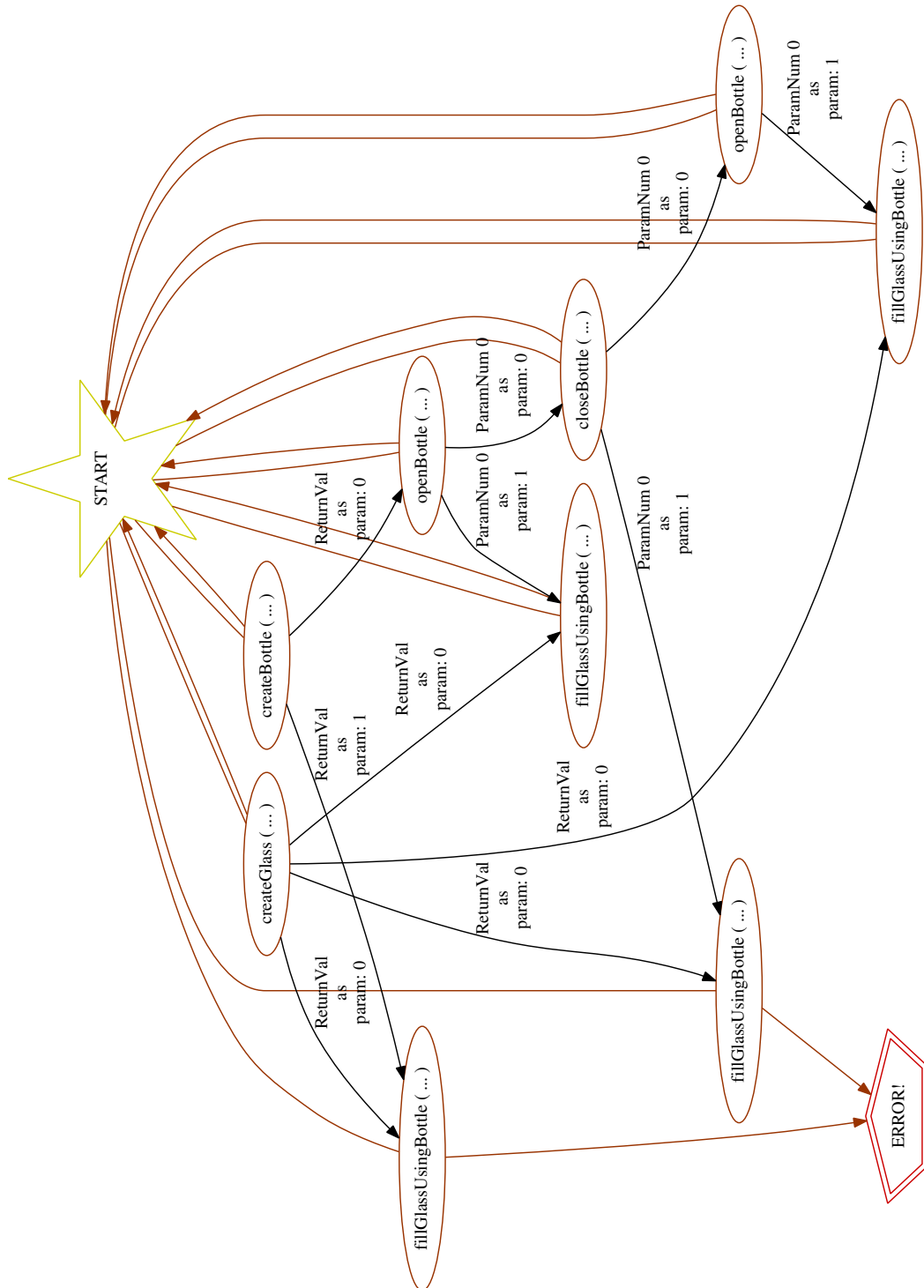


Figure 50: State machine after state merging process

the model needs to be able to make predictions for unobserved inputs, and we achieve this by creating loops in the transitions and parameter dependencies (data flow).

If we think about merging both `openBottle` transitions in Figure 50, we see that the source of their first parameter (parameter 0), is different: one comes from the result of `createBottle`, and the other one comes from being used as a parameter by `closeBottle`, thus, according to the third rule in the list of rules that we outlined in Section 6.3.3.1 (on page 180), both transitions are different.

In order to merge both `openBottle` transitions, we need to be able to have one transition that has two alternative sources for the same parameter. This seems simple in this case, but allowing for more than one possibility in each parameter has some implications in the general case, and it makes checking for non-determinism more difficult.

For example, imagine we model the `win` function in the rock-paper-scissors game, where two players simultaneously choose one item each and the winner is selected depending on the items chosen according to the following rules: paper beats rock, rock beats scissors, and scissors beats paper. If both players choose the same item the game ends in draw. We can consider that both `win(scissors, paper)` and `win(paper, rock)` are normal transitions, but we cannot merge both to form `win(scissors or paper, paper or rock)` since that would imply also `win(scissors, rock)`, which is incorrect. But if we have a function `win_or_draw`, we can merge `win_or_draw(scissors, paper)` and `win_or_draw(paper, paper)` to form `win_or_draw(scissors or paper, paper)`.

6.3.4.1 Semantics and restrictions

It is thus necessary for us to specify a more general semantics and representation for transitions that allows for several alternative sources to be used as parameters:

- Each parameter will contain a non-empty list of distinct alternative sources.
- We will consider that a symbol in a trace exercises a transition only if it uses as source, for each parameter, one of the alternatives listed by the transition for that parameter.
- Each time a transition `t1` is traversed, we will consider that a value is produced and accumulated in the transition. When, at some point, another

transition `t2` that depends on `t1` is traversed, it will remove/use the accumulated value on `t1`, and accumulate it in the slot of that parameter of `t2` instead. For example, when traversing a state machine, we are not allowed to obtain the following sequence of rewritten symbols (rewritten as described in Section 6.3.1 on page 177):

```
createBottle [], openBottle [ReturnValue 0],
openBottle [ReturnValue 0]
```

second `openBottle` does not have any value to use. If we were to represent the behaviour of opening the bottle twice we should do it by using a trace like the following:

```
createBottle [], openBottle [ReturnValue 0],
openBottle [ParamNum 0 of 1]
```

Note that, for example, if we traverse `createBottle` twice before using its result, we may have two values stored in the same node at the same time, this just means that we can use either of them. But this does not cause non-determinism about the result of which instance of `createBottle` to use in a particular trace since the input format specifies exactly which instance to use.

If our transition merging algorithm iteratively merges pairs of transitions that have the same source and destination state, and it does not introduce non-determinism in terms of the new semantics, our state machine will still fit the input data provided and, thus, the inference algorithm will remain sound.

Analogously to state merging, if merging a pair of transitions produces non-determinism, there is still a chance that we can solve the non-determinism by merging the conflicting transitions. Otherwise, we roll-back the chain of mergers.

6.3.4.2 Transition non-determinism

Under the new semantics, we say that two transitions belong to the same *source-group* if all of the following conditions apply:

- They depart from the same node.
- They have the same label (or symbol name).

- They have the same number of parameters.

Two transitions that belong to the same *source-group* will produce non-determinism if, for each of the parameter positions (for example: second parameter), there is at least one common source within the alternatives of both of the transitions. For example:

- `win_or_draw(scissors or paper, paper)` and `win_or_draw(paper, paper or rock)` would produce non-determinism (we would not know to which transition assign traces that contain `win_or_draw(paper, paper)`).
- `win_or_draw(scissors or paper, paper)` and `win_or_draw(paper, rock)` would not produce non-determinism (we can decide based on the last parameter).

Thus, merging two transitions can produce non-determinism, not only in their source state but also in the transitions that depend on the values produced by each of the merged transitions (the transitions that have them as a parameter source).

In other words, we can guarantee that non-determinism is not introduced if we make sure that for all transitions in the same *source-group*: each of the transitions in the group has at least one parameter position for which the transition only contains alternatives that are unique for that parameter position in the group.

For example, in the following set of symbols that belong to transitions in the same *source-group*:

- `valid_options(a or b, a, a)`
- `valid_options(c, a, b)`
- `valid_options(c, b, a)`

we know there cannot be non-determinism because:

1. The first trace has both `a` and `b` as unique elements in the 1st parameter.
2. The second trace has `b` as unique element in the 3rd parameter.
3. The third trace has `b` as unique element in the 2nd parameter.

6.3.4.3 General approach

Since merging a single pair of transitions may produce non-determinism with the transitions in the same group, our implementation recalculates the potential transitions that can be merged after merging each pair. Currently, our only heuristic when choosing which transitions to merge first is to start with the constructors (transitions that have no parameter dependencies).

It is left for future work to research more efficient ways of ordering transition merging.

6.3.4.4 Updating the formal model

In terms of the formal model, given two transitions with identifiers a and b such that $\delta(a) = (q, l, \langle s_{a,1}, s_{a,2}, \dots, s_{a,n} \rangle, q')$ and $\delta(b) = (q, l, \langle s_{b,1}, s_{b,2}, \dots, s_{b,n} \rangle, q')$. The result of merging both transitions will be a new model in which, in its transition function index δ' , there is a transition with identifier c that is defined as $\delta'(c) = (q, l, \langle s_{a,1} \cup s_{b,1}, s_{a,2} \cup s_{b,2}, \dots, s_{a,n} \cup s_{b,n} \rangle, q')$, and where the original transitions with identifiers a and b are not defined (unless, of course, the identifier c is equal to a or equal b). The set T must also be updated to reflect exactly the preimages of δ' that are defined.

6.3.4.5 Example

Figure 51 shows the result of applying transition merging to the state machine from Section 6.3.3 (on page 180).

We can see that all duplicate transitions have been merged except for the ones labelled `fillGlassUsingBottle`. The reason why these were not merged is that they have different destination states, one loops on the initial state, and the other one goes to the failing state. The algorithm has also detected equivalences between using the result of `createBottle` and using the result of `closeBottle`.

6.3.4.6 Frequency server

The Frequency server, as implemented in previous sections, does not benefit from the data abstraction provided by the parametrised automaton. This is due to the implicit state overlapping between the global state of the Frequency server itself and the state of each frequency with respect to the server (whether it is allocated

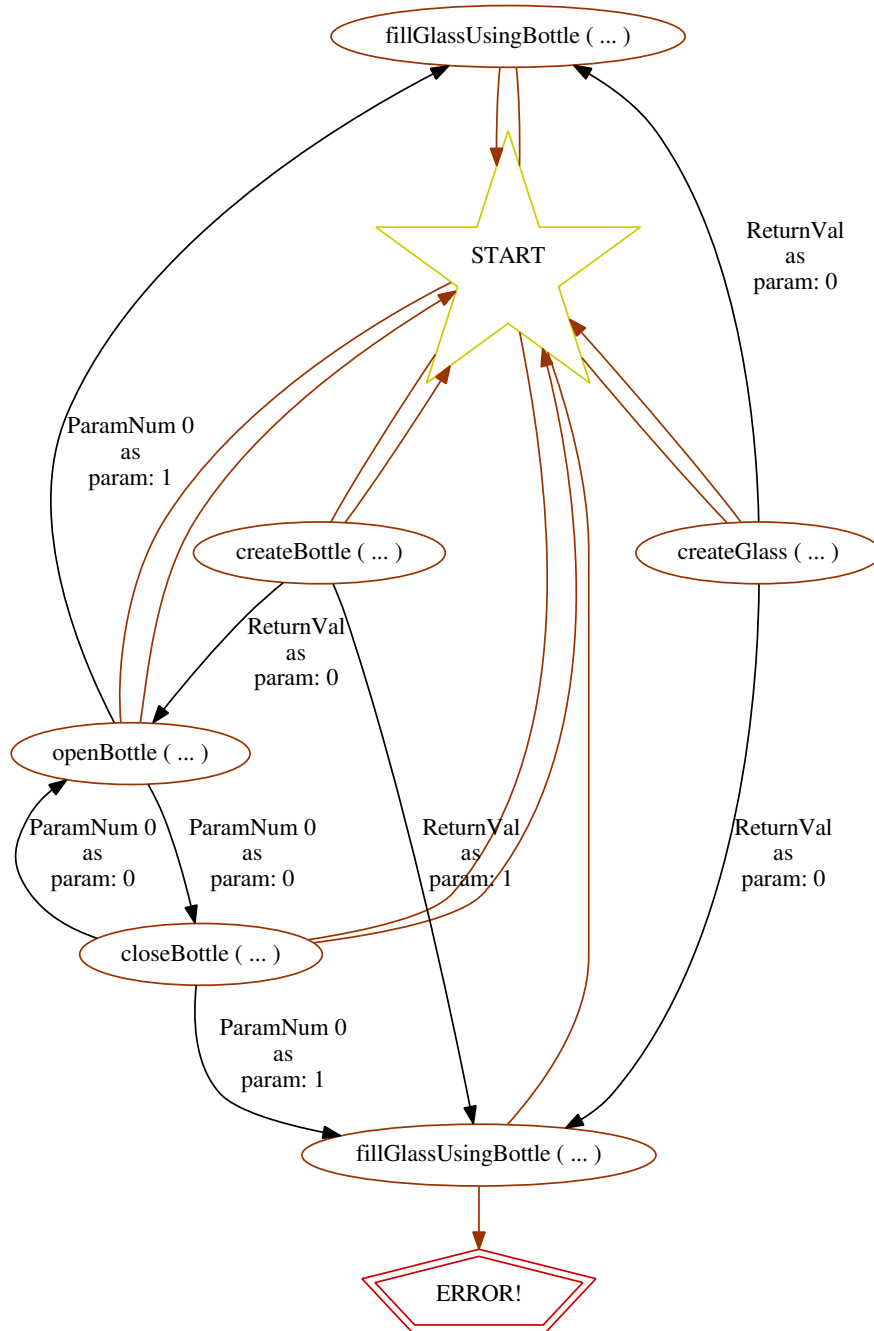


Figure 51: State machine after transition merging process

or not by it). When the Frequency server is turned off, all frequencies change state to deallocated, but because the `stop` transition does not take any frequencies as parameter, the only way for our parametrised automaton to represent this is by creating a new state, and by using fresh frequencies every time; this interpretation would lead to a parametrised automaton that has infinite states, which is much worse than what we get by using normal FSM inference (or, equivalently, by not specifying any data dependencies in the traces).

In practise, we can see the call to `stop` as implying a series of calls to `deallocate`. If we decouple `stop` and `deallocate`, we can think of a hypothetical Frequency server in which `stop` does not forget about the frequencies allocated, and in this scenario our inference algorithm can, indeed, infer a finite parametrised automaton. In Figure 52, we can see what a parametrised automaton for the alternative Frequency server with two frequencies looks like (without the negative transitions and state). We obtained this automaton by taking a set of traces chosen manually and then iteratively carried out the following two steps, using the Haskell implementation provided in (Lamela Seijas and Thompson 2016b):

1. Classify the traces using the model implementation (available as the function `freqServerUTL` in the module `Inference` of the implementation). Use the inference algorithm to generate a parametrised automaton from the classified available traces.
2. Generate a random traversal of the parametrised automaton and check whether the classification given by the current parametrised automaton matches the classification given by our model implementation for it:
 - If the classification is different, add the last traversal to the list of traces and continue on step 1.
 - If the classification is the same, continue on step 2 with a different random traversal. If this branch is taken many times in a row then we stop and use the current parametrised automaton as result.

We can see that:

- `START` state is stopped and has no frequencies allocated.
- State 1 is running and has no frequencies are allocated.

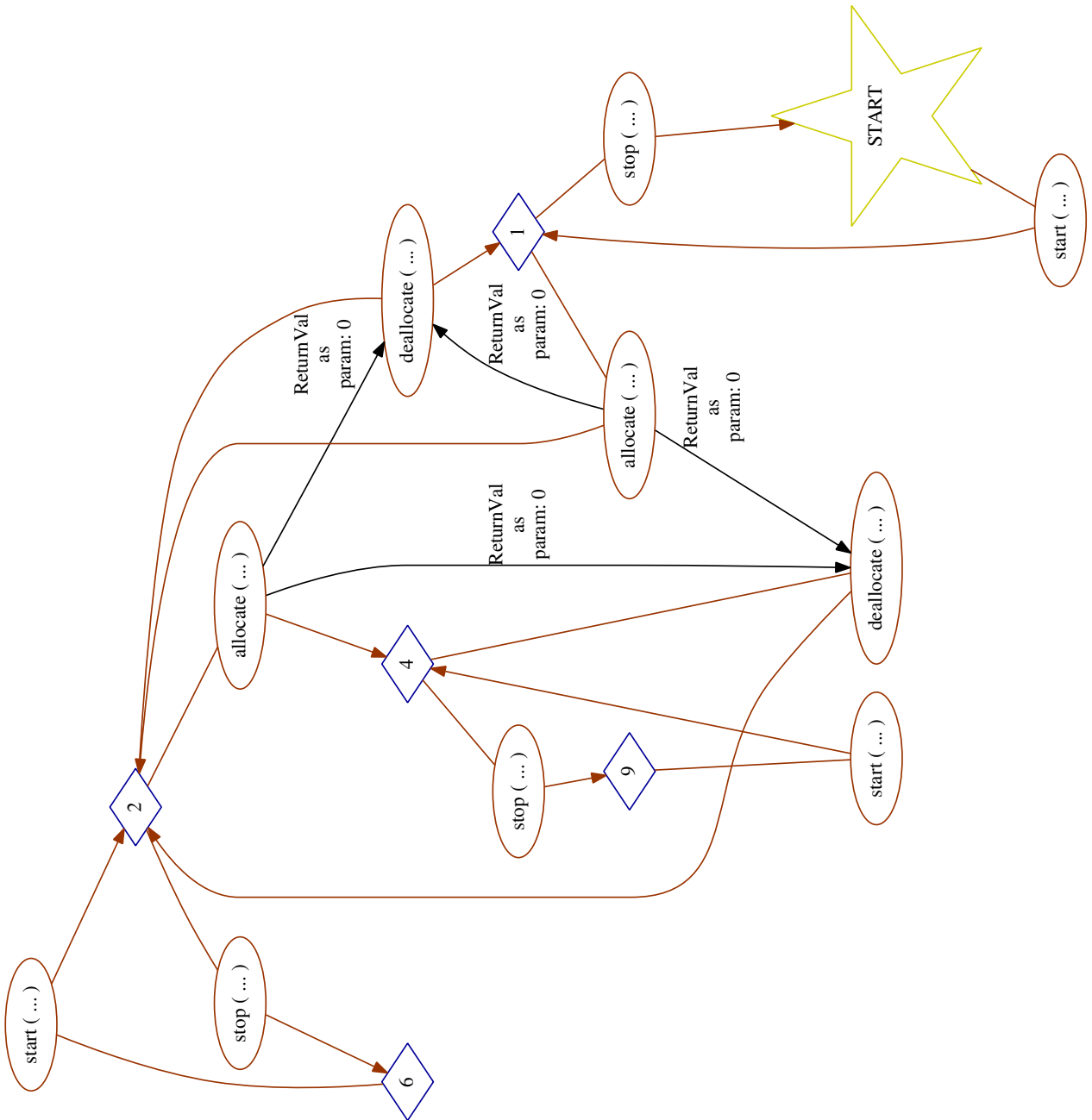


Figure 52: Parametrised automaton for finite Frequency server

- State 2 is running and has one frequencies allocated.
- State 4 is running and has two frequencies allocated.
- State 6 is stopped and has one frequency allocated.
- State 9 is stopped and has two frequencies allocated.

The diagram generated (Figure 52) does not seem to be much simpler than the ones we obtained for the two frequency configurations of the real Frequency server implementation in Section 4.4 (on page 103). But, curiously, we can obtain a very simple model if we remove the frequency limit on the Frequency server implementation and use the same procedure. The same Frequency server implementation when configured for allowing the allocation of an arbitrarily big number of frequencies produces the parametrised automaton in Figure 53.

Arguably, even though we can justify the manual decoupling of the behaviour of `stop` and `deallocate`, we may want to model a system in which `stop` cannot occur if there are any allocated frequencies. This would not necessarily require an infinite number of states with the model as it is defined in this chapter, but it would require at least as many states as frequencies, which makes it impossible to get a model for the general case with an arbitrary number of frequencies. This suggests that future work may want to add new types of data dependencies, in particular, an “inverse dependency” type, that can only be satisfied if a normal dependency of the same type would be impossible to satisfy. In our example, an “inverse dependency” could be used from `allocate` to `stop` to describe that there cannot be frequencies stored in `allocate` before calling `stop` successfully.

6.4 Soundness

We have mentioned that the soundness of our approach has been tested (Lamela Seijas and Thompson 2016b) through the use of a QuickCheck property (using QuickCheck for Haskell), this property:

- generates a random set of input traces (both positive and negative),
- infers a parametrised automata from the set of inputs (using the algorithm),

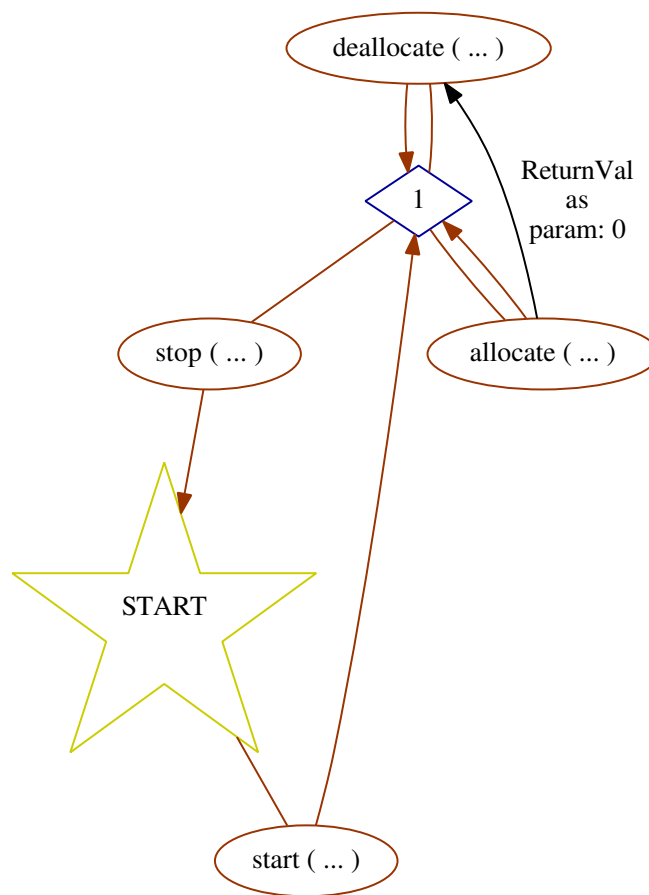


Figure 53: Parametrised automaton for infinite Frequency server

- picks one of the inputs,
- checks that the chosen input is classified correctly by the inferred automaton.

This property also checks indirectly that the algorithm produces automata that are deterministic for the traces that are given as input.

Ideally, we could use a proof assistant to verify that this property holds for every input, and, additionally, that the inferred automaton is deterministic for any input (independently of whether it was provided to the inference algorithm as input), but we leave that to future work.

However, we will sketch how the soundness proof could be approached. We foresee that it could be done through the following steps:

- Implementation of the inference algorithm. This is provided already (Lamela Seijas and Thompson 2016b), but it could be reimplemented in a way that is easier to verify.
- Implementation of a total function that takes an input trace and an automaton, and returns whether: the trace is accepted, the trace is rejected, the trace is not considered by the automaton, or there exists non-determinism in the automaton that affects the evaluation of the trace. This is also provided already (Lamela Seijas and Thompson 2016b).
- We can verify that the soundness property holds for the APTA. This should be straightforward since an APTA is just a recursive data structure that stores the trace information in the input format (also known as a trie).
- Then, we can verify that the state merging process does not introduce non-determinism nor loses information about the traces provided to the inference algorithm.
 - We know it does not introduce non-determinism because, at each step, it looks for non-determinism, and it either solves it or rolls back. So it should not be possible for there to be non-determinism at the end of the process.
 - In the same way, we can intuitively see that it does not lose information because no nodes from the APTA tree are deleted, they are only merged

with nodes that are equivalent, thus the traversals that were possible before a merger are also possible after it.

- Finally, we can verify that the transition merging process does not introduce non-determinism nor loses information about the traces provided to the inference algorithm. This seems to be the case since the process:
 - takes pairs of transitions that go between the same pair of states,
 - aims to combine them into transitions that subsume them (thus, the information from both the original pair of transitions is contained in the transition resulting from the merger),
 - and it rolls back whenever two transitions of the result are detected to produce non-determinism.

There are many details in this sketch that we have not discussed but are necessary for the soundness of the approach: like the requirement for all parameters to be different, or the uniqueness of the values used as a parameter. But we hope the previous overview gives confidence on the soundness of the algorithm and provides insight on how the proof could be approached.

6.5 Chapter conclusions

In this chapter, we have studied an alternative approach that tries to address one of the causes of state explosion by augmenting the notion of regular state machine and adapting the existing regular inference algorithm Blue-Fringe to work with the new definition.

Throughout this chapter, we have shown how to apply the ideas explored in Chapter 5 without losing soundness, and we have given some new evidence of the formal feasibility of adding parameters to symbols, and combining control and data flow in state machines.

The approach presented in this chapter has not been validated in practice (by using industrial examples), in the way it was done for the one presented in Chapter 5. This is mainly due to time constraints in the development of this thesis.

We acknowledge that the approach presented in this chapter may not be more convenient in practice than less precise approaches (like James or fuzz testing), since it still suffers from some kinds of state explosion, and it surely is not as efficient as it could be. But it is valuable in that it shows how we can obtain richer models without losing soundness or learnability.

Thus, future work should push the flexibility of the model further, to get rid of state explosions definitely without losing soundness nor learnability. If we manage to do that, the resulting system will necessarily be practical, if not for test generation from legacy tests, then maybe for reverse engineering, for generation of documentation, or even for automatic programming.

Chapter 7

Related Work

Throughout the previous chapters, we have covered several aspects of model inference, testing, and refactoring. In Chapter 2, we have covered the work on which this thesis rests, since it was necessary for its understanding. In this chapter, we look at previous existing work that exposes alternative or similar ideas to the ones presented here.

7.1 Testing web services

In Chapter 3 we studied the reuse of testing components by focussing on modelling one particular abstract aspect of web services. This work is by no means the first or only attempt to test or automate the testing of web services or REST web services.

Regarding REST web services, there exist several approaches aimed at ensuring the quality and RESTfulness of web services. This is the case of Test-the-REST (Chakrabarti and Kumar 2009), which provides a mechanism for specifying and validating test cases. In addition, there have been approaches aimed at validating the RESTfulness of web services by focussing on their connectedness (Chakrabarti and Rodriguez 2010) and by using temporal logic and model checking (Klein and Namjoshi 2011).

On the other hand, property-based testing has been used in the past for testing SOAP web services in an automated way.

(Lampropoulos and Sagonas 2012) shows a way to create properties to test SOAP services by using WSDL descriptors. Their method generates a set of

simple properties without the need of human intervention, and then they use these properties as a template for writing more complex ones.

Despite the fact that WSDL descriptors are a de facto standard for SOAP web services, they are rarely present in REST web services. For this reason we have used a different approach for the elaboration of generators. Nevertheless, (Lampopoulos and Sagonas 2012) describe a method for building custom generators for XML (see Section 3.2.3 on page 31) from XML Schemas. A similar approach could be used for JSON (see Section 3.2.2 on page 30) as an alternative to the mechanism used in this work.

Haskell QuickCheck has been used for SOAP web service testing (Zhang, Fu and Qian 2010), and automatic approaches for testing non-RESTful web services using both FSMs (Andrews, Offutt and Alexander 2005) and EFSMs (Keum et al. 2006) have been described in previous work.

(Lastres Guerrero 2012) describes the application of property-based testing (in particular QuickCheck `eqc_statem`) to a specific REST web service called Wriaki. Both the approaches describe a property-based model for testing REST web services. Our model in Chapter 3 differs in that we have used `eqc_fsm` in addition to `eqc_statem`, and that we reused it in two different web services and provide a technique for adapting it.

7.2 Modelling differences

In Chapter 4, we studied how to parametrise test models and software, in terms of their source and behaviour.

7.2.1 Source parametrisation

There exist many refactoring tools that aid the generalisation and parametrisation of code, one example is the Function Extraction refactoring included in Wrangler, prior to this work. Our integrated approach can be used (together with a tree matching algorithm) to suggest how such a generalisation can take place; it does not just provide a transformation that can be used manually.

7.2.1.1 IntelliJ IDEA

We can consider that the creation of behaviours in Erlang is conceptually equivalent to the creation of a superclass in Java, and IntelliJ IDEA (Jet Brains 2001) provides a refactoring called “extract superclass”, in particular with the option “rename the original class so that it becomes an implementation for the newly created superclass”, that conceptually achieves a similar aim to the interactive approach described in Section 4.3.1 (on page 62).

Nevertheless, unlike the refactorings described here, the IntelliJ IDEA refactoring only rearranges the methods and fields of the original class, without actually modifying their code or content (it does not do any function extraction).

In addition to this fundamental difference, the interactive refactoring differs from the IntelliJ IDEA refactoring in that it can be applied incrementally, and in that it is implemented and described in high-level, through Wrangler’s DSL, (thus demonstrating how complex refactorings can be built in terms of smaller ones).

The integrated approach, described in Section 4.3.2 (on page 72), differs from IntelliJ IDEA in that it takes two similar modules instead of one single module, and in that (when combined with a tree matching algorithm) it automatically searches for similarities in the code and automatically abstracts them out, leaving in the original modules those parts that are specific to them. Thus, while the aim of the refactoring is the same (that is: creating an abstraction), the process in the integrated approach has more to do with clone elimination than with IntelliJ IDEA’s refactoring.

7.2.1.2 Automatic generalisation

The work that is most similar to our integrated approach (described in Section 4.3.2 on page 72) is (Chawathe et al. 1996), which strongly inspired this work and applies tree comparison to the detection of differences between \LaTeX documents, and generates a \LaTeX file whose output highlights those differences. Obviously, this work was also inspired by the PLTSDiff work (Bogdanov and Walkinshaw 2009) also presented in Section 4.2.4.1 (on page 60).

Tree comparison has been researched extensively (Bille 2005), in big part targeted at XML (Peters 2005). The nature of XML is quite different to the one of Erlang, but they both share the use of nested structures and semi-structured

data.

7.2.1.3 Clone detection and elimination

Although fundamentally different, our integrated approach has commonalities with clone detection and elimination. Both try to find repeated code patterns and to abstract them to a single function. Our integrated approach is different in that it tries to map the structure of two particular modules.

There is an extensive literature on software clones: (Roy and Cordy 2007) surveys work in the area up to 2007, and there is also a regular International Workshop on Software Clones. Existing work on software clones places an emphasis on clone identification, analysis, and classification (Kapsner and Godfrey 2003), as well as on the evolution and tracking of clones.

7.2.1.4 Remodularisation

The refactorings for the introduction of behaviours described in this thesis are also related to the extensively researched topic of remodularisation (Hall 2013; Seng et al. 2005; Wiggerts 1997; Mitchell and Mancoridis 2006), in that both approaches aim to rearrange existing code in order to put related parts together and, thus, make maintenance easier.

Nevertheless, remodularisation often studies ways of reorganising functions in modules that are related semantically; that is, it analyses the relation between functions and tries to cluster similar functions by moving them to the same module in a way that a particular heuristic is optimised.

Our integrated refactoring also looks for similarities and puts them together; but those similarities are syntactic similarities, and the integrated refactoring does not only put similarities together in the same module, but it also combines them in such a way that replication is removed and only one instance remains.

On the other hand, remodularisation often targets whole systems (all functions in a system), and usually treats functions as indivisible units (it does not try to divide functions); whereas our approach targets very specific parts of the systems (one or two modules selected by the user, or even a single function), and may reorganise the internal structure of existing functions and create new ones.

For all these reasons, the approach carried out by the refactorings presented

in this thesis can be seen as complementary to most approaches that would be classified as modularisation. However, previous to this thesis, Wrangler already provides a series of inspection and refactoring mechanisms that support modularisation in the traditional sense, by aiding users on the detection and elimination of “modularity smells” (Li and Thompson 2010).

7.2.2 Model parametrisation

Regarding model parametrisation, this thesis contributes the case study described in Section 4.4 (on page 103); the PLTSDiff algorithm and tool existed previous to this work and, thus, we do not review state-machine comparison algorithms. Such a survey can be found in the original paper of the PLTSDiff algorithm (Bogdanov and Walkinshaw 2009).

State-machine comparison has been used in the past with different purposes. In (Walkinshaw and Bogdanov 2013), the authors compare the language accuracy and structural accuracy of two inference algorithms, in order to get an insight into their strengths and weaknesses.

7.3 Modelling control and data

In Chapter 5, we studied how to combine control and data flow information in our test models to obtain flexibility, and in Chapter 6 we found a sound way of inferring them.

There have been many previous approaches to specification extraction. (Pradel and Gross 2009; Dallmeier et al. 2006; Marchetto, Tonella and Ricca 2008), model the expected use of interfaces by focussing on the order in which commands are usually executed (control flow). One limitation of these approaches is that they do not usually infer how to create the parameters that the commands require.

There exist many approaches to regular inference in the literature, and we have already presented some of them in Chapter 2.

There have been previous attempts to combine data and control, but previous approaches usually rely on data representation, either for clustering (Berg, Jonsson and Raffelt 2006), or by inferring invariants for parameters and then using them to disambiguate the commands in FSMs (Lorenzoli, Mariani and Pezzè 2006;

Shahbaz, Li and Groz 2007).

These approaches have limited effectiveness when inferring complex properties, or arbitrary semi-structured data, and they do not take advantage of the dependency information provided by legacy unit tests. On the other hand, these approaches have the advantage of being suitable for black-box interfaces.

Our approach abstracts away from particular values of the data parametrised, relying instead on how the data is generated and used by actions within the system: parameters are treated as black boxes.

There has also been some work on inference of richer models like Visibly Push-down Automaton (Isberner 2015), EFSM (Walkinshaw, Taylor and Derrick 2016; Cassel et al. 2016), and context-free grammars (Wyard 1993; Javed et al. 2004); the most ambitious approaches often rely to certain extent on general techniques like machine learning, evolutionary programming, and SMT-solvers.

In (Biermann and Krishnaswamy 1976), the authors present an interactive system for inferring programs from examples of their execution, but it depends on the details of the actual algorithm to infer (users must specify the conditions considered when branching, the organisation of the algorithm in functions, and recursive calls).

The work implemented on Strawberry tool (Bertolino et al. 2009) is probably the most similar approach to the one presented in Chapter 5, it also models control and data flow information for extracting specifications of web services, and uses testing to verify conformance.

However, in Strawberry, control flow is inferred from data flow, whereas our approach extracts data and control information simultaneously. Our approach can do this because it takes examples of execution as input, whereas Strawberry takes a WSDL and examples of input data. Chapter 6 goes further by formally defining an extension to the FSM model.

It can be argued that the use of concrete data content in models can make them easier to understand. But abstracting it out allows us to apply our approach to parameters that cannot be serialised, or that have a representation that is too lengthy or complicated.

Test generation The main topic researched in Chapters 5 and 6 is not test generation but model inference. Nevertheless, in Chapter 5, we talk about how to

create models that generate tests, even though the actual test generation is done by QuickCheck, which is a tool existing previous to this work. Chapter 6 does not talk about test generation. Even though this thesis does not contribute much in the realm of test generation, it does present a framework that, as a whole, aims to generate new tests, and there has been many approaches in the past to generating new tests, to extending existing unit tests suites, and to using models to generate tests; thus, it is worth looking at existing work on test generation. We present a brief overview of some representative approaches that are similar to ours, and we describe how they differ. A much more extensive survey on test generation techniques and the use of oracles can be found in (Harman et al. 2013).

One of the reasons why our approach differs from others is in that many are stateless, they consider tests as input-output relations. As we mentioned, James is targeted at stateful systems, so it models side-effects and sequences of events whereas, for example, (Harder, Mellen and Ernst 2003), produces specifications in terms of the inputs in a broad sense.

Sequences of events can also be considered inputs, but a general approach would foreseeably be less effective for finding invariants in such an structured input; in the case of Daikon (Ernst et al. 2007), it would require a specific algebra (invariant building blocks), whereas James is designed for sequences of events.

Another difference with (Harder, Mellen and Ernst 2003) is that models constructed by James depend on the way the JUnit tests used as input are written, whereas the approach described by (Harder, Mellen and Ernst 2003) does not necessarily depend on the tests used as input, in fact, it does not require any tests at all.

Regarding stateless test generation, there is also a big literature about techniques like fuzzing (Sutton, Greene and Amini 2007), and the use of symbolic execution for test generation (Păsăreanu and Visser 2009).

There are other approaches which are also stateful, like Randoop (Pacheco and Ernst 2007) and model-based testing techniques (Dias Neto et al. 2007); they often rely on the existence of an existing specification or annotations. Other approaches like (Fraser and Arcuri 2011) are guided by other criteria like coverage, those (and many other approaches) usually differ from ours in that they require constraint solving, which in the case of boolean conditions, is an NP-complete problem (SAT), and is undecidable for some constraints.

Chapter 8

Conclusions

The ability of software to behave correctly in response to a variety of different inputs makes it convenient to use but also hard to build correctly. In this thesis, we have studied different ways of using generalisation and models to improve our confidence that software systems behave in the way we expect them to behave. We have also tried to automate this generalisation process and we have seen that one of the greatest difficulties for doing it correctly is imposed by the requirement of knowledge about semantics and the broader context of the system in question.

8.1 Generalisation

Throughout this work we have studied three main approaches to generalising models:

- Manual creation of reusable components for testing.
- Comparison of examples and parametrisation of differences.
- Inference of state machines through the detection of equivalent states (in particular the state of instances of subsystems).

8.1.1 Components

We have shown that, in the same way that common software functionality can be generalised in reusable libraries, testing effort can be reused in the form of abstract models. This opens the door for potential frameworks that allow a modular way

of testing, which would both save effort and potentially increase the effectiveness of tests.

8.1.2 Comparison

One natural way of generalising is by looking at examples and finding common patterns and differences. In Chapter 4 we did this in a very concrete way for both source code and FSMs. As a result, we obtained useful tools that aid the process of generalisation while programming, and, through our experiments, we obtained insight into how subsystems cause replication on state machines and, thus, how to address it.

8.1.3 State merging

Finally, we dedicated two chapters to adding flexibility to state machine models with the aim of addressing one type of state explosion. The state merging idea, upon which we have built our approaches, also relies on finding commonalities: states that are common (equivalent). Our work aimed at finding commonalities in subsystems, represented through dependencies in symbols.

We have both obtained an informal approach that has proven to be useful in practice even though it is not formally sound; and a more formal approach that shows that adding flexibility by merging control and data flow does not necessarily sacrifice either learnability or soundness.

8.2 Models

The concept of model or specification is very broad, and we have discussed that it could even be considered to include the implementations themselves. To make it clearer, at the beginning of this thesis we established a series of properties that a good model should have, namely: simplicity, redundancy, flexibility, and modifiability.

We have aimed to adhere to these properties throughout the thesis, both by building upon existing broadly accepted models that already show these properties, and by working on improving them directly:

- **Simplicity:** we have consistently tried to simplify models by making them smaller and concise. We have done this by allowing systems to be modelled at a higher level (through the use of components, Chapter 3), by automating the removal of redundancies (Section 4.3 on page 61), and by avoiding state explosion (Chapters 5 and 6).
- **Redundancy:** the creation of testing components (Chapter 3) could potentially allow users to reuse testing effort from other users, this would intrinsically provide them with an independent interpretation on how the components should behave. On the other hand, if we infer models from examples (Chapters 5 and 6) or specially from an oracle, we necessarily will be inheriting some information from them; but inferred models shown in this thesis still give users an alternative perspective that can be validated visually, this already provides some redundancy.
- **Flexibility:** Chapters 5 and 6 are mainly about increasing the expressiveness of state machine models through the combination of control and data flow. Section 4.3 (on page 61) shows how to automatically make code flexible (by parametrising its varying parts), and in particular how to automatically make QuickCheck models more flexible.
- **Modifiable:** on the one hand, we can argue that automating the generation of models already makes it easier to change them, since a new model can be generated with less human intervention and replace the old one with little effort. But additionally, we have seen how changes to a model can be parametrised automatically, this directly makes models able to adapt to change by making them more general.

8.3 Future work

In this section, we conclude by writing down our thoughts on how the work done here could be continued, extended, and improved in the future.

8.3.1 Components

The work on components presented here models a very specific (although commonly used) behaviour of web services. In order to obtain a flexible testing framework, *two* approaches suggest themselves:

- More components must be modelled. This is made much easier if standards exist for the implementation of those models. REST and HTTP already go a long way to homogenise the interface of web services, but similar components and their interfaces are still implemented quite differently among different web services.
- Coordination between components must be modelled. The components by themselves are less useful than combined, and in order to test their interactions (integration testing) we must be able to model them.

8.3.2 Comparison

We have presented a series of refactorings which require different levels of user interaction and aid users on the necessary process of generalisation of source code. Future work could find ways of allowing the user to choose what code to generalise and how, for example: by function extraction or by function generalisation; in particular, the ability of comparing specific functions (as opposed to whole modules) has proven convenient.

In general, future work may analyse the work of developers in order to find new fundamental refactorings that act as blocks and can be composed into bigger refactorings, and also new ways of combining them.

Regarding the comparison of state machines, it would be now possible to create an alternative of the PLTSDiff algorithm that works with parametrised state machines.

8.3.3 State merging

We have already used QuickCheck for Haskell to validate the soundness of our inference algorithm for parametrised state machines, but it would be desirable to also obtain a formal proof and to validate it through a proof assistant, this would

give us more confidence, and it may also give us new insights about how we can make it even more flexible without losing soundness.

During our experiments with the inference algorithm we have made several modifications to the implementation with the aim of improving its efficiency, but a careful study could probably find many more improvements. In the same line, future work could also study how to update existing parametrised state machines with new input traces more efficiently (without having to rebuild the whole state machine from scratch).

It would also be useful in practice to have instrumentation mechanisms (in the same way that James has) that would allow users to transform unit tests into parametrised traces. In addition, it would be interesting to have an easy mechanism to compare and synchronise parametrised state models with the systems they model, as done in (Arts, Lamela Seijas and Thompson 2011) with Blue-Fringe.

Parametrised state machines can indeed help reduce some kinds of state explosion and allow us to model systems that would be impossible to model by using “regular” FSMs. Nevertheless, they still suffer from other kinds of state explosion, like that caused by implicit relationships between different values.

One possible improvement that would give parametrised state machines flexibility would be to allow a new kind of parameter dependency (which we may call “dependency unsatisfiability requirement”) that, instead of reusing a value from a source, it requires the source to be empty in order for the transition to be valid (in other words: it would be satisfiable whenever a normal dependency would not). For example, we could imagine a `stop` transition for the Frequency server that is only valid if no frequencies are allocated at a given time (this could be represented with a “dependency unsatisfiability requirement” that points to the result of the `allocate` transition).

It is left to future work to explore these and other kinds of dependencies, and to find out how they can be learned automatically without losing soundness.

Bibliography

- Al-Ekram, R., Adma, A. and Baysal, O. (2005). diffX: an algorithm to detect changes in multi-version XML documents. In *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, IBM Press, pp. 1–11. 4.3.2.1
- Andrews, A. A., Offutt, J. and Alexander, R. T. (2005). Testing web applications by modeling with FSMs. *Software & Systems Modeling*, 4(3), pp. 326–345. 7.1
- Armstrong, J. (2007). *Programming Erlang: software for a concurrent world*. Pragmatic Bookshelf. 2.1
- Arts, T., Lamela Seijas, P. and Thompson, S. (2011). Extracting quickcheck specifications from eunit test cases. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, ACM, pp. 62–71. 8.3.3
- Arts, T. et al. (2006). Testing telecoms software with quviq QuickCheck. In *ERLANG '06: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, New York, NY, USA: ACM, pp. 2–10. 2.4
- Arts, T. et al. (2015). D6.5 – Pilots in property based testing, preliminary version – Pilot report #5. http://www.prowessproject.eu/wp-content/uploads/2012/10/D6.5_final.pdf, [last accessed 17-01-17]. 5.1, 5.7
- Berg, T., Jonsson, B. and Raffelt, H. (2006). Regular inference for state machines with parameters. In *International Conference on Fundamental Approaches to Software Engineering*, Springer, pp. 107–121. 7.3
- Bertolino, A. et al. (2009). Automatic synthesis of behavior protocols for composable web-services. In *Proceedings of the the 7th joint meeting of the European*

- software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ACM, pp. 141–150. 7.3
- Biermann, A. W. and Feldman, J. A. (1972). On the Synthesis of Finite-State Machines from Samples of their Behavior. *IEEE Transaction on Computers*, 21. 2.3, 2.3.2
- Biermann, A. W. and Krishnaswamy, R. (1976). Constructing programs from example computations. *IEEE Transactions on Software Engineering*, (3), pp. 141–153. 7.3
- Bille, P. (2005). A survey on tree edit distance and related problems. *Theoretical computer science*, 337(1), pp. 217–239. 7.2.1.2
- Bogdanov, K. and Walkinshaw, N. (2009). Computing the structural difference between state-based models. In *2009 16th Working Conference on Reverse Engineering*, IEEE, pp. 177–186. 4.2.4.1, 7.2.1.2, 7.2.2
- Bogdanov, K., Walkinshaw, N. and Taylor, R. (2007). StateChum. <http://statechum.sourceforge.net/> [last accessed 14-09-16]. 4.2.4
- Carlsson, R. (2009). Edoc – the erlang program documentation generator. <http://erlang.org/doc/man/edoc.html>, [last accessed 03-08-17]. 1.8.1, 4.1.1, 4.3.2.3
- Cassel, S. et al. (2016). Active learning for extended finite state machines. *Formal Aspects of Computing*, 28(2), pp. 233–263. 7.3
- Castro, L. M. and Arts, T. (2011). Testing Data Consistency of Data-Intensive Applications Using QuickCheck. *Electr Notes Theor Comput Sci*, 271, pp. 41–62. 3.5.2
- Cesarini, F. and Thompson, S. (2009). *Erlang Programming*. O’Reilly Media, Inc. 2.1, 2.1.5
- Chakrabarti, S. K. and Kumar, P. (2009). Test-the-rest: An approach to testing restful web-services. In *Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009. COMPUTATIONWORLD’09. Computation World.*, IEEE, pp. 302–308. 3, 7.1

- Chakrabarti, S. K. and Rodriguez, R. (2010). Connectedness testing of restful web-services. In *Proceedings of the 3rd India software engineering conference*, ACM, pp. 143–152. 7.1
- Chawathe, S. S. et al. (1996). Change detection in hierarchically structured information. In *ACM SIGMOD Record*, vol. 25, ACM, pp. 493–504. 7.2.1.2
- Claessen, K. and Hughes, J. (2011). QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices*, 46(4), pp. 53–64. 2.4
- Claessen, K., Smallbone, N. and Hughes, J. (2010). Guessing formal specifications using testing. In *Tests and Proofs: 4th International Conference, TAP 2010, Málaga, Spain, July 1-2, 2010, Proceedings*, vol. 6143, Springer, p. 6. 1.4, 5
- Clarke, E. et al. (2000). *Counterexample-Guided Abstraction Refinement*, Berlin, Heidelberg: Springer Berlin Heidelberg. pp. 154–169. 5.6.2
- Contentful (2013). Contentful. <https://www.contentful.com/>, [last accessed 05-07-17]. 3.4.1
- Cordy, J. R. (2003). Comprehending Reality: Practical Barriers to Industrial Adoption of Software Maintenance Automation. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, IEEE Computer Society. 4.3
- Dallmeier, V. et al. (2006). Mining object behavior with ADABU. In *Proceedings of the 2006 international workshop on Dynamic systems analysis*, ACM, pp. 17–24. 7.3
- Dias Neto, A. C. et al. (2007). A survey on model-based testing approaches: a systematic review. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, ACM, pp. 31–36. 7.3
- Dupont, P. et al. (2008). The QSM algorithm and its application to software behavior model induction. *Applied Artificial Intelligence*, 22. 2.3.1

- Ericsson AB (1999). Erlang/OTP documentation. <http://www.erlang.org/doc>, [last accessed 19-10-16]. 2.1
- Ernst, M. D. et al. (2007). The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1), pp. 35–45. 1.4, 7.3
- Falleri, J.-R. et al. (2014). Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, ACM, pp. 313–324. 4.3.2.1
- Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. Phd thesis, University of California. 3.2.1, 3.3.2
- Fielding, R. T. (2008). REST APIs must be hypertext-driven. <http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>, [last accessed 13-04-15]. 3.2.1
- Fluri, B. et al. (2007). Change distilling: Tree differencing for fine-grained source code change extraction. *Software Engineering, IEEE Transactions on*, 33(11). 4.3.2.1
- Francisco, M. Á. (2014a). Frequency Server Tests by Interoud Innovation. https://github.com/palas/freq_server_test_ma [last accessed 14-09-16]. 1.8.1, 5.1.1, 5.4, 5.5
- Francisco, M. Á. (2014b). VoDKATV Case Study slides (PROWESS Mid-term workshop). http://www.prowessproject.eu/wp-content/uploads/2014/05/interoud_vodkatv_intro.pdf, [last accessed 23-01-17]. 5.7
- Fraser, G. and Arcuri, A. (2011). Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ACM, pp. 416–419. 7.3
- Freeman, E. et al. (2004). *Head First Design Patterns: A Brain-Friendly Guide*. " O'Reilly Media, Inc." . 2.6.2
- Graphviz (1999). Graphviz - Graph Visualization Software. <http://www.graphviz.org>, [last accessed 24-10-16]. 2.5

- Hall, M. J. (2013). *Improving Software Remodularisation*. Ph.D. thesis, University of Sheffield. 7.2.1.4
- Harder, M., Mellen, J. and Ernst, M. D. (2003). Improving test suites via operational abstraction. In *Proceedings of the 25th international conference on Software engineering*, IEEE Computer Society, pp. 60–71. 7.3
- Harman, M. et al. (2013). A comprehensive survey of trends in oracles for software testing. *University of Sheffield, Department of Computer Science, Tech Rep CS-13-01*. 5.6.2, 7.3
- Heller, M. (2007). REST and CRUD: the Impedance Mismatch. <http://www.infoworld.com/article/2640739/application-development/rest-and-crud--the-impedance-mismatch.html>, [last accessed 19-10-16]. 3.2.1.1, 3.3.2
- Henkel, J. and Diwan, A. (2003). Discovering algebraic specifications from Java classes. In *European Conference on Object-Oriented Programming*, Springer, pp. 431–456. 5.9.3
- Holmes, V. t. and Langford, J. (1976). Comprehension and recall of abstract and concrete sentences. *Journal of Verbal Learning and Verbal Behavior*, 15(5), pp. 559–566. 4.3.2
- Hopcroft, J. E., Motwani, R. and Ullman, J. D. (2006). Automata theory, languages, and computation. *International Edition*, 24. 2.2
- Isberner, M. (2015). *Foundations of active automata learning: an algorithmic perspective*. Ph.D. thesis. 7.3
- Javed, F. et al. (2004). Context-free grammar induction using genetic programming. In *Proceedings of the 42nd annual Southeast regional conference*, ACM, pp. 404–405. 7.3
- Jet Brains (2001). Extract Superclass. <https://www.jetbrains.com/idea/help/extract-superclass.html>, [last accessed 02-11-16]. 7.2.1.1
- JVM-TI (2006). JVM Tool Interface. <http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html>, [last accessed 13-04-15]. 3.2.4, 5.8.1

- Kapsner, C. and Godfrey, M. W. (2003). Toward a taxonomy of clones in source code: A case study. In *ELISA workshop*, p. 67. 7.2.1.3
- Kapsner, C. and Godfrey, M. W. (2006). “Clones Considered Harmful” Considered Harmful. In *Proc. Working Conf. Reverse Engineering (WCRE)*. 4.3
- Keum, C. et al. (2006). Generating test cases for web services using extended finite state machine. In *IFIP International Conference on Testing of Communicating Systems*, Springer, pp. 103–117. 7.1
- Kieyzun, A. et al. (2009). Automatic creation of sql injection and cross-site scripting attacks. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, IEEE, pp. 199–209. 3.3, 5.2.2.1
- Klein, U. and Namjoshi, K. S. (2011). Formalization and automated verification of RESTful behavior. In *International Conference on Computer Aided Verification*, Springer, pp. 541–556. 7.1
- Konietzke, S. (2013). From StorageRoom to Contentful. <https://www.contentful.com/blog/2013/06/23/storageroom-to-contentful/>, [last accessed 05-07-2017]. 3.4.1
- Lamela Seijas, P. (2014a). Frequency Server Tests. https://github.com/palas/freq_server_test [last accessed 12-09-17]. 1.8.1, 5.1.1
- Lamela Seijas, P. (2014b). Frequency Server Web Service. https://github.com/palas/freq_server [last accessed 19-12-16]. 1.8.1, 2.1.5, 5.1.1, 5.4, 5.5
- Lamela Seijas, P. (2014c). Java Erlang Bridge. <https://github.com/palas/jeb> [last accessed 29th June 2017]. 1.8.1, 5.1.1, 5.7.2
- Lamela Seijas, P., Li, H. and Thompson, S. (2013). Towards property-based testing of RESTful web services. In *Proceedings of the twelfth ACM SIGPLAN workshop on Erlang*, ACM, pp. 77–78. 3.1
- Lamela Seijas, P. and Thompson, S. (2014). James. <https://github.com/palas/james> [last accessed 14-09-16]. 1.8.1, 5, 5.1, 5.1.1

- Lamela Seijas, P. and Thompson, S. (2016a). Identifying and introducing interfaces and callbacks using wrangler. In *Proceedings of the 28th Symposium on the Implementation and Application of Functional Programming Languages*, ACM, p. 11. 4.1
- Lamela Seijas, P. and Thompson, S. (2016b). Parametrised Inference. <https://github.com/palas/detparaminf>, [last accessed 29-12-16]. 1.8.1, 6, 6.1.1, 6.3.4.6, 6.4
- Lamela Seijas, P., Thompson, S. and Francisco, M. (2012). D2.3 Property extraction. http://www.prowessproject.eu/wp-content/uploads/2012/10/Prowess_D2-3.pdf, [last accessed 17-01-17]. 5.1
- Lamela Seijas, P., Thompson, S. and Francisco, M. Á. (2016). Model extraction and test generation from JUnit test suites. In *Proceedings of the 11th International Workshop on Automation of Software Test*, ACM, pp. 8–14. 5.1
- Lamela Seijas, P. et al. (2014). Synapse: automatic behaviour inference and implementation comparison for Erlang. In *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang*, ACM, pp. 73–74. 4.1
- Lampropoulos, L. and Sagonas, K. (2012). Automatic WSDL-guided Test Case Generation for PropEr Testing of Web Services. *arXiv preprint arXiv:12106110*. 7.1
- Lang, K. J., Pearlmutter, B. A. and Price, R. A. (1998). Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In *International Colloquium on Grammatical Inference*, Springer, pp. 1–12. 2.3, 2.3.1
- Lastres Guerrero, R. (2012). Testing a distributed Wiki web application with QuickCheck. 7.1
- Li, H. and Thompson, S. (2006). The open source refactoring tool for Erlang. <http://www.cs.kent.ac.uk/projects/wrangler/>, [last accessed 19-10-16]. 4.2.3
- Li, H. and Thompson, S. (2009). Clone detection and removal for Erlang/OTP within a refactoring environment. In *Proceedings of the 2009 ACM SIGPLAN*

- workshop on Partial evaluation and program manipulation*, ACM, pp. 169–178.
4.2.3.2, 4.3.5
- Li, H. and Thompson, S. (2010). Refactoring support for modularity maintenance in erlang. In *Source Code Analysis and Manipulation (SCAM), 2010 10th IEEE Working Conference on*, IEEE, pp. 157–166. 7.2.1.4
- Li, H. and Thompson, S. (2011). A User-extensible Refactoring Tool for Erlang Programs. *University of Kent, Tech Rep.* 4.2.3.3
- Li, H. et al. (2008). Refactoring with Wrangler, updated. In *ACM SIGPLAN Erlang Workshop*, vol. 2008. 4.2.3
- Lo, D. et al. (2011). *Mining Software Specifications: Methodologies and Applications*. CRC Press. 5.8.2
- Lorenzoli, D., Mariani, L. and Pezzè, M. (2006). Inferring state-based behavior models. In *Proceedings of the 2006 international workshop on Dynamic systems analysis*, ACM, pp. 25–32. 7.3
- Lyu, M. R. et al. (1996). *Handbook of software reliability engineering*. IEEE computer society press CA. 1
- Marchetto, A., Tonella, P. and Ricca, F. (2008). State-based testing of Ajax web applications. In *2008 1st International Conference on Software Testing, Verification, and Validation*, IEEE, pp. 121–130. 7.3
- Mealy, G. H. (1955). A method for synthesizing sequential circuits. *Bell Labs Technical Journal*, 34(5), pp. 1045–1079. 5.4.2
- Miller, H. W. (1991). Information technology: Creation or evolution? *Journal of Systems Management*, 42(4), p. 23. 4.3.2
- Mitchell, B. S. and Mancoridis, S. (2006). On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3), pp. 193–208. 7.2.1.4
- Monden, A. et al. (2002). Software Quality Analysis by Code Clones in Industrial Legacy Software. In *METRICS '02*, Washington, DC, USA. 4.3

- Moore, E. F. (1956). Gedanken-experiments on sequential machines. *Automata Studies: Annals of Mathematics Studies Number 34*, (34), p. 129. 5.4.2
- Naur, P. and Randell, B. (1969). Software Engineering: Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany, 7th to 11th October 1968. Nato. 2
- Pacheco, C. and Ernst, M. D. (2007). Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, ACM, pp. 815–816. 7.3
- Papadakis, M. and Sagonas, K. (2011). A proper integration of types and function specifications with property-based testing. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, ACM, pp. 39–50. 2.4
- Păsăreanu, C. S. and Visser, W. (2009). A survey of new trends in symbolic execution for software testing and analysis. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(4), pp. 339–353. 7.3
- Peters, L. (2005). Change detection in XML trees: a survey. In *3rd Twente Student Conference on IT*. 7.2.1.2
- Pierce, B. C. (2002). *Types and programming languages*. MIT press. 4.3
- Pradel, M. and Gross, T. R. (2009). Automatic generation of object usage specifications from large method traces. In *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*, IEEE, pp. 371–382. 7.3
- QuickCheck documentation (2006). Quviq's QuickCheck documentation. <http://quviq.com/documentation/eqc/>, [last accessed 21-02-17]. 2.4.1
- Rapps, S. and Weyuker, E. J. (1985). Selecting software test data using data flow information. *IEEE transactions on software engineering*, (4), pp. 367–375. 5.2.2.1
- RFC 2616 (1999). Hypertext Transfer Protocol – HTTP/1.1. <https://tools.ietf.org/html/rfc2616>, [last accessed 13-04-15]. 3.3.1, 3.3.2

- RFC 5789 (2010). PATCH Method for HTTP. <https://tools.ietf.org/html/rfc5789>, [last accessed 13-04-15]. 3.3.2, 3.6.1.2
- RFC 6902 (2013). JavaScript Object Notation (JSON) Patch. <https://tools.ietf.org/html/rfc6902>, [last accessed 13-04-15]. 3.6.1.2
- RFC 7159 (2014). The JavaScript Object Notation (JSON) Data Interchange Format. <https://tools.ietf.org/html/rfc7159>, [last accessed 13-04-15]. 3.2.2, 3.5.4
- Richardson, L., Amundsen, M. and Ruby, S. (2013). *RESTful Web APIs*. O'Reilly Media, Inc. 3, 3.3.1, 3.3.2, 3.5.1
- Roy, C. K. and Cordy, J. R. (2007). A survey on software clone detection research. *Queen's School of Computing TR*, 541(115), pp. 64–68. 7.2.1.3
- Seng, O. et al. (2005). Search-based improvement of subsystem decompositions. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, ACM, pp. 1045–1051. 7.2.1.4
- Shahbaz, M., Li, K. and Groz, R. (2007). Learning and integration of parameterized components through testing. In *Testing of Software and Communicating Systems*, Springer, pp. 319–334. 7.3
- Storage Room (2010). Storage Room. <http://storageroomapp.com>, [last accessed 13-03-15]. 3.4.1
- Sutton, M., Greene, A. and Amini, P. (2007). *Fuzzing: brute force vulnerability discovery*. Pearson Education. 7.3
- Svenningsson, J. et al. (2014). An expressive semantics of mocking. In *International Conference on Fundamental Approaches to Software Engineering*, Springer, pp. 385–399. 5.2.1.2
- Taylor, R. (2013). Synapse. <https://github.com/ramsay-t/Synapse> [last accessed 14-09-16]. 4.2.4.2
- Taylor, R., Bogdanov, K. and Derrick, J. (2013). Automatic inference of erlang module behaviour. In *International Conference on Integrated Formal Methods*, Springer, pp. 253–267. 4.4

- dets (1997). dets manual. <http://www.erlang.org/doc/man/dets.html>, [last accessed 19-10-16]. 4.2.2
- eqc_fsm (2006). eqc_fsm documentation. http://quviq.com/documentation/eqc/eqc_fsm.html, [last accessed 26-10-16]. 2.4.2
- eqc_statem (2004). eqc_statem documentation. http://quviq.com/documentation/eqc/eqc_statem.html, [last accessed 26-10-16]. 2.4.1
- ets (1997). ets manual. <http://www.erlang.org/doc/man/ets.html>, [last accessed 19-10-16]. 4.2.2
- Thompson, S. and Li, H. (2013). Refactoring tools for functional languages. *Journal of Functional Programming*, 23(3). 4.2.3, 4.2.3.1
- Tsankov, P., Dashti, M. T. and Basin, D. (2013). Semi-valid input coverage for fuzz testing. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ACM, pp. 56–66. 5.2.2.1
- Tsantalis, N., Mazinanian, D. and Krishnan, G. P. (2015). Assessing the refactorability of software clones. *IEEE Transactions on Software Engineering*, 41(11), pp. 1055–1090. 4.3.2
- Vlissides, J. et al. (1995). Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49(120), p. 11. 2.6
- W3 XML (2008). W3 XML Specification. <https://www.w3.org/TR/xml/>, [last accessed 21-10-16]. 3.2.3
- Walkinshaw, N. and Bogdanov, K. (2013). Automated comparison of state-based software models in terms of their language and structure. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(2), p. 13. 7.2.2
- Walkinshaw, N., Derrick, J. and Guo, Q. (2009). Iterative refinement of reverse-engineered models by model-based testing. In *International Symposium on Formal Methods*, Springer, pp. 305–320. 1.1, 5.6.2
- Walkinshaw, N., Taylor, R. and Derrick, J. (2016). Inferring extended finite state machine models from software executions. *Empirical Software Engineering*, 21(3), pp. 811–853. 7.3

- Wiggerts, T. A. (1997). Using clustering algorithms in legacy systems modularization. In *Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on*, IEEE, pp. 33–43. 7.2.1.4
- Wyard, P. (1993). Context free grammar induction using genetic algorithms. In *Grammatical Inference: Theory, Applications and Alternatives, IEE Colloquium on*, IET, pp. P11–1. 7.3
- Xie, T. and Notkin, D. (2004). Automatic extraction of sliced object state machines for component interfaces. In *Proc. 3rd Workshop on Specification and Verification of Component-Based Systems*, pp. 39–46. 5.9.3
- Yuan, H. and Xie, T. (2005). Automatic extraction of abstract-object-state machines based on branch coverage. In *Proc. 1st International Workshop on Reverse Engineering To Requirements at WCRE 2005 (RETR 2005)*, pp. 5–11. 5.9.3
- Zhang, Y., Fu, W. and Qian, J. (2010). Automatic testing of web services in haskell platform. *Journal of Computational Information Systems*, 6(9), pp. 2859–2867. 7.1

Appendix A

Frequency server base implementation

frequency.erl

```
%% Code from  
%% Erlang Programming  
%% Francesco Cesarini and Simon Thompson  
%% O'Reilly, 2008  
%% http://oreilly.com/catalog/9780596518189/  
%% http://www.erlangprogramming.org/  
%% (c) Francesco Cesarini and Simon Thompson  
%% Modified by: Pablo Lamela on Dec 2013  
  
-module(frequency).  
-export([start/0, stop/0, allocate/0, deallocate/1]).  
-export([init/0]).  
  
%% These are the start functions used to  
%% create and initialize the server.  
  
start() ->  
    register(frequency, spawn(frequency, init, [])).
```

```
init() ->
    process_flag(trap_exit, true),
    Frequencies = {get_frequencies(), []},
    loop(Frequencies).

% Hard Coded
get_frequencies() -> [10,11].

%% The client Functions

stop()          -> call(stop).
allocate()      -> call(allocate).
deallocate(Freq) -> call({deallocate,Freq}).

%% We hide all message passing and the
%% message protocol in a functional
%% interface.
call(Message) -> frequency ! {request, self(), Message},
    receive
        {reply, Reply} -> Reply
    end.

reply(Pid, Message) -> Pid ! {reply, Message}.

loop(Frequencies) ->
    receive
        {request, Pid, allocate} ->
            {NewFrequencies, Reply} =
                allocate(sortfreqs(Frequencies), Pid),
            reply(Pid, Reply),
            loop(NewFrequencies);
```

```

    {request, Pid, {deallocate,Freq}} ->
        NewFrequencies = deallocate(Frequencies, Freq),
        reply(Pid, ok),
        loop(NewFrequencies);
    {'EXIT', Pid, _Reason} ->
        NewFrequencies = exited(Frequencies, Pid),
        loop(NewFrequencies);
    {request, Pid, stop} ->
        reply(Pid, ok)
end.

sortfreqs({Freqs, Allocated}) -> {lists:sort(Freqs), Allocated}.

allocate({[], Allocated}, _Pid) ->
    {{[], Allocated}, {error, no_frequencies}};
allocate({[Freq|Frequencies], Allocated}, Pid) ->
    link(Pid),
    {{Frequencies, [{Freq,Pid}|Allocated]}, {ok,Freq}}.

deallocate({Free, Allocated}, Freq) ->
    {value, {Freq,Pid}} = lists:keysearch(Freq,1,Allocated),
    unlink(Pid),
    NewAllocated=lists:keydelete(Freq,1, Allocated),
    {[Freq|Free], NewAllocated}.

exited({Free, Allocated}, Pid) ->
    case lists:keysearch(Pid,2,Allocated) of
        {value, {Freq,Pid}} ->
            NewAllocated = lists:keydelete(Freq,1, Allocated),
            {[Freq|Free],NewAllocated};
        false -> {Free,Allocated}
    end.
end.

```

Appendix B

Frequency server web service main parts

GenericServerImpl.java

```
/**
 * Created: 2014-05-25
 */
package eu.prowessproject.freq_server.state.impl;

import eu.prowessproject.freq_server.state.IFreqServerImpl;
import eu.prowessproject.freq_server.state.IFreqServerStateImpl;
import eu.prowessproject.freq_server.state.exceptions.AlreadyStarted;
import eu.prowessproject.freq_server.state.exceptions.NoFrequenciesAvailable;
import eu.prowessproject.freq_server.state.exceptions.NotAllocated;
import eu.prowessproject.freq_server.state.exceptions.NotRunning;

public class GenericServerImpl implements IFreqServerImpl {

    protected static final
        int INITIAL_FREQUENCIES [] = {10, 11, 12, 13, 14};

    private IFreqServerStateImpl freqServerState;
    private GenericServerImpl parent;

    protected GenericServerImpl()
    {
```

```

    }

    public GenericServerImpl(IFreqServerStateImpl freqServerState) {
        this.changeStateInternal(freqServerState);
    }

    private void setParent(GenericServerImpl parent)
    {
        this.parent = parent;
    }

    private void changeStateInternal(IFreqServerStateImpl freqServerState) {
        this.freqServerState = freqServerState;
        if (freqServerState instanceof GenericServerImpl) {
            ((GenericServerImpl) this.freqServerState).setParent(this);
        } else {
            throw new RuntimeException("States must extend " +
                GenericServerImpl.class.getSimpleName());
        }
    }

    protected void changeState(IFreqServerStateImpl freqServerState) {
        this.parent.changeStateInternal(freqServerState);
    }

    @Override
    public void start() throws AlreadyStarted {
        freqServerState.start_impl();
    }

    @Override
    public void stop() throws NotRunning {
        freqServerState.stop_impl();
    }

    @Override
    public Integer allocate() throws NoFrequenciesAvailable, NotRunning {
        return freqServerState.allocate_impl();
    }
}

```

```
@Override
public void deallocate(Integer frequency) throws NotAllocated, NotRunning {
    freqServerState.deallocate_impl(frequency);
}
}
```


StoppedServerStateImpl.java

```

/**
 * Created: 2014-05-25
 */
package eu.prowessproject.freq_server.state.impl;

import eu.prowessproject.freq_server.state.IFreqServerStateImpl;
import eu.prowessproject.freq_server.state.exceptions.AlreadyStarted;
import eu.prowessproject.freq_server.state.exceptions.NoFrequenciesAvailable;
import eu.prowessproject.freq_server.state.exceptions.NotAllocated;
import eu.prowessproject.freq_server.state.exceptions.NotRunning;

public class StoppedServerStateImpl
    extends GenericServerImpl implements IFreqServerStateImpl {

    @Override
    public void start_impl() throws AlreadyStarted {
        this.changeState(new RunningServerStateImpl());
    }

    @Override
    public void stop_impl() throws NotRunning {
        throw new NotRunning();
    }

    @Override
    public Integer allocate_impl() throws NoFrequenciesAvailable, NotRunning {
        throw new NotRunning();
    }

    @Override
    public void deallocate_impl(Integer frequency)
        throws NotAllocated, NotRunning
    {
        throw new NotRunning();
    }
}

```

RunningServerStateImpl.java

```

/**
 * Created: 2014-05-25
 */
package eu.prowessproject.freq_server.state.impl;

import java.util.HashSet;
import java.util.LinkedList;
import java.util.NoSuchElementException;
import java.util.Queue;

import eu.prowessproject.freq_server.state.IFreqServerStateImpl;
import eu.prowessproject.freq_server.state.exceptions.AlreadyStarted;
import eu.prowessproject.freq_server.state.exceptions.NoFrequenciesAvailable;
import eu.prowessproject.freq_server.state.exceptions.NotAllocated;
import eu.prowessproject.freq_server.state.exceptions.NotRunning;

public class RunningServerStateImpl
    extends GenericServerImpl implements IFreqServerStateImpl {

    private Queue<Integer> freeFrequencies = new LinkedList<Integer>();
    private HashSet<Integer> allocatedFrequencies = new HashSet<Integer>();

    RunningServerStateImpl() {
        for (int freq : INITIAL_FREQUENCIES) {
            freeFrequencies.add(freq);
        }
    }

    @Override
    public void start_impl() throws AlreadyStarted {
        throw new AlreadyStarted();
    }

    @Override
    public void stop_impl() throws NotRunning {
        this.changeState(new StoppedServerStateImpl());
    }

    @Override

```

```
public Integer allocate_impl() throws NoFrequenciesAvailable, NotRunning {
    try {
        Integer allocatedFreq = freeFrequencies.remove();
        allocatedFrequencies.add(allocatedFreq);
        return allocatedFreq;
    } catch (NoSuchElementException e) {
        throw new NoFrequenciesAvailable();
    }
}

@Override
public void deallocate_impl(Integer frequency)
    throws NotAllocated, NotRunning
{
    if (allocatedFrequencies.contains(frequency)) {
        allocatedFrequencies.remove(frequency);
        freeFrequencies.add(frequency);
    } else {
        throw new NotAllocated();
    }
}
}
```

Appendix C

JUnit tests by Interoud Innovation

FreqServerTest.java

```
/**
 * Copyright (c) 2014, Miguel Ángel Francisco Fernández
 *
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * 1. Redistributions of source code must retain the above copyright notice,
 * this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright notice,
 * this list of conditions and the following disclaimer in the documentation
 * and/or other materials provided with the distribution.
 *
 * 3. Neither the name of the copyright holder nor the names of its
 * contributors may be used to endorse or promote products derived from this
 * software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
```

```
* SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
* INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
* CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
* POSSIBILITY OF SUCH DAMAGE.
*
* Created: 2014-08-04
*/
package com.interoud.freqserver.test;

import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collection;

import javax.xml.bind.JAXB;

import junit.framework.Assert;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import com.interoud.freqserver.test.parser.FreqServerResponse;
import com.interoud.util.net.HTTPUtils;

@SuppressWarnings("restriction")
public class FreqServerTest {

    private static final String BASEURL = "http://localhost:8080/freq_server/";
    private static final String OK_RESPONSE = "OK";
    private static final String ERROR_RESPONSE = "ERROR";
    private static final String ERROR_TYPE_ALREADY_STARTED = "ALREADY_STARTED";
    private static final String ERROR_TYPE_NOT_RUNNING = "NOT_RUNNING";
    private static final String ERROR_TYPE_NOT_ALLOCATED = "NOT_ALLOCATED";

    /*
     * List of allocated frequencies
     */
    private Collection<Integer> allocatedFrequencies;
```

```

@Before
public void setUp() throws IOException {
    allocatedFrequencies = new ArrayList<Integer>();
}

@After
public void tearDown() throws IOException {
    /*
     * Deallocate all frequencies and stop the server
     */
    Collection<Integer> frequencies =
        new ArrayList<Integer>(allocatedFrequencies);
    for(Integer frequency : frequencies) {
        deallocateFrequency(frequency);
    }

    stopServer();
}

/* =====
 * Tests
 * =====*/
@Test
public void testStart() throws IOException {
    FreqServerResponse startServerResponse = startServer();
    checkNoErrors(startServerResponse);
}

@Test
public void testStop() throws IOException {
    startServer();
    FreqServerResponse stopServerResponse = stopServer();
    checkNoErrors(stopServerResponse);
}

@Test
public void testStartTwice() throws IOException {
    startServer();
    FreqServerResponse startServerResponse = startServer();
    checkNotAlreadyStartedError(startServerResponse);
}

```

```
@Test
public void testStopTwice() throws IOException {
    startServer();
    stopServer();
    FreqServerResponse stopServerResponse = stopServer();
    checkNotRunningError(stopServerResponse);
}

@Test
public void testAllocate() throws IOException {
    startServer();
    FreqServerResponse allocateFrequencyResponse = allocateFrequency();
    checkNoErrors(allocateFrequencyResponse);
    Assert.assertNotNull(
        allocateFrequencyResponse.getResult().getFrequencyAllocated());
}

@Test
public void testAllocateTwice() throws IOException {
    startServer();
    FreqServerResponse allocateFrequencyResponse1 = allocateFrequency();
    checkNoErrors(allocateFrequencyResponse1);
    Assert.assertNotNull(
        allocateFrequencyResponse1.getResult().getFrequencyAllocated());

    FreqServerResponse allocateFrequencyResponse2 = allocateFrequency();
    checkNoErrors(allocateFrequencyResponse2);
    Assert.assertNotNull(
        allocateFrequencyResponse2.getResult().getFrequencyAllocated());

    Assert.assertTrue(
        !allocateFrequencyResponse1.getResult().getFrequencyAllocated()
            .equals(allocateFrequencyResponse2.getResult()
                .getFrequencyAllocated()));
}

@Test
public void testAllocateNotStarted() throws IOException {
    FreqServerResponse allocateFrequencyResponse = allocateFrequency();
```

```
        checkNotRunningError(allocateFrequencyResponse);
    }

    @Test
    public void testDeallocate() throws IOException {
        startServer();
        FreqServerResponse allocateFrequencyResponse = allocateFrequency();
        Integer allocatedFrequency =
            allocateFrequencyResponse.getResult().getFrequencyAllocated();

        FreqServerResponse deallocateFrequencyResponse =
            deallocateFrequency(allocatedFrequency);
        checkNoErrors(deallocateFrequencyResponse);
    }

    @Test
    public void testDeallocateTwice() throws IOException {
        startServer();
        FreqServerResponse allocateFrequencyResponse = allocateFrequency();
        Integer allocatedFrequency =
            allocateFrequencyResponse.getResult().getFrequencyAllocated();

        deallocateFrequency(allocatedFrequency);

        FreqServerResponse deallocateFrequencyResponse =
            deallocateFrequency(allocatedFrequency);
        checkNotAllocatedError(deallocateFrequencyResponse);
    }

    @Test
    public void testDeallocateNotExistingFrequency() throws IOException {
        startServer();

        FreqServerResponse deallocateFrequencyResponse =
            deallocateFrequency(new Integer(0));
        checkNotAllocatedError(deallocateFrequencyResponse);
    }

    @Test
    public void testDeallocateNotStarted() throws IOException {
        FreqServerResponse deallocateFrequencyResponse =
```



```

        deallocateFrequency(new Integer(0));
        checkNotRunningError(deallocateFrequencyResponse);
    }

    /* =====
     * Check responses
     * =====*/
    private void checkNoErrors(FreqServerResponse response) {
        Assert.assertEquals(OK_RESPONSE, response.getState());
        Assert.assertTrue(response.getError().isEmpty());
    }

    private void checkNotAlreadyStartedError(FreqServerResponse response) {
        Assert.assertEquals(ERROR_RESPONSE, response.getState());
        Assert.assertEquals(1, response.getError().size());
        Assert.assertEquals(ERROR_TYPE_ALREADY_STARTED,
            response.getError().get(0).getErrorType());
    }

    private void checkNotRunningError(FreqServerResponse response) {
        Assert.assertEquals(ERROR_RESPONSE, response.getState());
        Assert.assertEquals(1, response.getError().size());
        Assert.assertEquals(ERROR_TYPE_NOT_RUNNING,
            response.getError().get(0).getErrorType());
    }

    private void checkNotAllocatedError(FreqServerResponse response) {
        Assert.assertEquals(ERROR_RESPONSE, response.getState());
        Assert.assertEquals(1, response.getError().size());
        Assert.assertEquals(ERROR_TYPE_NOT_ALLOCATED,
            response.getError().get(0).getErrorType());
    }

    /* =====
     * API operations
     * =====*/
    private FreqServerResponse startServer() throws IOException {
        return httpPost(BASEURL + "StartServer");
    }

    private FreqServerResponse stopServer() throws IOException {

```

```

        return httpPost(BASEURL + "StopServer");
    }

    private FreqServerResponse allocateFrequency() throws IOException {
        FreqServerResponse response = httpPost(BASEURL + "AllocateFrequency");
        if(response.getResult() != null &&
            response.getResult().getFrequencyAllocated() != null) {
            allocatedFrequencies.add(
                response.getResult().getFrequencyAllocated());
        }
        return response;
    }

    private FreqServerResponse deallocateFrequency(Integer frequency)
        throws IOException
    {
        String body = null;
        if(frequency != null) {
            body = frequency.toString();
        }
        FreqServerResponse response =
            httpPost(BASEURL + "DeallocateFrequency", body);
        if(OK_RESPONSE.equals(response.getState())) {
            allocatedFrequencies.remove(frequency);
        }
        return response;
    }

    /* =====
     * Utilities
     * =====*/
    private FreqServerResponse httpPost(String url) throws IOException {
        return httpPost(url, null);
    }

    private FreqServerResponse httpPost(String url, String body)
        throws IOException
    {
        String result = HTTPUtils.doPost(url, body, new Integer(5000),
            new Integer(5000));
        return JAXB.unmarshal(new ByteArrayInputStream(result.getBytes()),

```

```
        FreqServerResponse.class);  
    }  
}
```

Appendix D

James eqc_fsm for Freq Server

In this appendix, we provide part of the code of an `eqc_fsm` generated by James for the Frequency server, from the tests provided by Interoud Innovation (see Appendix C).

The `eqc_fsm` model is composed of:

- Two modules that are provided as part of James and are generic (they provide a patch that modifies the model so that it prints the tests generated): `iface.erl` and `utils.erl`
- Four modules that were generated by James:
 - `iface_eqc.erl` – Contains the callbacks for the `eqc_fsm` model, it describes the FSM embedded in the control flow of the model.
 - `iface_check.erl` – Obtains the identifiers for the nodes that have the postconditions for a given node.
 - `iface_dep.erl` – Resolves the data flow for a node, by creating a nested symbolic call structure.
 - `iface_used_dep.erl` – Resolves data flow for postconditions. It solves data flow up to a node in the control flow.
- A diagram (Figure 54 on page 259) that is linked to the code generated (it provides a map to understand the numbering of the states in the code).

iface.erl (template provided)

```

%%%-----
%%% @author Pablo Lamela <P.Lamela-Seijas@kent.ac.uk>
%%% @doc
%%% Example of target interface for eqc fsm
%%% @end
%%% Created : 13 Nov 2014 by Pablo Lamela Seijas
%%%-----

-module(iface).
-export([callback/3, actual_callback/5, evaluate/2]).

-include_lib("eqc/include/eqc.hrl").

% Callback dummy iface
callback({SymState, RawState}, Code, P) ->
  {SymSuperState, SymSubState} =
    case SymState of
      empty -> {1, utils:initial_state_sym()};
      {N, M} -> {N, M}
    end,
  {call, iface, evaluate,
   [Code|?LET(Params, P,
    begin
      UpSymState = utils:update_symsubstate(Params, SymSubState),
      ?LET({CheckParams, UpSymState2},
        ?SIZED(Size, utils:add_checks(Size, Code, UpSymState)),
        [{{SymSuperState + 1, UpSymState2}, RawState,
         [Params|CheckParams]})]
      end)}}}.

evaluate(_, {_SymState, RawState, [Params|Checks]}) ->
  {RawSuperState, RawSubState} =
    case RawState of
      empty -> io:format("~n"),
        {1, utils:initial_state_raw()};
      {N, M} -> {N, M}
    end,
  [_Result, NewRawSubState|_Inter] =
    lists:reverse(eqc_symbolic:eval(
      utils:serialise_trace_with_state(RawSubState, Params))),

```

```

{FinalRawSubState, _} =
  lists:foldr(fun evaluate_checks/2, {NewRawSubState, 1}, Checks),
case Checks of
  [] -> ok;
  _ -> io:format("// End of postconditions~n")
end,
{RawSuperState + 1, FinalRawSubState}.

evaluate_checks(Param, {RawState, N}) ->
io:format("// Postcondition: ~p~n", [N]),
[_Result, NewRawSubState|_Inter] =
  lists:reverse(eqcsymbolic:eval(
    utils:serialise_trace_with_state(RawState, Param))),
{NewRawSubState, N + 1}.

actual_callback(State, Code, WhatToReturn, Info, []) ->
{NewState, Result} = actual_callback(State, Code, Info, []),
case WhatToReturn of
  return -> {NewState, Result}
end;

actual_callback(State, Code, WhatToReturn, Info, {This, Params}) ->
{NewState, Result} = actual_callback(State, Code, Info, {This, Params}),
case WhatToReturn of
  return -> {NewState, Result};
  this -> {NewState, This};
  {param, N} -> {NewState, lists:nth(N, Params)}
end.

actual_callback(State, Code, #{obj_info := #{}},
  type := Type,
  value := Value}, []) ->
Num = utils:get_num_var_raw(State),
Result = case Type of
  null -> {jvar, Num, is_null};
  _ -> {jvar, Num}
end,
io:format("~s ~s = ~p;~n", [type_to_java(Type),
  name_for({Result, no_cast}), Value]),
{utils:add_all_params_to_state_raw(Code, State, [Result]), Result};
actual_callback(State, Code, #{class_signature := ClassSignature,
  method_name := "<init>"},

```

```

                                method_signature := Signature},
                                {static, ParamList}) ->
Result = {jvar, utils:get_num_var_raw(State)},
io:format("~s ~s = new ~s(~s);~n",
          [class_to_normal_notation(ClassSignature),
           name_for({Result, no_cast}),
           class_to_normal_notation(ClassSignature),
           mk_param_list(ParamList,Signature)]),
{utils:add_all_params_to_state_raw(Code, State,
                                   [Result, static | ParamList]), Result};
actual_callback(State, Code, #{class_signature := ClassSignature,
                              method_name := Name,
                              method_signature := Signature},
               {static, ParamList}) ->
Result = {jvar, utils:get_num_var_raw(State)},
Ret = return_from_sig(Signature),
case Ret of
  "void" -> io:format("~s.~s(~s);~n",
                    [class_to_normal_notation(ClassSignature),
                     Name, mk_param_list(ParamList,Signature)]);
  _ -> io:format("~s ~s = ~s.~s(~s);~n",
                [Ret,
                 name_for({Result, no_cast}),
                 class_to_normal_notation(ClassSignature),
                 Name, mk_param_list(ParamList,Signature)])
end,
{utils:add_all_params_to_state_raw(Code, State,
                                   [Result, static | ParamList]), Result};
actual_callback(State, Code, #{method_name := Name,
                              method_signature := Signature},
               {This, ParamList}) ->
Result = {jvar, utils:get_num_var_raw(State)},
Ret = return_from_sig(Signature),
case Ret of
  "void" -> io:format("~s.~s(~s);~n",
                    [name_for({This, no_cast}), Name,
                     mk_param_list(ParamList,Signature)]);
  _ -> io:format("~s ~s = ~s.~s(~s);~n",
                [Ret, name_for({Result, no_cast}),
                 name_for({This, no_cast}), Name,
                 mk_param_list(ParamList,Signature)])
end,
{utils:add_all_params_to_state_raw(Code, State,
                                   [Result, static | ParamList]), Result};

```

```

end,
{utils:add_all_params_to_state_raw(Code, State,
                                   [Result, This | ParamList]), Result};
actual_callback(State, Code, Rec, ParamList) ->
  io:format("~nState:~p~nCode:~p~nRec:~p~nParamList:~p~n",
            [State, Code, Rec, ParamList]),
  Result = {jvar, utils:get_num_var_raw(State)},
  io:format("Result: ~p~n~n", [Result]),
  {utils:add_all_params_to_state_raw(Code, State, [Result | ParamList]),
   Result}.

mk_param_list(ParamList,Signature) ->
  list_to_commasep_str(
    lists:map(fun name_for/1,
              lists:zip(ParamList,
                        get_param_types(Signature)))).

get_param_types([$(|List)] -> get_param_types(List);
get_param_types([$|_]) -> [];
get_param_types(List) ->
  case get_param_types_aux(List) of
    {Result, RemList} -> [Result|get_param_types(RemList)]
  end.

get_param_types_aux([$L|Rest]) ->
  {Class, [_|Remaining]} = lists:splitwith(difffrom($;), Rest),
  {class_to_normal_notation([$L|Class] ++ ";"), Remaining};
get_param_types_aux([$|Rest]) ->
  {_, Remaining} = get_param_types_aux(Rest),
  {null, Remaining}; % We return null because arrays are not implemented
get_param_types_aux([Char|Rest]) ->
  {class_to_normal_notation([Char]), Rest}.

return_from_sig(String) ->
  class_to_normal_notation(tl(lists:dropwhile(difffrom($), String))).

difffrom(Char) -> fun (T) -> Char /= T end.

type_to_java(integer) -> "int";
type_to_java(float) -> "float";

```



```

type_to_java(double) -> "double";
type_to_java(null) -> "Object";
type_to_java(string) -> "String";
type_to_java(stringBuffer) -> "java.lang.StringBuffer";
type_to_java(stringBuilder) -> "java.lang.StringBuilder";
type_to_java(class) -> "Class<?>";
type_to_java(boolean) -> "boolean";
type_to_java(char) -> "char";
type_to_java(short) -> "short";
type_to_java(long) -> "long".

name_for({this, _}) -> "this";
name_for({{jvar, Num, is_null}, Cast}) when Cast != no_cast ->
  "(" ++ Cast ++ " var" ++ integer_to_list(Num);
name_for({{jvar, Num}, _}) -> "var" ++ integer_to_list(Num);
name_for({{jvar, Num, is_null}, _}) -> "var" ++ integer_to_list(Num).

class_to_normal_notation([]) -> [];
class_to_normal_notation([$B]) -> "byte";
class_to_normal_notation([$C]) -> "char";
class_to_normal_notation([$D]) -> "double";
class_to_normal_notation([$F]) -> "float";
class_to_normal_notation([$I]) -> "int";
class_to_normal_notation([$J]) -> "long";
class_to_normal_notation([$S]) -> "short";
class_to_normal_notation([$Z]) -> "boolean";
class_to_normal_notation([$V]) -> "void";
class_to_normal_notation([$L|Rest]) -> class_to_normal_notation_aux(Rest).
class_to_normal_notation_aux(";") -> [];
class_to_normal_notation_aux([$|Rest]) ->
  [$|class_to_normal_notation_aux(Rest)];
class_to_normal_notation_aux([Char|Rest]) ->
  [Char|class_to_normal_notation_aux(Rest)].

list_to_commas_sep_str(L) -> lists:flatten(list_to_commas_sep_str_aux(L)).
list_to_commas_sep_str_aux([]) -> "";
list_to_commas_sep_str_aux([H|[]]) -> [io_lib:format("~s", [H])];
list_to_commas_sep_str_aux([H|T]) ->
  [io_lib:format("~s, ", [H])|list_to_commas_sep_str_aux(T)].

```

utils.erl (template provided)

```

%%%-----
%%% @author Pablo Lamela <P.Lamela-Seijas@kent.ac.uk>
%%% @doc
%%% Utils needed by the eqc suites generated
%%% @end
%%% Created : 11 Nov 2014 by Pablo Lamela Seijas
%%%-----

-module(utils).

-include_lib("eqc/include/eqc.hrl").

-export([serialise_trace_with_state/2, update_symsubstate/2,
        initial_state_sym/0, add_result_to_state_sym/2,
        get_instances_of_sym/4, get_num_var_sym/1,
        initial_state_raw/0, add_all_params_to_state_raw/3,
        get_instance_of_raw/3, get_instance_of_raw_aux/3,
        get_num_var_raw/1, add_checks/3, control_add/3,
        used_and_res/1, used_and_fix/2, used_or/1, remove_result_tag/1,
        add_weights/3, normalise_weights/1, set_weights/2]).

% Symbolic state accessors
initial_state_sym() -> {1, dict:new()}.
add_result_to_state_sym(Code, {N, Dict}) ->
    {N + 1, dict:update(Code, fun (Old) -> [N|Old] end, [N], Dict)}.
get_instances_of_sym(Code, WhatToReturn, {_N, Dict}, _RawState) ->
    case dict:find(Code, Dict) of
        {ok, List} -> [{jcall, ?MODULE, get_instance_of_raw_aux,
                        [WhatToReturn, Entry]} || Entry <- List];
        error -> []
    end.
get_last_instance_num_of_sym(Code, {_N, Dict}) ->
    case dict:find(Code, Dict) of
        {ok, List} -> lists:max(List);
        error -> 0
    end.
get_num_var_sym({_N, _}) -> N.
% Raw state accessors
initial_state_raw() -> {1, dict:new()}.

```

```

add_all_params_to_state_raw(_Code, {N, Dict}, Result) ->
  {N + 1, dict:store(N, Result, Dict)}.
get_instance_of_raw(Code, WhatToReturn, {_N, Dict}) ->
  lists:nth(case WhatToReturn of
            return -> 1;
            this -> 2;
            {param, N} -> 2 + N
            end, dict:fetch(Code, Dict)).
get_instance_of_raw_aux(RawState, WhatToReturn, Code) ->
  {RawState, get_instance_of_raw(Code, WhatToReturn, RawState)}.
get_num_var_raw({N, _}) -> N.

control_add(State, WhatToReturn, Code) ->
  case utils:get_instances_of_sym(Code, WhatToReturn, State, dummy) of
  [] -> error;
  List -> {ok, oneof(List)}
  end.

used_and_fix(_Def, error) -> error;
used_and_fix(Def, _Res) -> {ok, Def}.
used_and_res({This, List}) ->
  case used_and_res(place_static(This, List)) of
  error -> error;
  Else -> replace_static(This, Else)
  end;
used_and_res(List) when is_list(List) ->
  try used_and_aux(List) of
  Res -> Res
  catch
  has_error -> error
  end.

place_static(static, List) -> List;
place_static(Else, List) -> [Else|List].
replace_static(static, List) -> {static, List};
replace_static(_Else, [This|List]) -> {This, List}.

used_and_aux([error|_] -> throw(has_error);
used_and_aux([{ok, E1}|Rest]) -> [E1|used_and_aux(Rest)];
used_and_aux([]) -> [].

used_or(List) ->

```

```

    case remove_result_tag(List) of
      [] -> error;
      Else -> {ok, frequency(Else)}
    end.

remove_result_tag([{ok, Sth}|Rest]) -> [Sth|remove_result_tag(Rest)];
remove_result_tag([error|Rest]) -> remove_result_tag(Rest);
remove_result_tag([]) -> [].

add_weights(_State, _Node, error) -> error;
add_weights(State, Node, {ok, Val}) ->
  Weight = get_last_instance_num_of_sym(Node, State),
  {ok, {Weight, Val}}.

normalise_weights(List) ->
  SrtList = lists:sort(fun weight_sort_criteria/2, List),
  {Ok, Errors} = lists:splitwith(fun is_ok/1, SrtList),
  {NotZero, Zero} = lists:splitwith(fun is_not_zero/1, Ok),
  NormalisedNotZero = normalise_not_zero(NotZero),
  BiasedZero = bias_zero(Zero),
  NormalisedNotZero ++ BiasedZero ++ Errors.

weight_sort_criteria(error, _) -> false;
weight_sort_criteria(_, error) -> true;
weight_sort_criteria({ok, {0, _}}, _) -> false;
weight_sort_criteria(_, {ok, {0, _}}) -> true;
weight_sort_criteria({ok, {N, _}}, {ok, {M, _}}) when N =< M -> false;
weight_sort_criteria(_, _) -> true.

is_ok({ok, _}) -> true;
is_ok(_) -> false.

is_not_zero({ok, {0, _}}) -> false;
is_not_zero(_) -> true.

normalise_not_zero(List) ->
  Elems = length(List),
  ZippedList = lists:zip(lists:reverse(lists:seq(1, Elems)), List),
  [{ok, {((Pos * 10) div Elems) + 30, Val}} || {Pos, {ok, {_, Val}}}
   <- ZippedList].

```

```

bias_zero(List) ->
  [{ok, {10, Val}} || {ok, {0, Val}} <- List].

add_checks(Size, Code, SymSubState) ->
  ?LET(Checks,
    remove_result_tag(
      [iface_used_dep:used_args_for(Size, SymSubState, return, Check)
      || Check <- iface_check:checks_for(Code), Check /= Code]),
    begin
      NewSymSubState = lists:foldr(fun update_symsubstate/2,
                                   SymSubState, Checks),
      {Checks, NewSymSubState}
    end).

serialise_trace_with_state(State, Trace) ->
  {{STrace, _}, {_, _}} = serialise_trace_with_state_aux(Trace, {1, State}),
  STTrace.

serialise_trace_with_state_aux({jcall, Mod, Fun, Args}, {AccIn, State}) ->
  {ArgsRes, {InnerAcc, PreState}} =
    lists:mapfoldl(fun serialise_trace_with_state_aux/2,
                  {AccIn, State}, Args),
  {ReqArgs, SymArgs} = lists:unzip(ArgsRes),
  IA = fun (X) -> InnerAcc + X end,
  IAV = fun (X) -> {var, IA(X)} end,
  {{lists:concat(ReqArgs)
    ++ [{set, IAV(0), {call, Mod, Fun, [PreState|SymArgs]}},
        {set, IAV(1), {call, erlang, element, [1, IAV(0)]}},
        {set, IAV(2), {call, erlang, element, [2, IAV(0)]}}],
    IAV(2)},
   {IA(3), IAV(1)}};

serialise_trace_with_state_aux(Else, Acc) when is_tuple(Else) ->
  {{ReqRes, SymRes}, NewAcc} =
    serialise_trace_with_state_aux(tuple_to_list(Else), Acc),
  {{ReqRes, list_to_tuple(SymRes)}, NewAcc};

serialise_trace_with_state_aux(Else, Acc) when is_list(Else) ->
  {ElsRes, NewAcc} =
    lists:mapfoldl(fun serialise_trace_with_state_aux/2, Acc, Else),
  {ReqEls, SymEls} = lists:unzip(ElsRes),
  {{lists:concat(ReqEls), SymEls}, NewAcc};

```

```
serialise_trace_with_state_aux(Else, Acc) -> {[[]], Else}, Acc}.

update_symsubstate({jcall, _Mod, actual_callback, Args}, State) ->
  PreState = lists:foldl(fun update_symsubstate/2, State, Args),
  utils:add_result_to_state_sym(hd(Args), PreState);
update_symsubstate(Else, Acc) when is_tuple(Else) ->
  update_symsubstate(tuple_to_list(Else), Acc);
update_symsubstate(Else, Acc) when is_list(Else) ->
  lists:foldl(fun update_symsubstate/2, Acc, Else);
update_symsubstate(_Else, Acc) -> Acc.

set_weights(N, List) -> [{N, E1} || E1 <- List].
```

iface_eqc.erl (generated by James)

```

-module(iface_eqc).

-include_lib("eqc/include/eqc.hrl").

-include_lib("eqc/include/eqc_fsm.hrl").

-compile(export_all).

initial_state() -> state_init.

state_init(State) ->
  [{state_33,
    callback(State, "33",
              ?SIZED(Size,
                    (iface_dep:args_for(Size div 10, return, State,
                                         "33"))))}],
   {state_240,
    callback(State, "240",
              ?SIZED(Size,
                    (iface_dep:args_for(Size div 10, return, State,
                                         "240"))))}],
   {state_172,
    callback(State, "172",
              ?SIZED(Size,
                    (iface_dep:args_for(Size div 10, return, State,
                                         "172"))))}].

state_203(State) ->
  [{state_240,
    callback(State, "240",
              ?SIZED(Size,
                    (iface_dep:args_for(Size div 10, return, State,
                                         "240"))))}].

state_172(_) -> [].

state_342(State) ->
  [{state_38,
    callback(State, "38",

```

```

        ?SIZED(Size,
            (iface_dep:args_for(Size div 10, return, State,
                "38")))))]].

state_10(_) -> [].

state_240(_) -> [].

state_38(_) -> [].

state_33(State) ->
    [{state_342,
        callback(State, "342",
            ?SIZED(Size,
                (iface_dep:args_for(Size div 10, return, State,
                    "342"))))},
        {state_240,
            callback(State, "240",
                ?SIZED(Size,
                    (iface_dep:args_for(Size div 10, return, State,
                        "240"))))},
        {state_196,
            callback(State, "196",
                ?SIZED(Size,
                    (iface_dep:args_for(Size div 10, return, State,
                        "196"))))},
        {state_10,
            callback(State, "10",
                ?SIZED(Size,
                    (iface_dep:args_for(Size div 10, return, State,
                        "10")))))]].

state_196(State) ->
    [{state_203,
        callback(State, "203",
            ?SIZED(Size,
                (iface_dep:args_for(Size div 10, return, State,
                    "203"))))},
        {state_196,
            callback(State, "196",
                ?SIZED(Size,

```



```

        (iface_dep:args_for(Size div 10, return, State,
                            "196")))]}.

precondition(_From, _To, _S, _) -> true.

initial_state_data() -> {empty, empty}.

next_state_data(_From, _To, _S, RawState,
                {call, iface, evaluate,
                 [_, {SymState, _OldRawState, _Params}]}) ->
    {SymState, RawState}.

postcondition(_From, state_error, _S, _Call, R) ->
    case R of
        'EXIT', _} -> true;
        _ -> false
    end;
postcondition(_From, _To, _S,
              {call, iface, evaluate,
               [_, {_SymState, _OldRawState, _Params}]},
              R) ->
    case R of
        'EXIT', _} -> false;
        _ -> true
    end.

prop_iface() ->
    ?FORALL(Cmds, (commands(?MODULE)),
            begin
                {_History, S, Res} = run_commands(?MODULE, Cmds),
                cleanup(S),
                Res == ok
            end).

callback(X1, X2, X3) ->
    catch iface:callback(X1, X2, X3).

cleanup(_S) -> none.

```

iface_check.erl (generated by James)

```
-module(iface_check).

-include_lib("eqc/include/eqc.hrl").

-compile(export_all).

checks_for("122") ->
    checks_for("124") ++ checks_for("134");
checks_for("203") -> checks_for("207");
checks_for("41") -> ["41"];
checks_for("320") -> ["320"];
checks_for("172") -> checks_for("174");
checks_for("66") -> ["66"];
checks_for("345") -> ["345"];
checks_for("13") -> ["13"];
checks_for("207") -> ["207"];
checks_for("342") -> checks_for("345");
checks_for("10") -> checks_for("13");
checks_for("240") ->
    checks_for("298") ++ checks_for("244");
checks_for("120") -> checks_for("122");
checks_for("38") -> checks_for("41");
checks_for("134") -> ["134"];
checks_for("244") -> ["244"];
checks_for("124") -> ["124"];
checks_for("174") -> ["174"];
checks_for("33") -> checks_for("320");
checks_for("298") -> ["298"];
checks_for("196") ->
    checks_for("120") ++ checks_for("66").
```

iface_dep.erl (generated by James)

Omitted parts marked with "...".

```

-module(iface_dep).

-include_lib("eqc/include/eqc.hrl").

-compile(export_all).

args_for_op(Size, WhatToReturn, {empty, empty} = State,
            Code) ->
    ?LAZY((args_for(Size, WhatToReturn, State, Code)));
args_for_op(Size, WhatToReturn,
            {_, SymState}, RawState} = State, Code) ->
    case utils:get_instances_of_sym(Code, WhatToReturn,
                                    SymState, RawState)
    of
    [] -> args_for(Size, WhatToReturn, State, Code);
    List ->
        ?LAZY((frequency(utils:set_weights(3, List) ++
                        utils:set_weights(1,
                                          [args_for(Size, WhatToReturn,
                                                    State, Code)]))))
    end.

args_for(_Size, WhatToReturn, _State, "266") ->
    {jcall, iface, actual_callback,
     ["266", WhatToReturn,
      #{obj_info => #{}, type => integer, value => 0}, []]};
args_for(_Size, _WhatToReturn, _State,
         "diamond329o185") ->
    ?LAZY((oneof([])));
args_for(Size, _WhatToReturn, State,
         "diamond240o297") ->
    ?LAZY((oneof([args_for_op(Size, return, State, "293")]
                ++
                [args_for_op(Size - 1, return, State, "201")
                 || Size > 0])));
args_for(_Size, _WhatToReturn, _State,
         "diamond147o85") ->

```

```

?LAZY((oneof([])));

. . .

args_for(Size, WhatToReturn, State, "41") ->
{jcall, iface, actual_callback,
 ["41", WhatToReturn,
  #{class_signature =>
    "Lcom/interoud/freqserver/test/FreqServerTest;",
    http_request => no, is_dynamic => true,
    method_name => "checkNotRunningError",
    method_signature =>
      "(Lcom/interoud/freqserver/test/parser/FreqSer"
      "verResponse;)V",
    params => [object], return => void, this => object},
  {args_for_op(Size, return, State, "167"),
   [args_for_op(Size, return, State, "38")]]}};

. . .

args_for(Size, WhatToReturn, State, "196") ->
{jcall, iface, actual_callback,
 ["196", WhatToReturn,
  #{class_signature =>
    "Lcom/interoud/freqserver/test/FreqServerTest;",
    http_request =>
      {post, "/freq_server/AllocateFrequency"},
    is_dynamic => true, method_name => "allocateFrequency",
    method_signature =>
      "()Lcom/interoud/freqserver/test/parser/FreqSe"
      "rverResponse;",
    params => [], return => object, this => object},
  {args_for_op(Size, return, State, "167"), []}}}.

```

iface_used_dep.erl (generated by James)

Omitted parts marked with "...".

```
-module(iface_used_dep).

-compile(export_all).

used_args_for(_Size, State, WhatToReturn, "203") ->
    utils:control_add(State, WhatToReturn, "203");
used_args_for(_Size, State, WhatToReturn, "172") ->
    utils:control_add(State, WhatToReturn, "172");
used_args_for(_Size, State, WhatToReturn, "147") ->
    utils:control_add(State, WhatToReturn, "147");
used_args_for(_Size, State, WhatToReturn, "10") ->
    utils:control_add(State, WhatToReturn, "10");
used_args_for(_Size, State, WhatToReturn, "342") ->
    utils:control_add(State, WhatToReturn, "342");
used_args_for(_Size, State, WhatToReturn, "240") ->
    utils:control_add(State, WhatToReturn, "240");
used_args_for(_Size, State, WhatToReturn, "38") ->
    utils:control_add(State, WhatToReturn, "38");
used_args_for(_Size, State, WhatToReturn, "329") ->
    utils:control_add(State, WhatToReturn, "329");
used_args_for(_Size, State, WhatToReturn, "33") ->
    utils:control_add(State, WhatToReturn, "33");
used_args_for(_Size, State, WhatToReturn, "196") ->
    utils:control_add(State, WhatToReturn, "196");
used_args_for(_Size, _State, WhatToReturn, "266") ->
    {ok,
     {jcall, iface, actual_callback,
      ["266", WhatToReturn,
       #{obj_info => #{}, type => integer, value => 0}, []]}};
used_args_for(_Size, _State, _WhatToReturn, "diamond329o185") -> error;
used_args_for(Size, State, _WhatToReturn, "diamond240o297") ->
    utils:used_or(
        utils:normalise_weights(
            [utils:add_weights(State,
                               Node,
                               used_args_for(NewSize,
                                             State,
```

```

        WhatToReturn,
        Node))
    || {NewSize, WhatToReturn, Node} <- [{Size, return, "293"}]
        ++ [{Size - 1, return, "201"} || Size > 0]);
used_args_for(_Size, _State, _WhatToReturn, "diamond147o85") -> error;
used_args_for(_Size, _State, _WhatToReturn, "diamond240o242") -> error;
used_args_for(_Size, _State, _WhatToReturn, "diamond196o65") -> error;

. . .

used_args_for(Size, State, WhatToReturn, "41") ->
  case utils:control_add(State, WhatToReturn, "41") of
  error ->
    Params = {used_args_for(Size, State, return, "167"),
              [used_args_for(Size, State, return, "38")]},
    CParams = utils:used_and_res(Params),
    utils:used_and_fix(
      {jcall, iface, actual_callback,
       ["41", WhatToReturn,
        #{class_signature =>
          "Lcom/interoud/freqserver/test/FreqServerTest;",
          http_request => no, is_dynamic => true,
          method_name => "checkNotRunningError",
          method_signature =>
            "(Lcom/interoud/freqserver/test/parser/"
            "FreqServerResponse;)V",
          params => [object], return => void,
          this => object},
          CParams]}},
      CParams);
  Else -> Else
end;

. . .

used_args_for(Size, State, WhatToReturn, "66") ->
  case utils:control_add(State, WhatToReturn, "66") of
  error ->
    Params = {used_args_for(Size, State, return, "167"),
              [used_args_for(Size, State, return, "196")]},
    CParams = utils:used_and_res(Params),

```

```

    utils:used_and_fix(
      {jcall, iface, actual_callback,
       ["66", WhatToReturn,
        #{class_signature =>
          "Lcom/interoud/freqserver/test/FreqServerTest;",
          http_request => no, is_dynamic => true,
          method_name => "checkNoErrors",
          method_signature =>
            "(Lcom/interoud/freqserver/test/parser/"
            "FreqServerResponse;)V",
          params => [object], return => void,
          this => object},
         CParams]}},
      CParams);
  Else -> Else
end;

. . .

used_args_for(Size, State, WhatToReturn, "199") ->
  case utils:control_add(State, WhatToReturn, "199") of
  error ->
    Params = {used_args_for(Size, State, return, "196"),
              []},
    CParams = utils:used_and_res(Params),
    utils:used_and_fix(
      {jcall, iface, actual_callback,
       ["199", WhatToReturn,
        #{class_signature =>
          "Lcom/interoud/freqserver/test/parser/"
          "FreqServerResponse;",
          http_request => no, is_dynamic => true,
          method_name => "getResult",
          method_signature =>
            "()Lcom/interoud/freqserver/test/parser/Result;",
          params => [], return => object,
          this => object},
         CParams]}},
      CParams);
  Else -> Else
end.

```

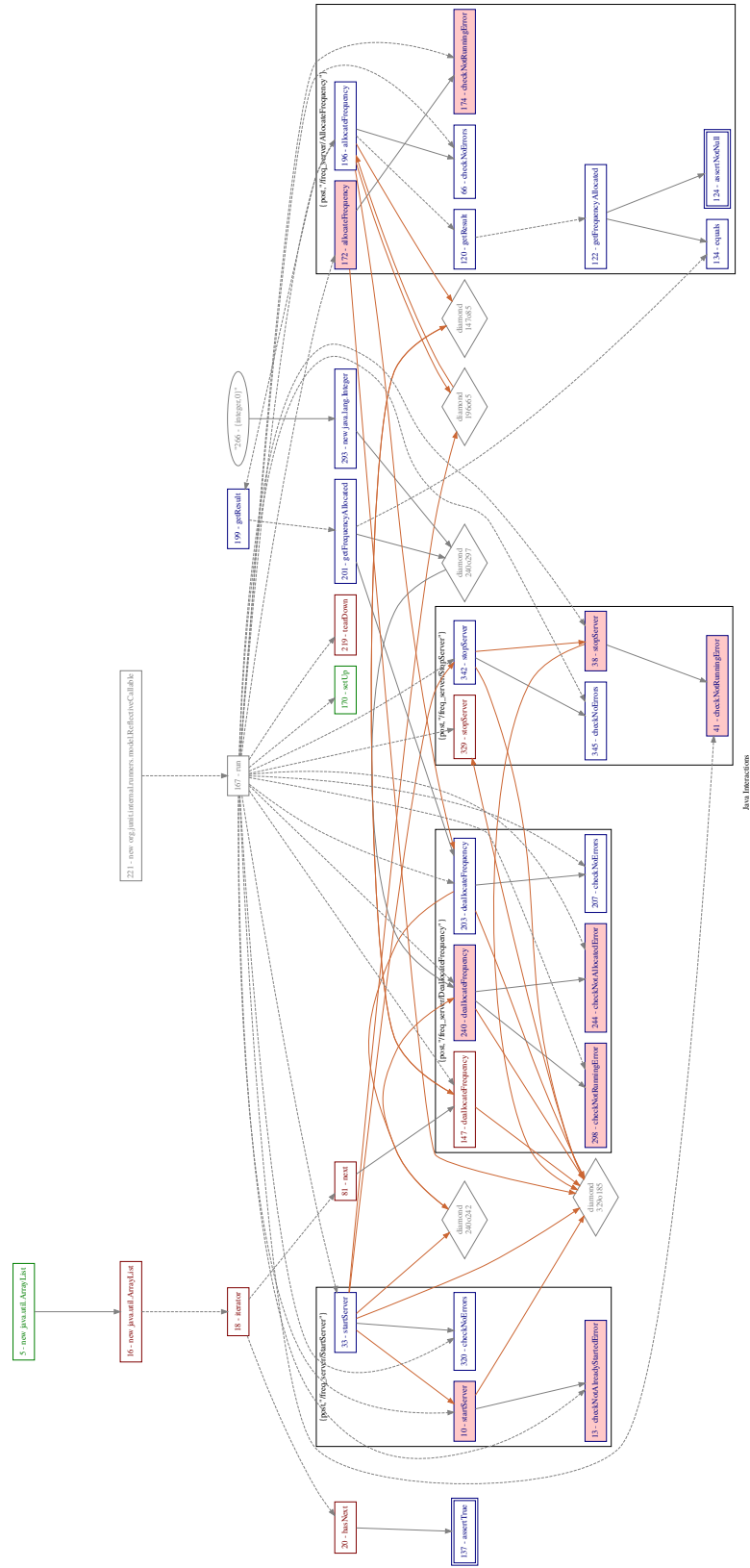


Figure 54: Linked model diagram