2016 11th IEEE/ACM International Workshop in Automation of Software Test

Model extraction and test generation from JUnit test suites

Pablo Lamela Seijas
University of Kent
United Kingdom
pl240@kent.ac.uk

Simon Thompson
University of Kent
United Kingdom
sjt@kent.ac.uk

Miguel Ángel Francisco
Interoud Innovation S.L.
Spain
miguel.francisco@interoud.com

ABSTRACT

In this paper we describe how to infer state machine models of systems from legacy unit test suites, and how to generate new tests from those models. The novelty of our approach is to combine control dependencies and data dependencies in the same model, in contrast to most other work in this area. Combining both kinds of dependency helps us to build more expressive models, which in turn allows us to produce smarter tests. We illustrate those techniques with examples from our implementation, the James tool, designed to apply these techniques in practice to Java code and tests.

1. INTRODUCTION

Testing is the most commonly used approach to validating systems, both when they are constructed and as they evolve. Testing is a costly process, and at the same time necessarily partial, exploring the system only at the points specified in the test suite. In this paper we show how existing unit tests can be leveraged to provide more testing value through inferring a model for the system from the tests. We make four specific contributions.

1. We define a new approach to inferring a state machine model for a system from an existing test suite and an implementation of the system. The state machine is inferred using a combination of data flow and control flow information: existing approaches have tended to use just one of these.
2. We show how to automatically derive potential new test cases for the system under test from this model. The new tests are generated from the model using the QuickCheck [1] property-based testing (PBT) tool, which exercises the model and prints examples of sequences of calls and postconditions.
3. We give 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.
4. 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 behaviour described by the model aims to be more general than the original unit tests. This generalisation allows us to generate new tests simply by randomly traversing the model. In doing this we follow earlier work [2] for Erlang in which finite-state machine models and properties were extracted from EUnit test suites.

However, as with any automatic generalisation, some aspects of the models generated and, as a consequence, some of the tests generated, may not correspond to the intended behaviour of the system. Tests generated may need to be manually reviewed before they are added to a test suite, and models generated may need to be manually corrected before being used in practice. Nevertheless, adapting approximate models and tests is, in general, less costly than writing them from scratch, and they may explore scenarios that humans did not consider when doing the work manually.

The techniques described here have been implemented in a tool called James. The source code of James is available[1]; and more technical details can also be found[2].

The work described here and the implementation of James are both targeted on Web Services, since they identify the interface by looking for HTTP requests (see Sect. 4.3). Thus, it is a requirement that the target system is a Web Service. However, the main ideas presented here should be straightforward to apply them to other types of API, (e.g. of a dynamic library), as long as the system under test (SUT) is tested like a black-box and has a clear interface.

The paper is structured as follows: after introducing property-based testing (Sect. 2) and related work (Sect. 3), we motivate the approach taken (Sect. 4). We then explain the architecture of our solution (Sect. 5), the model generation (Sect. 6), the test generation (Sect. 8), and the pilot study (Sect. 9). We discuss future work and conclude in Sect. 10.

2. PROPERTY-BASED TESTING

Property-based testing (PBT) was first developed for Haskell [7], and has been transferred to a range of other programming languages. Quviq QuickCheck [1] (hereafter QuickCheck) supports random testing of Erlang (and C via a foreign function interface).

Properties of programs are stated in a subset of first-order logic, embedded in Erlang syntax. QuickCheck verifies these properties for collections of Erlang data values generated randomly, with user guidance in defining the generators where necessary. When a counterexample is found, QuickCheck tries to generate a simpler, more comprehensible, counterexample in a constructive manner; this process is called *shrinking*.

When testing state-based systems it makes sense to build an abstract model of the system, and to use this model to drive the testing of the real system. The abstract state machine can be implemented as a client module of the pre-defined QuickCheck behaviour *eqc_fsm*. *eqc_fsm* state machines consist of a finite set

AST'16, May 14-15 2016, Austin, TX, USA

ISBN 978-1-4503-4151-6/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2896921.2896927

[1] https://github.com/palas/james

[2] http://www.prowessproject.eu/wp-content/uploads/2012/10/Prowess_D2-3.pdf

of *("control") states*, together with *state data* which is modified by the transitions of the machine. These models are variants of *extended finite state machines (EFSMs)*, and so more expressive than FSMs.

3. RELATED WORK

Previous approaches [13], [8], and [12], model the expected use of interfaces by focusing on the order in which commands are usually executed. One limitation of these approaches is that they do not usually infer how to create the parameters that the commands require.

In some cases invariants for parameters are inferred [11], and then used to disambiguate commands in Finite State Machines. This has limited effectiveness when inferring complex properties, or arbitrary semi-structured data.

These approaches have the advantage of being suitable for black-box interfaces. But they do not take advantage of the dependency information provided by legacy unit tests.

On the other hand, [3] shows similar data inference to the one used in this paper. However, control dependencies are inferred directly from data dependencies, since examples are not used as input for the algorithm.

In addition, the merging algorithm used in this work is strongly influenced by previous regular inference algorithms, in particular K-tails [4] and QSM [9].

We have also used QSM as the core algorithm for our previous work in test generation [2], but the work presented in this paper differs from it in that we are now combining data dependencies on the model. This addition allows it to convey aspects of the system under test that go beyond the ones learnable by the pure regular inference.

There is also been more recent work on software reverse-engineering using regular inference [5] and even EFSM inference with help of machine learning [14].

4. EXTRACTION OF DEPENDENCIES

We now give a rationale for the approach we have taken for extracting dependency information from the existing artefacts.

4.1 Taking advantage of data reuse

Existing JUnit tests provide concrete examples of how data can be reused, and how it can be generated. By modifying or generalising these procedures, it is likely that we will find new valid input examples that have not been generated before. Moreover, even if generated inputs are invalid, they may be appropriate for negative test cases. Because invalid input generated this way will be structurally similar to the valid input, it is foreseeable that it will help us detect 'corner case' errors.

For example, if when serialising a request the unit tests manually add quotes surrounding a value, then the generalisation of the request may add quotes in places were they should not occur. This kind of test case would help us detect problems like SQL injection, which can enforce security.

4.2 Approaches to instrumentation

Most research that presents techniques to extract information from existing software falls into one or both of two categories, namely *static* and *dynamic*. In this section we assess the advantages and disadvantages of these approaches for our work, and explain our preferred approach.

Static approaches An approach is considered to be *static* if it analyses the software without executing it. Static approaches, do not require the code to be run and usually work directly on source code but may analyse bytecode or even machine code.

In the particular case of source code analysis, a static approach can potentially analyse 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 already omits some of the abstraction.

Dynamic approaches Dynamic approaches analyse the way in which a working piece of software executes by instrumenting it prior to its execution. This has the advantage of acquiring real values that are known to work, and to produce complete traces of actual valid executions. In the case of unit tests, which are usually deterministic, a single execution will reveal all the scenarios that are being tested.

One disadvantage of this approach is that it requires a working implementation, which limits its applicability to test-driven development.

Java Virtual Machine Tool Interface For this work we have chosen a mainly dynamic approach through using the JVM Tool Interface, (or JVMTI[3]). JVMTI 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`

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 could also be used to alter the behaviour of the target program.

The agent acts as a debugger, and should not modify the results of the tests and work seamlessly regardless of the framework, configuration, or JVM used for executing them (assuming that there are no bugs in the implementation of James and the JVM, and that the tests do not rely on timing or other unusual kinds of context information).

4.3 Data and control dependencies

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

Extraction of data dependencies Data dependencies represent the flow of information in the tests. In our experiments we register data dependencies by tracking all the objects in the system, and registering the methods or functions that take them as parameters or produce them as a result. This way we obtain information about the way in which requests are constructed.

Most of the time, requests are composed out of small pieces of information, like numeric values or dates, which are composed into bigger structures and then serialised, or appended inside of a string template.

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

Extraction of control dependencies In addition to data flow, we track and model the control flow of methods that produce HTTP requests. Nodes that represent these methods are linked in execution order, and the links are preserved during the merging process.

James records the order in which these particular methods are executed because they are the ones that may cause the state of

[3]http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html

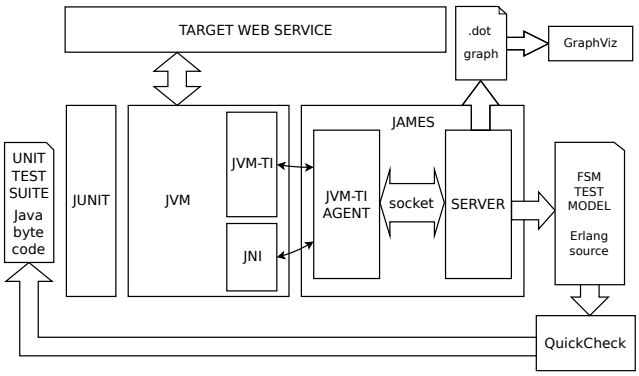


Figure 1: The architecture of James

the server to change. We explain how these methods are identified in Sect. 5.3 below.

5. ARCHITECTURE OF THE SOLUTION

In Fig. 1 we show the architecture of the tool that we have designed and implemented.

The JVM is instrumented by a JVMTI agent in C++ that instruments every method entry and method exit event, and reports it to an Erlang server through a socket. In practice, method entries correspond to method calls, and method exits allow us to track the result of the method executions.

This process produces a long list of method calls, most of which 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. We do this by checking for annotations using Java's reflection API. But we store in a cache the classes that are found to not contain JUnit tests (otherwise this introduces a big overhead).

This procedure is also used to distinguish the *set-up* and *clean-up* procedures and the actual test *body*, since they have different annotations, (i.e: `@Before`, `@After`, `@Test`).

Calls that produce objects that are used within the tests, even when these are not part of the tests themselves, must be tracked too, otherwise James will not know how to create those objects when the new tests are generated.

5.1 Technical limitations

There are some limitations to our approach. James can track objects, but not every variable in Java is an object. Some variables have primitive types, (e.g: `int`, `char`, `boolean`), which cannot be tracked by JVMTI directly. Some operators like `+` or `&&` are also treated differently from methods.

Our current implementation tracks primitives by identifying repeated values; but this produces inaccuracies when dealing with frequently used primitives like `false` and `0`.

Both these issues could be circumvented by using dynamic bytecode modification to replace the primitives and operators with objects and functions respectively, or by using static analysis to detect the data flow of primitive values. But because James was built as a prototype we bypassed the problem by replacing primitives manually.

In addition, some artefacts used in Java code are translated into compiler-generated methods, and some methods are implemented natively. The JVMTI does not always provide information like local variables for these methods.

Even for normal methods, the amount of information that can be retrieved by JVMTI depends on whether the code was compiled with debug information. In our aim to get a more usable system, we chose to use ways of extracting information that rely on JVMTI methods that also work on Java code that was not compiled with debug information.

5.2 Conceptual limitations

One conceptual limitation is that, in our approach, control dependencies are only tracked for methods that issue HTTP requests (see Sect. 4.3). This means that the model will not consider the consequences of side-effects that are produced by the rest of methods. In future this approach could be extended to cover other methods too.

A generic problem with dynamic approaches – already reported in previous research [10] – is the large number of traces produced by the instrumentation of Java programs, which causes the analysis of relatively small test suites to require a substantial amount of memory and slows down the process considerably. This problem is mitigated by a careful early filtering of the traces collected, as described earlier.

5.3 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 connecting the proxy to the JVMTI agent. Nevertheless, this approach would require a context change between the JVMTI agent and the proxy for each method call, and this would introduce a delay that would slow down the whole process and increase its complexity.

Instead, we track the Java methods that produce HTTP requests. In our case the methods used were `openConnection` and `setRequestMethod` from the class `HttpURLConnection`. Other programs could use different methods but James could be adjusted easily to detect those instead.

6. MODEL GENERATION

Once we have retrieved the dependency information we may use it to generate a model. When displayed as diagrams, models can highlight issues in our test suite, and could ultimately be used as documentation. In Sect. 7 we study an example of one of these diagrams in detail.

6.1 Common dependency graph

Initially, James generates a graph where every call to a method executed directly from the tests is represented as a square-boxed node (see Fig. 2).

Because we are mainly interested in the level of abstraction expressed by the tests, we only incorporate in the model the calls that are executed directly from the tests. But we still include calls necessary to satisfy the data dependencies of other calls already included.

For data dependencies, gray arrows connect the methods that produce a result with those that take that result as a parameter, or those that use the result as a base object, i.e: those methods that are called "on the object returned". The latter are represented with dashed arrows.

For control dependencies, brown arrows connect methods that issued HTTP requests, in order of execution.

6.2 Merging process

A graph generated as described so far would generally be too dense to understand, i.e: it would have too many nodes and arrows. The merging process tries to generalise and simplify the graph while keeping the important information by joining paths with the same topology, similarly to the K-Tails algorithm [4].

James searches every subtree in the graph, alternately following the arrows directly, and in reverse. Then it merges subtrees that contain pairs of methods with the same name and signature, and

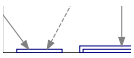
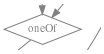
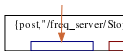
	**Negative instance classes** Calls with keywords like "error" or "fail", and their dependencies are considered negative tests, and marked in pink.
	**Methods `@Test`, `@Before`, and `@After`** The outline represents the places where the command was found, see Table 2.
	**Arrows** Data dependencies are represented with grey arrows. Arrows connect the methods that produce an object as result, with methods that take it as a parameter.
	**Dashed arrows** When an object produced as the result of a method is used as target of another method, (i.e: the `this` object of the method), the dependency relationship is represented with a dashed grey arrow.
	**Brown arrows** Control dependencies are represented through brown arrows. These are created following the order in which the methods were originally executed in the unit tests.
	**Loop highlighting** Loops in control dependencies are represented with thicker arrows.
	**`oneOf` diamonds** We depict only as many continuous grey arrows ending in each node as parameters it takes. To achieve this, James groups arrows by using the `oneOf` diamond nodes.
	**HTTP request grouping (subgraphs)** Methods that are inferred to be related to a HTTP request to the same URL are grouped in subgraphs surrounded by a black rectangle. The tuple in the rectangle denotes the method and URL used.
	**Double outline** Static methods are denoted with double outline. Methods double outline must not have an incoming dashed arrow.

Table 1: Diagram symbol legend

that are connected with the same topology of dependencies both in data and in control.

Longest subtrees are merged first, down to a minimum length K. All tails of the graph (leaf and root nodes) are allowed a lower K; because 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 matching subtrees is due to their small sizes, not to their differences.

Methods that issue HTTP requests are classified into "normal" and "erroneous" (see Table 1) according to whether they have dependent nodes that represent method calls whose name contains the keywords `error` or `fail`.

In addition to these rules, "normal" nodes are never merged with "erroneous" nodes, and data arrows are never merged with control arrows or with arrows that provide dependencies for a different parameter.

When two executions of a method get merged, we may get new alternative paths for satisfying data dependencies of methods. Alternatives for the same parameter are grouped together with a diamond node `oneOf`.

Since we merge only subtrees of a minimum depth, it is likely

@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 2: Outline colour legend for methods

that all the sequences merged have the same or a similar semantics. This way we get new connections and loops both in the dependency and control flow. To make the diagram clearer we group methods that issue the same kind of HTTP request, (i.e: a request to the same URL and with the same HTTP method), into the same subgraph.

Nodes that hang from these nodes and are not a dependency for other nodes that produce a different HTTP request are also included in the same subgraph. We add these nodes to the subgraphs too because, in our experience, they tend to be related to the HTTP request (they are the ones that unmarshall the result or check that the results are correct).

7. DETAILED EXAMPLE

In this section we discuss in detail the result of applying our model extraction methodology by running our James tool on a *frequency server* example, as also used in our original work on model extraction for Erlang/EUnit [2]. The fully extracted machine is presented in Fig. 2.

7.1 Frequency Server example

Frequency Server is a Web Service written using Java that is inspired by the example in the book "Erlang Programming" [6]. 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. In [2], we already used the original version of this example for illustrating the tool for transforming EUnit tests into PBT models.

The API provides four commands: `startServer`, `stopServer`, `allocateFrequency`, and `deallocateFrequency`.

Fig. 2 illustrates the behaviour of the Frequency Server as inferred by the James tool from a set of unit tests.

7.2 Testing the Frequency Server

A test suite has been provided by an independent party and is available[4], the implementation of the SUT used is also accessible in the same link. By using the models generated by James we can generate new relevant tests that explore possibilities that were missing in the original tests. For example, in our particular implementation there is a limit on the number of frequencies that can be allocated, but this limit was not explored by the existing unit tests.

Nevertheless, a random test generator (see Sect. 8) that would randomly traverse the control flow of our model (see Fig. 2) could try to allocate enough frequencies to do so, since there is a control loop around the allocation command. At some point the server will return an error.

Even though in this case the limit in the number of frequencies is an expected functionality, in a bigger example it could be due to a bug, not revealed by legacy unit tests.

[4]https://github.com/palas/freq_server_test_ma

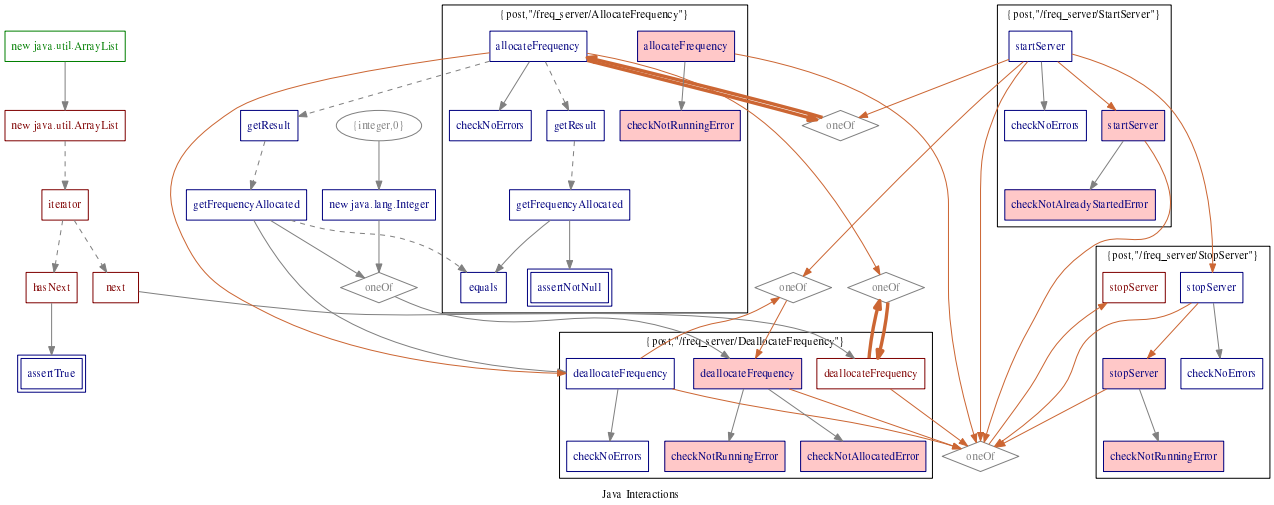


Figure 2: Diagram extracted by James from the Frequency Server

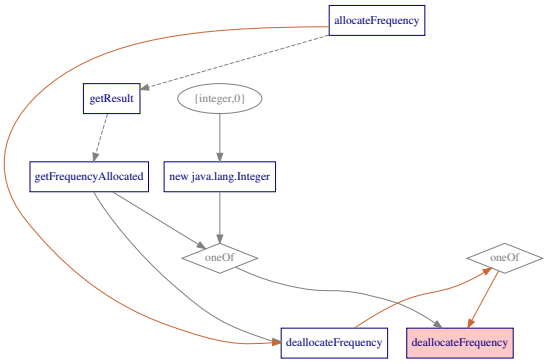


Figure 3: Detail displaying exceptional behaviour

7.3 Interaction of the different features

Looking simultaneously at both control and data flow, we can get a better picture of what the system is expected to do. For example, if we look at the piece of diagram highlighted on Fig. 3, we can see that it is possible to extract the result of the call `allocateFrequency`, and reuse it later when calling `deallocateFrequency`. The first call to `deallocateFrequency` should be valid.

But if, after doing this, we call `deallocateFrequency` a second time, as shown in the same diagram (Fig. 3), we will produce an error, as indicated by the pink background in the left `deallocateFrequency` node. We could also obtain an error result by using the integer `0` as argument, instead of the result of `allocateFrequency`, (we know this is true because the implementation of the Frequency Server used in our experiments starts allocating the frequencies from 10).

8. GENERATING NEW TESTS

Using the approach presented in Sect. 6 we are able to build a comprehensive overview 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), but also to construct a QuickCheck Finite State Machine model for the SUT that will generate new tests when executed. We outline how this is done here, building on the approach first presented in [2].

8.1 Building a state machine

A state machine (namely, a QuickCheck state machine) can be built by translating the different elements of the diagram.

1. State transitions can be defined to match the control flow, (including looping behaviour), given by the brown links in the visualisations. This proceeds according to the mechanism outlined in [2].
2. The data flow dependencies for parameters give an indication of how generators for parameters must call each other recursively and how the values produced by these generators can satisfy the data dependencies for each call 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 Extended Finite State Machine (EFSM). Fig. 3 shows how the result of *allocateFrequency* must be stored in order to be used as a parameter for *deallocateFrequency*.
4. Similarly to the way data flow dependencies are satisfied, we include generators for inverse data dependencies within each subgraph. These will produce the postconditions in terms of the result of the method execution.
5. 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 ensuring that we eventually force the dependency resolution to follow a path with a strictly decreasing depth. In methods with several parameters the depth must include the minimum depths for all parameters.

8.2 Generation of new tests

The QuickCheck models generated as described in Sect. 8.1 are analogous to the diagrams that we can visualise. In Fig. 5 we can see the representation of part of the internal structure used to generate a QuickCheck FSM model for the Frequency Server, and overlayed in black we see the traversal QuickCheck did to generate the test in Fig. 4.

Roughly the following steps are followed:

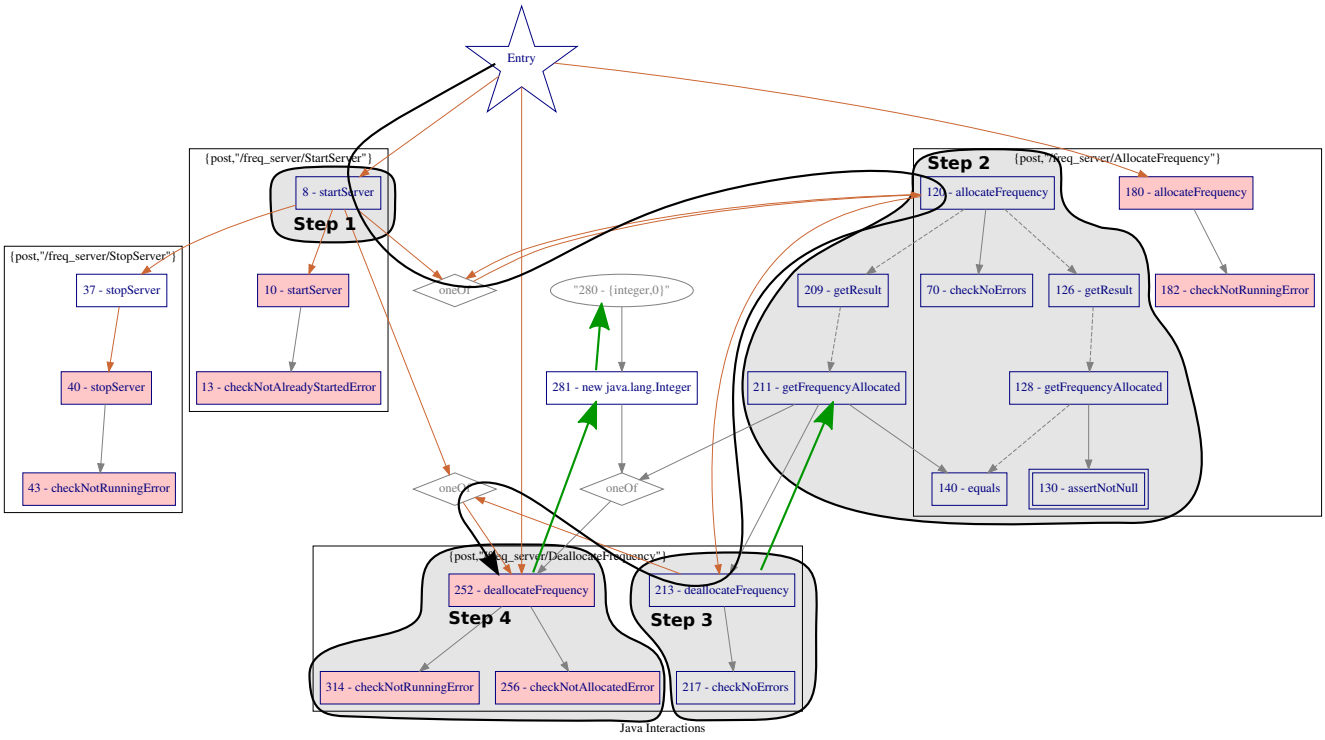


Figure 5: Test generated by James in the diagram

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

Figure 4: Example of test generated by James after manually removing some package qualifiers

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 to the API.
2. For each *step*, we generate the parameters required by following data dependencies upwards (possibly reusing values from previous steps), as shown by the green arrows.
3. Optionally, for each *step*, we generate the postconditions by traversing the data dependencies downwards within the sub-graph.

Given the nature of Web Services, the tests are not supposed to raise any (intended) exceptions. Instead, the results returned by the Web Service can be classified as positive or negative depending on whether they represent an error or a normal result

Once the new test is suitably classified then it is possible to rerun the extraction process and, thus, potentially generate a refined model of the system.

Generated state machines, when run, print new JUnit test cases that can, after manual inspection, be added to the original suite.

Unfortunately, tests generated may not necessarily be correct. That is also the case of the example provided in Fig. 4. Some of the postconditions, e.g:

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

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

9. PILOT STUDY

We have made a pilot study using VoDKATV, which is 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 using web services.

Using the approach explained here, we were able to generate a model from the execution of the JUnit test suite. New test cases were generated automatically from that model, and these new tests allowed developers of the platform to find a previously unknown bug in the implementation.

We now summarise the results of the pilot study; a more detailed report can be found[5].

[5]http://www.prowessproject.eu/wp-content/uploads/2012/10/D6.5_final.pdf

Number of tests needed by the automatic extraction mechanism. It was a concern that the number of tests needed for James to work could be infeasibly big. But during the pilot, James was able to generate new tests even from a single one, even if the tests generated are not very diverse in that case.

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.

Number of additional tests cases needed for the extraction of useful models. The initial set of 28 tests available was enough to produce a useful model for the 20 target operations tested, and so no tests needed to be added. Nevertheless, it was necessary to adapt the exiting JUnit suite in order for James to produce a clear model. In particular:

- Some functions were encapsulated, i.e: making the tests more high level.
- Some methods were rewritten to avoid side effects, i.e: rewriting some parts to use a pure functional style.
- One aspect of the set-up was unfolded into the tests so that generated tests were more accurate.

Number of new test cases generated. During the pilot, James was able to generate thousands of JUnit test cases. Some of these tests where replicated, but this problem can be solved by modifying the generated model to use the QuickCheck macro `?ONCEONLY`.

Some tests were generated that where 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 create a device in a room that does not exist.
- Trying to create two devices with the same MAC.

Number of bugs revealed by means of the extracted models. James helped developers 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.

Time and computational resources. The original suite took to execute between 2.8 and 3.5 seconds, 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.

Developers' rating. The developer of the tests was asked to comment and rate James on a scale from 0 to 10 for accuracy, quality, and usefulness. In summary, their 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 ... 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 ... structure of the original JUnit test suite had to be modified slightly"*

10. CONCLUSION

This paper presents a set of techniques to generate models that combine information from both data and control flow, and for using this model to generate new tests. These techniques have been tested and illustrated with examples extracted from executions of the James tool, and it has been tested in a pilot study involving an industrial Web Service.

We have shown how to extract both control-flow and data-flow information from a JUnit test suite, and implemented that extraction, visualisation of the results, and automatic test generation in the James tool.

For the future, another line of research would be to build on another aspect of [2] and to construct a model from a JUnit test suite without the need of an implementation of the system. Future work could also aim to improve the accuracy and expressiveness of generated models by applying existing techniques like active learning and invariant inference.

Acknowledgment

The authors would like to thank the European Commission for their support of this work through the project PROWESS, http://www.prowess-project.eu/, grant number 317820.

11. REFERENCES

[1] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing telecoms software with Quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*. ACM, 2006.

[2] T. Arts, P. L. Seijas, and S. Thompson. Extracting QuickCheck Specifications from EUnit Test Cases. In *Procs. 10th ACM SIGPLAN Workshop on Erlang*. ACM, 2011.

[3] A. Bertolino et al. Automatic synthesis of behavior protocols for composable web-services. In *Symposium on The foundations of software engineering*. ACM, 2009.

[4] A. W. Biermann and J. A. Feldman. On the Synthesis of Finite-State Machines from Samples of their Behavior. *IEEE Transaction on Computers*, 21, 1972.

[5] K. Bogdanov, N. Walkinshaw, and R. Taylor. StateChum. http://statechum.sourceforge.net/ [last accessed 25-01-2016].

[6] F. Cesarini and S. Thompson. *ERLANG Programming*. O'Reilly Media, Inc., 1st edition, 2009.

[7] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP '00*, pages 268–279. ACM, 2000.

[8] V. Dallmeier et al. Mining object behavior with ADABU. In *Proceedings of the 2006 international workshop on Dynamic systems analysis*. ACM, 2006.

[9] P. Dupont, B. Lambeau, C. Damas, and A. V. Lamsweerde. The QSM algorithm and its application to software behavior model induction. *Applied Artificial Intelligence*, 22, 2008.

[10] D. Lo, S.-C. Khoo, J. Han, and C. Liu. *Mining Software Specifications: Methodologies and Applications*. CRC, 2011.

[11] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *Proceedings of ICSE 2008*. ACM, 2008.

[12] A. Marchetto, P. Tonella, and F. Ricca. State-Based Testing of Ajax Web Applications. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, April 2008.

[13] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *Automated Software Engineering, 2009. ASE'09.* IEEE, 2009.

[14] N. Walkinshaw, R. Taylor, and J. Derrick. Inferring extended finite state machine models from software executions. In *WCRE*. IEEE, 2013.