



Kent Academic Repository

Seijas, Pablo Lamela, Thompson, Simon and Francisco, Miguel Ángel (2018)
Model extraction and test generation from JUnit test suites. **Software Quality Journal, 26 . pp. 1519-1552. ISSN 0963-9314.**

Downloaded from

<https://kar.kent.ac.uk/66343/> The University of Kent's Academic Repository KAR

The version of record is available from

<https://doi.org/10.1007/s11219-017-9399-x>

This document version

Author's Accepted Manuscript

DOI for this version

Licence for this version

UNSPECIFIED

Additional information

Versions of research works

Versions of Record

If this version is the version of record, it is the same as the published version available on the publisher's web site. Cite as the published version.

Author Accepted Manuscripts

If this document is identified as the Author Accepted Manuscript it is the version after peer review but before type setting, copy editing or publisher branding. Cite as Surname, Initial. (Year) 'Title of article'. To be published in *Title of Journal*, Volume and issue numbers [peer-reviewed accepted version]. Available at: DOI or URL (Accessed: date).

Enquiries

If you have questions about this document contact ResearchSupport@kent.ac.uk. Please include the URL of the record in KAR. If you believe that your, or a third party's rights have been compromised through this document please see our [Take Down policy](https://www.kent.ac.uk/guides/kar-the-kent-academic-repository#policies) (available from <https://www.kent.ac.uk/guides/kar-the-kent-academic-repository#policies>).

Model extraction and test generation from JUnit test suites

Pablo Lamela Seijas · Simon Thompson ·
Miguel Ángel Francisco

Received: date / Accepted: date

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 dependencies helps us to build more expressive models, which in turn allows us to produce smarter tests. We illustrate those techniques with real examples produced by our implementation, the James tool, designed to apply these techniques in practice to Java code and tests.

Keywords Model Inference · JUnit · Test Generation · Property Inference · Web Services · Property-based testing · James · QuickCheck

1 Introduction

The effort required to improve the quality of any system – including any web service – is often seen as difficult to justify, because it is not visible in the short term. In the presence of deadlines, therefore, testing and documentation are the phases of development that are the easiest to skip. Nevertheless, quality

Pablo Lamela Seijas
University of Kent
United Kingdom
E-mail: pl240@kent.ac.uk

Simon Thompson
University of Kent
United Kingdom
E-mail: sjt@kent.ac.uk

Miguel Ángel Francisco
Interoud Innovation S.L.
Spain
E-mail: miguel.francisco@madsgroup.org

assurance is what ultimately makes systems reliable, and it can significantly reduce the effort needed for maintaining a system in the long term. In cases of critical systems, a failure could also produce huge economic losses, or worse.

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.

Following Dijkstra’s famous dictum¹, unit tests have a limited effectiveness. They can only cover a finite and predefined set of scenarios, and they are also costly to write. Recent research has delivered new techniques for testing (e.g: property-based testing [2]) that have proved to be more powerful and effective. Unfortunately, these approaches are not very popular because they are less straightforward, and they do not take advantage of the information available in legacy systems.

Property-based testing (PBT) is a technique that uses controlled randomised input to check that the system under test (SUT) complies with a set of expected properties [2]. By using a PBT tool, new tests can be generated automatically and, because of this, more scenarios can be tested with less human effort than by using traditional unit test suites.

In this paper, we provide a mechanism to ease the transition from unit tests to PBT, by studying how to automatically take advantage of legacy unit tests. In particular, we show how existing unit tests can be leveraged to provide more testing value through inferring a model for the SUT. 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 [2] 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 behaviours 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 [3] for Erlang in which finite-state machine models and properties were extracted from EUnit test suites.

¹ “Testing shows the presence, not the absence of bugs.” [24]

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 available², and more technical details can also be found³. This paper is an extended version of [19]; this version includes more detailed explanations and discussions of the algorithm, more information about the pilot study, and new statistics about the tests generated by James for the system used in the pilot study.

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 Section 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 to other types of API (e.g. of a dynamic library) as long as the SUT is tested like a black-box and has a clear interface.

The paper is structured as follows: after introducing property-based testing (Section 2) and related work (Section 3), we motivate the approach taken (Section 4). We then explain the architecture of our implementation (Section 5), the model generation and interpretation (Sections 6 and 7), the test generation (Section 8), and the pilot study (Section 9). We discuss future work and conclude in Section 10.

2 Property-based testing

Property-based testing (PBT) was first developed for Haskell [11], and has been transferred to other programming languages. Quviq QuickCheck [2] (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

² <https://github.com/palas/james>

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

a finite set 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 finite-state machines (FSMs).

3 Related work

Previous approaches [25], [13], and [23], model the expected use of interfaces by focusing 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 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.

An important obstacle to finite-state machine inference is the state explosion problem: the exponential increase in the size of a finite-state model as the number of system components grows [12].

There are several approaches to addressing the state explosion problem. Some (like ours) also combine data and control, but they often rely on data representation, either for clustering [4], or by inferring invariants for parameters and then using them to disambiguate commands in finite-state machines [22, 26].

It can be argued that the use of concrete data content in models can make them easier to understand. But the use of invariants has limited effectiveness when inferring complex properties, or arbitrary semi-structured data, and, again, makes the generation of valid values for parameters challenging, since it becomes a satisfiability problem (finding parameters that satisfy invariants is non-trivial).

Our approach abstracts away from particular values of the data parametrized, relying instead on how the data is generated and used by actions within the system: parameters are treated as black boxes. Thus, it can be applied to parameters that cannot be serialised, or that have a representation that is too lengthy. The approach presented in this paper also provides a mechanism for generating input data explicitly, which makes it more convenient and efficient for test generation purposes.

The work implemented on the Strawberry tool [5] is probably the most similar approach to the one presented in this paper; it also models control and data flow information for extracting specifications of web services, and uses testing to verify conformance. However, in Strawberry, control dependencies are inferred from data dependencies, 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.

There has also been some work on inference of richer models like Visibly Pushdown Automata [16], extended finite-state machines [29, 9], and context-

free grammars [30,17]; the most ambitious approaches often rely to certain extent on general techniques like machine learning, evolutionary programming, and SMT-solvers.

In [7], 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).

Another approach to reducing state-explosion and the amount of information to process when reverse engineering is slicing. Our approach uses dynamic program slicing since it discards traces that do not belong or do not satisfy dependencies required by the JUnit tests used as input. Slicing has been used in the past for model extraction [12,15]. The survey in [27] overviews different existing approaches to program slicing, and [1] surveys the application of slicing to state-based models.

In addition, the merging algorithm applied in this work is strongly influenced by previous regular inference algorithms, in particular K-tails [6] and QSM [14], and by the reverse engineering tool Statechum [8].

We have also used QSM as the core algorithm for our previous work in test generation [3], 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.

Regarding finite-state machine inference (in addition to K-tails and QSM), there has been considerable effort in finding increasingly efficient and accurate state-merging algorithms, which motivates competitions like Abbadingo [20] and Stamina [28].

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. We consider valid inputs to be the ones that satisfy the preconditions (implicit or explicit) required by the SUT. Ideally, invalid inputs will simply cause an exception in the SUT, often an error message will be returned, but the system will remain unaffected (its internal

state will mostly remain unchanged). But some invalid inputs may cause the system to halt or to behave in unexpected ways, and this kind of behaviour is the one we want to avoid, since it can often be translated into a DoS (denial of service) vulnerability that a malicious user could exploit and, in turn, prevent other users from using the system normally.

An example of how almost valid input can be useful would be if, when serialising a request, the unit tests explicitly add quotes surrounding a value, then the generalisation of the request may add quotes in places where they should not occur. This kind of test case would help us detect problems like SQL injection, which, in time, 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 developers, 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 JVMTI⁴). 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`

⁴ <http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html>

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

5 Architecture of the implementation

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

James instruments the JUnit test by attaching to the JVM during their execution. The JVM is observed through a JVMTI agent written in C++ that listens to every method entry and method exit event, and reports it to an Erlang server through a socket. In practice, method entries correspond to

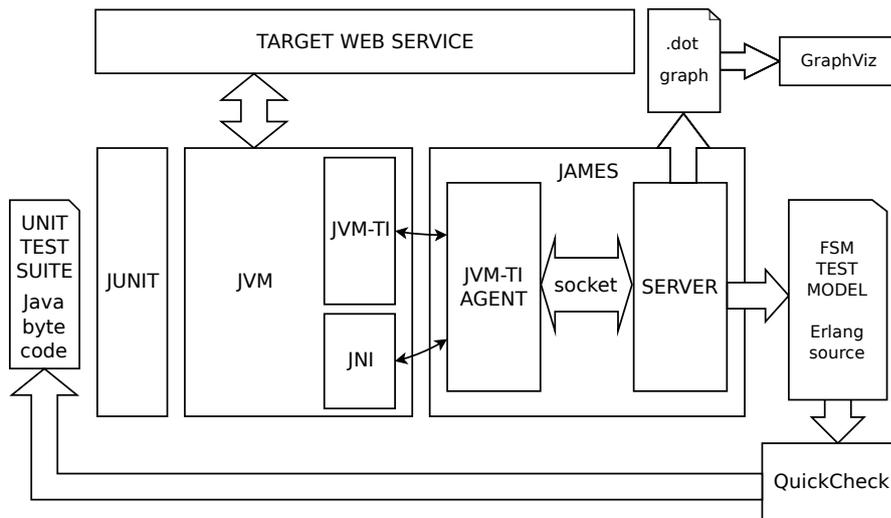


Fig. 1 The architecture of James

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, which is accessed from the JVMTI through the Java Native Interface (JNI).

However, using reflection for each call traced introduces an overhead that makes the whole test suite run unreasonably slowly. To alleviate this problem, James stores, in a cache at the JVMTI agent level, all the classes that are found not to be annotated as JUnit tests, and this way we only use reflection for non-JUnit classes once per class. After this optimisation the instrumentation still impacts the execution time, but to a much lesser extent.

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.

Once filtered, the Erlang server creates a model (see Section 6) that can be serialised as either a `dot` graph that can be rendered with the GraphViz tool, or as a QuickCheck finite-state machine model that can be executed with QuickCheck to generate new JUnit tests (see Section 8).

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 Section 4.3). This means that the model will not consider the consequences of side-effects that are produced by the rest of methods. In the future, this approach could be extended to cover other methods too.

A generic problem with dynamic approaches – already reported in previous research [21] – 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.

A limitation of the application of model inference to test generation is that the inference algorithm cannot guess what the original intention of the developers was and generate correct tests accordingly. The reason is that the model is a simplified description of the actual behaviour of the system, the implementation of the system is typically more intricate and, presumably, closer to the intention of the developers than the model.

As a consequence, postconditions in the tests will often fail due to false positives. But tests generated often explore behaviours that were not considered before, and they provide executable examples of command sequences and

postconditions that could be incorporated to the original test suite after some manual refinement.

Classifying tests into passing and failing can be done just by running them. In fact, James can be connected to the SUT for automatically commenting out the lines of the tests generated that produce exceptions (while they are being generated) so that all tests generated are guaranteed to pass. At the time of writing, this can be done by manually connecting the generated tests to the Erlang version of the Java Erlang Bridge (JEB [18]) and the Java version of the interface JEB to the test suite in the SUT.

Of course, forcing James to generate tests that pass will prevent it from finding any bugs by itself, but it could still be useful as a means of increasing the number of scenarios exercised by a test suite. For example, this would help us increase our confidence in that a refactoring does not alter the behaviour of the system. We can automatically generate tests that pass before the refactoring and then check that they still pass in the refactored version.

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 `URLConnection`. 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 Section 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.

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.

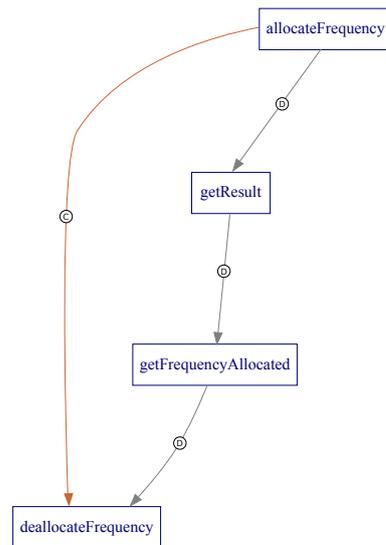


Fig. 2 Small example of control and data dependencies combined

For data dependencies, gray arrows (in this paper also marked with a “D” in a circle) 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 (in this paper also marked with a “C” in a circle) connect methods that issued HTTP requests, in order of execution.

In Figure 2, we can see a small example extracted from a bigger model where both control and data dependencies are present.

6.2 Merging process

A graph generated by following only the steps 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 [6].

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 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.

The usage of a constant K is inspired by the K-Tails algorithm [6]. 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 constant K is a parameter of the algorithm that 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 K is, the more conservative the merging process is.

Our merging algorithm uses the idea of K 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 K to limit the exploration of both. Lastly, our nodes have labels, they do not represent states but “method calls”, thus, even for a 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 the bigger K , and 1 for the lower K .

In Figures 3, 4, 5, and 6, we present an approximate pseudocode description of the merging algorithm.

The `merging_algorithm` function in Figure 3 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 1 (on page 35).

In Figure 5, 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

```

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())
        }
    }
}

```

Fig. 3 Merging algorithm base function pseudo-code

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 4 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 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 6 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.

In addition to that, methods that issue HTTP requests are classified into “normal” and “erroneous” (coloured in pink and, in this paper, also marked with an “N” in a circle, see Figure 7) according to whether they have dependent nodes that represent method calls whose name contains the keywords `error` or `fail`.

To make the diagram clearer we also 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 (which is marked with a black rectangle that has the URL and HTTP method in the title, see Figure 7).

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,

```

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
  }
}

```

Fig. 4 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"
}

```

Fig. 5 Example subtree data type pseudo-code

they tend to be related to the HTTP request (they are the ones that unmarshall the result or check that the results are correct).

“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` (see Figure 8).

Since we merge only subtrees of a minimum depth, it is likely 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 (see Figure 9). James highlights loops in diagrams by making all arrows that are part of a loop thicker than arrows that are not.

A legend with examples of the ways in which information is presented in the generated diagrams can be found in Table 1 (on page 35), and information about the meaning of the different colors in the outlines of nodes can be found in Table 2 (on page 36).

```

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)
        }
    }
}

```

Fig. 6 Equivalent subtree search pseudo-code

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 [3]. The fully extracted machine is presented in Figure 10.

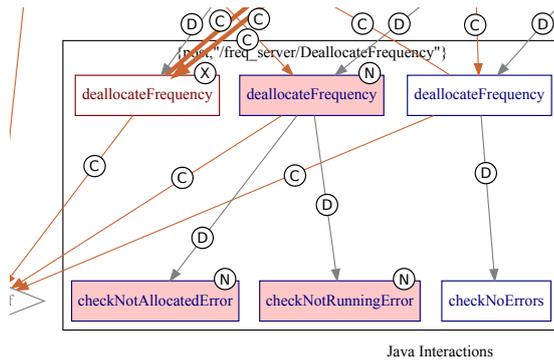


Fig. 7 Small example of “normal” and “erroneous” method calls

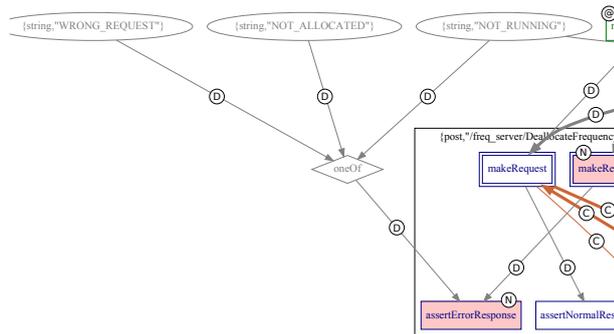


Fig. 8 Small example of one-of diamond

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” [10]. 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 [3], 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`.

Figure 10 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 (the company Interoud Innovation) and is available⁵; the implementation of the SUT used is also

⁵ https://github.com/palass/freq_server_test_ma

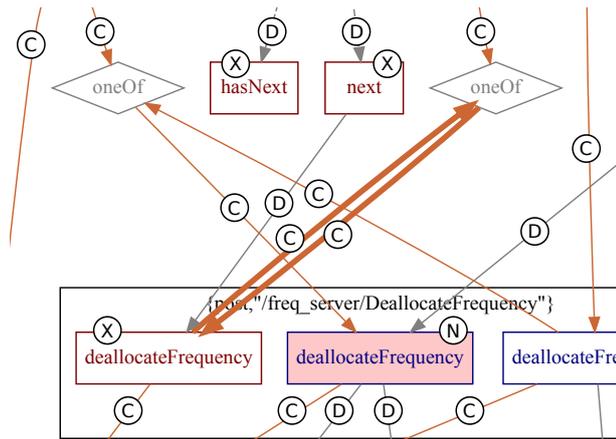


Fig. 9 Small example of loop in control flow

accessible in the same link. By using the models generated by James we can generate new tests that have a similar structure to the ones provided but still 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 Section 8) that would randomly traverse the control flow of our model (see Figure 10) 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.

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 Figure 11, 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 (Figure 11), we will produce an error, as indicated by the pink background and (in this paper) an “N” in a circle in the right `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).

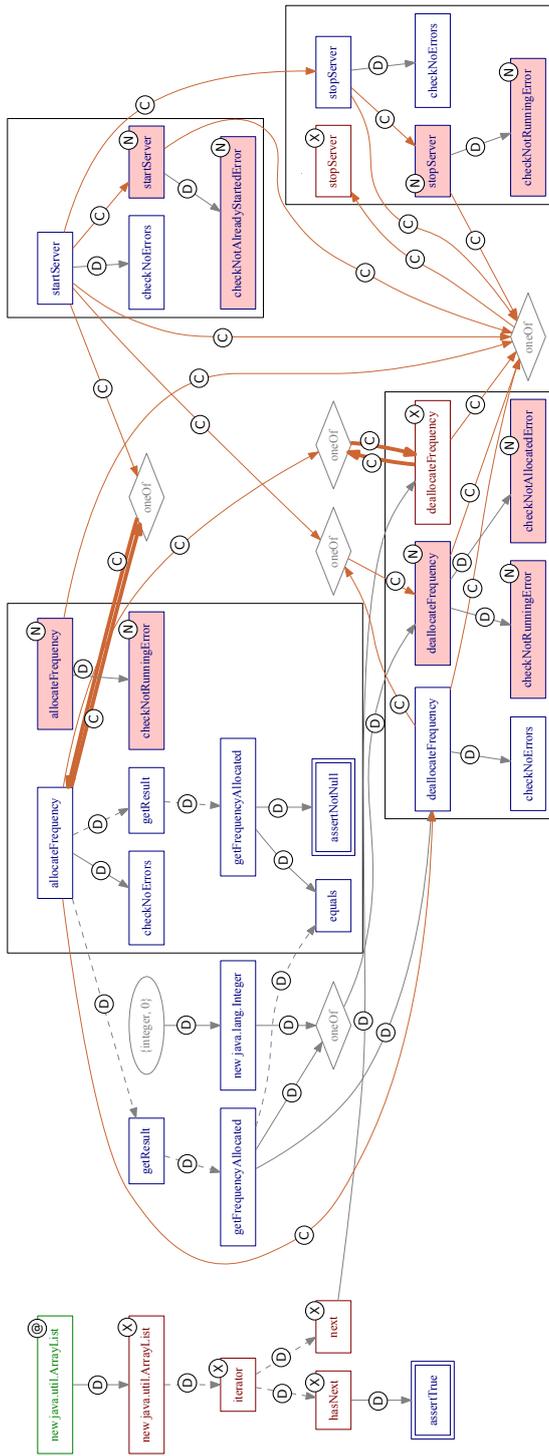


Fig. 10 Diagram extracted by James from the Frequency Server (circle labels added manually)

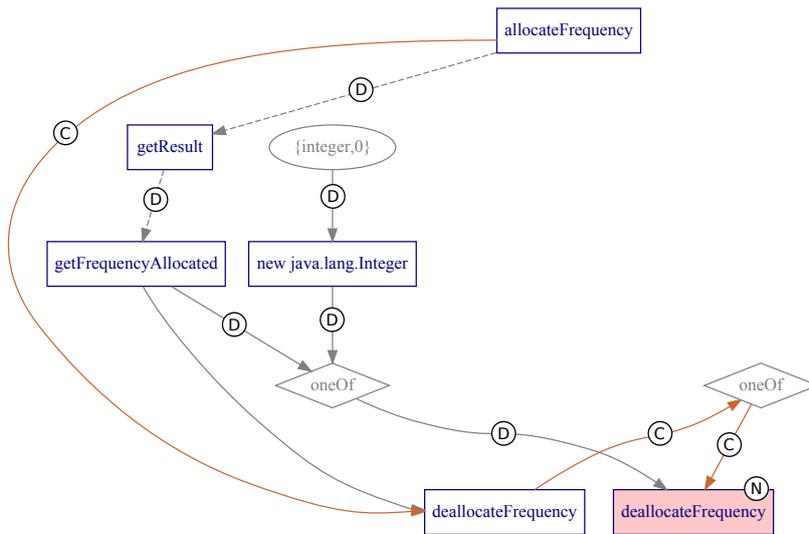


Fig. 11 Detail displaying exceptional behaviour

8 Generating new tests

Using the approach presented in Section 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 [3].

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 [3].
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). Figure 11 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 Section 8.1 are analogous to the diagrams that we can visualise. In Figure 12, we can see the representation of part of the internal structure used to generate a QuickCheck FSM model for the Frequency Server, and overlaid in black (arrow marked with the letter “B” in a circle) we see the traversal QuickCheck did to generate the test in Figure 13 (the tests generated by James have full package qualifiers for every class, we have removed some of them manually for clarity).

Roughly the following steps are followed:

1. The graph is traversed randomly through the control path (from the entry star through the brown arrows, marked with a “C” in a circle), 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 (marked with the letter “G” in a circle).
3. Optionally, for each *step*, we generate the postconditions by traversing the data dependencies downwards within the subgraph.

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 Figure 13. Some of the postconditions, e.g:

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

will fail, and this issue needs to be solved manually. The difficulty in solving the errors in tests is not getting them to pass (this can be done automatically

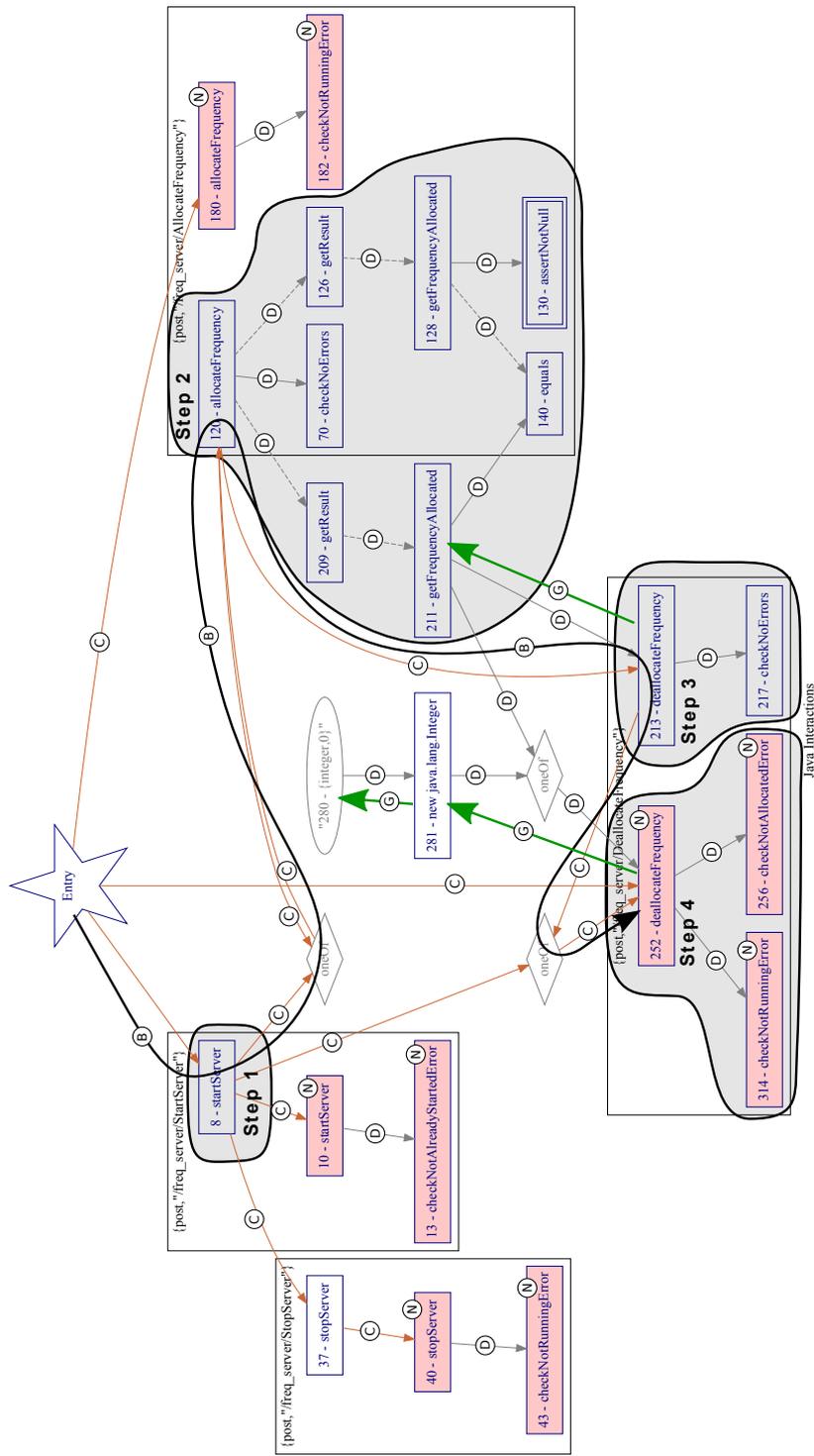


Fig. 12 Test generated by James in the diagram

```

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

```

Fig. 13 Example of test generated by James after removing some package qualifiers

by commenting out the postconditions that fail when running the tests), but in making sure that it is the test what is wrong and not the system. Alternatively, the tests can be fixed by replacing wrong postconditions with appropriate ones. We say appropriate ones because it is also trivial to replace invalid postconditions with valid ones by negating them, for example: replacing `assertTrue` with `assertFalse`, or `assertEqual` with `assertNotEqual`.

9 Pilot study

As part of this work we have carried out a pilot study using part of VoDKATV as system under test. VoDKATV is a middleware that provides end users with access to different services on a TV screen, e.g: watching Internet TV channels, accessing paid video-on-demand services, browsing the Internet, playing games, etc. VoDKATV is the system that connects all the media streams and provides the contents to the user. VoDKATV supports different types of devices to access the system, like set-top-boxes, PCs, smartphones, or tablets; in order to provide end-users with a real multi-screen experience.

In this section, we present the results of the pilot study; a more detailed report can be found⁶.

9.1 VoDKATV

The architecture of the VoDKATV system is depicted in Figure 14, in which we represent the main components of the implementation and their interactions. The VoDKATV-server component is the core component of the VoDKATV system. This component stores information about the users that can access the system, their access rights, resource consumption, preferences, etc. This component provides different integration APIs to access the user information.

VoDKATV-server offers an HTTP/XML integration API so that administration applications can access and modify the data stored on the system. Thus, this API is used by system administrators to create new users in the system, manage channels, permissions, etc.

Prior to the study, there were some JUnit test cases for the HTTP/XML API, written manually by the company Interoud Innovation. These test cases were implemented using JUnit and XPath expressions to check that the XML files returned by VoDKATV contain the expected information.

The HTTP/XML API contains 186 operations, with 256 JUnit tests cases in total. For the pilot study we considered a subset of 28 JUnit tests that target 20 operations related to rooms and devices. We chose the subset of tests related to rooms and set-top-boxes because it was both simple enough to be easy to understand, and complex enough to produce interesting results and exploit the potential of James (for example, the models obtained have loops). We could have chosen other parts of the API, but they were either too simple (like API operations to configure key-value parameters), or they had many dependencies with other parts (for example, the API part related to users depends on the rooms part).

We tried combining tests for different modules of the system, but the combined models often had independent “islands” in the model. Due to the mechanism used for test generation, we know that methods that are in separate islands (methods that have no connection in the control or the data flow) will not produce any tests that combine the methods in those islands. Thus, it is not beneficial to try to simultaneously model test that have no commonalities (which is often the case with unitary tests, since they are targeted at individual units or components of the system). On the other hand, modelling tests for several components simultaneously, increases the complexity and memory necessary to run the inference algorithm, this (together with the need for human effort) justifies the application of the method to subsystems.

By using the selected subset of JUnit tests as input, James was used to generate a model. New test cases were generated automatically from that model, and these new tests allowed a developer of the platform to find a previously unknown bug in the implementation.

⁶ http://www.prowessproject.eu/wp-content/uploads/2012/10/D6.5_final.pdf

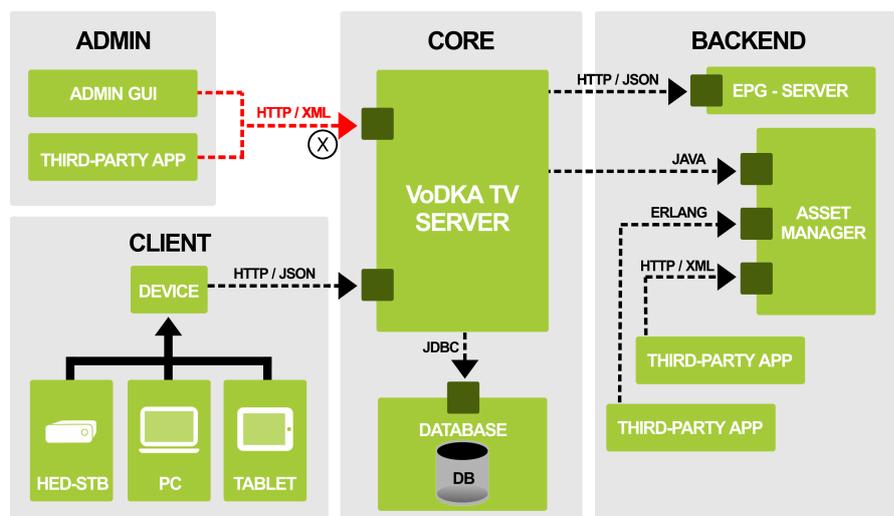


Fig. 14 VoDKATV (target of the pilot study in red and marked with an “X” in a circle)

The entity “room” can represent different things depending on the environment where VoDKATV is installed; for instance, it represents a physical room when VoDKATV is installed in a hotel, or a household in a telco deployment.

In addition, a VoDKATV “room” can contain several “devices”, usually, set-top-boxes, which are identified by their MAC address. Rooms and devices can have additional attributes, such as a description or a label.

The operations selected for the pilot study allow to create, modify, delete and search for, using different criteria, rooms and devices. For example, the names of some operations as they appear in the WSDL specification are: `CreateRoom`, `FindAllRooms`, `DeleteRoom`, `CreateDevice`, `FindDeviceById`, `FindDevices`, `FindDevicesByRoom`, `UpdateDevice`, `DeleteDevice`, etc.

9.2 Research Questions

The pilot study allowed us to evaluate James using a real software system. In order to guide the pilot study, we aimed to answer a series of four research questions that tried to evaluate the usefulness of James in the real world:

- 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? Can this be shown by discovering new faults in the system under test?
- D: Is the method assessed as accurate, quality enhancing, and useful, by the developer involved in the pilot study?

9.3 Experiments

In order to answer the research questions, during the pilot study, we measured the following data:

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

9.4 Results

Using the 28 existing tests of VoDKATV, we applied James to generate a model which is too big to include in this paper (it consists 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.

For this extended version of the paper, we have run James on the same 28 tests a second time for gathering extra information, with the aim of giving a better sense on the quality of the tests generated by James. In this second run of James, we have generated 10,000 random tests of which 3,206 were unique. By using the JEB interface [18], James ran the tests as they were generated and commented out the instructions that caused exceptions. Using this information we obtained the following results.

In Table 3 (on page 36), we show the distribution of postconditions generated in tests and the amount of passing postconditions in each case. Cells with gray 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 tests generated raised any exceptions; thus, all generated failing tests were due to failing postconditions.

In Table 4 (on page 36), we show how many methods in each test issue HTTP requests in the tests generated, and the average number of methods, postconditions, and instructions per test.

In Table 5 (on page 37), we show the results of manually classifying the first 30 tests generated during the second run. The manual classification was carried out by a member of Interoud Innovation by using the following criteria:

- 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 none 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 3 (on page 36).

In Table 6 (on page 37), we show the manual classification of the tests that had at least one interesting part, according to the following categories:

- Positive tests: e.g, 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: e.g, an attempt is done to delete a non-existent device or to create a room with incorrect data, and the postcondition checks that the result is an error.

Number of tests in a unit test suite needed for automatic extraction of useful test models. During the pilot study, James was able to generate new tests even from a single one, even if the tests generated are not very diverse in that case.

From a single test that creates and deletes a room, and checks whether the room was created and deleted correctly, new test cases have been generated by James. For example, James generated a test case that tries to delete a non-existing room. The following is an excerpt of that test in which the package qualifiers and some irrelevant instructions have been manually removed. It can be seen which instructions were removed from the excerpt because the variable names contain the instruction number on which they were originally assigned; for example, `var9` was assigned on the ninth instruction of the original test as generated by James, this tells us that, between the places where `var5` and `var9` were assigned, two instructions were removed.

```
@Test
public void testGenerated2() throws Exception {
    String var1 = getRandomStrId();
    String var2 = "sample description";
    Object var3 = null;
    String var4 = this.generateRoomXML(var1, var2, var3);
    String var5 = this.createRoom(var4);
    this.checkThatRoomWasCreated(var4, var1, var2, var3, var5);
    String var9 = getRandomStrId();
    String var10 = this.deleteRooms(var9);
}
```

```

    this.checkThatRoomWasDeleted(var1, var10);
}

```

This test case fails, because the result returned by the operation `deleteRooms` is an error that shows that the room with the identifier stored in the variable `var9` (generated using the function `getRandomStrId`, which generates a unique string) cannot be deleted because it does not exist in the VoDKATV system, since it has not been created before. In this case, the tester must fix the test by removing the call to the function `checkThatRoomWasDeleted`, and checking that VoDKATV returns a `not_found` error.

Number of additional test cases needed for the automatic extraction of useful models. The initial set of 28 tests available during the pilot study was enough to produce a useful model for the 20 target operations tested, and so no tests needed to be added. Therefore, it was not necessary to include additional test cases in the original test suite to generate useful JUnit test cases automatically. Nevertheless, it was necessary to adapt the existing 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.

The effort required for this adaptation depends on the original structure and functionalities of Java required by the tests used as input. In the case of this pilot study, the rewriting was carried out by a single person and took approximately one day of work.

For example, the following code in one of the tests:

```

Map<String, String[]> deleteParams =
    new HashMap<String, String[]>();
deleteParams.put(API.CONFIGURATION_DELETEDEVICE_PARAM_DEVICEID,
    new String[] {deviceId1, deviceId2});
logger.info("deleteParams: {}",
    HTTPFacade.paramsToString(deleteParams));
String resultDelete =
    HTTPFacade.doGet(API.CONFIGURATION_DELETEDEVICE,
        deleteParams);
logger.info("resultDelete: {}", resultDelete);
assertFalse(resultDelete.isEmpty());
assertXPathNotExists("/devices/errors", resultDelete);
assertXPathExists("/devices/device/id[.=' " + deviceId1 + "']",
    resultDelete);
assertXPathExists("/devices/device/id[.=' " + deviceId2 + "']",
    resultDelete);

```

Was rewritten to:

```

List<String> deviceIds = new ArrayList<String>();
addString(deviceId1, deviceIds);
addString(deviceId2, deviceIds);
String resultDelete = deleteDevices(deviceIds);
checkThatDeviceWasDeleted(deviceIds, resultDelete);

```

This avoids the use of constructors with templates and arrays (which are not supported by the current version of James), and allows James to generalise the test to any number of devices easily, since it only needs to add more invocations of the method `addString` and the rest of the code can stay unaltered.

Number of new test cases added to test suite by means of the extracted models. During the pilot, James was able to generate thousands of JUnit test cases.

Some tests were generated that explore aspects that were 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.

Other tests were replicated, i.e: James generated several test cases that were equivalent to other of the tests generated or to the ones used as input. The problem with tests generated several times can be solved by modifying the generated model to use the QuickCheck macro `?ONCEONLY`; of course, this does not prevent QuickCheck from generating tests that are semantically equivalent. Detecting that two tests are equivalent is undecidable in the general case.

Additionally, some of the tests generated by James make no sense, for example, the following test case, generated automatically by James (after removing some irrelevant instructions and package qualifiers):

```
@Test
public void testGenerated3() throws Exception {
    String var1 = getRandomStrId();
    String var2 = "sample description 1";
    String var4 = this.generateRoomXML(var1, var2, null);
    String var5 = this.createRoom(var4);
    String var6 = "sample description";
    this.checkThatRoomWasCreated(var4, var1, var6, null, var5);
    String var9 = getRandomStrId();
    java.util.ArrayList var10 = new java.util.ArrayList();
    java.util.List var11 = this.addString(var9, var10);
    String var12 = this.deleteRooms(var11);
    this.checkThatRoomWasDeleted(var1, var12);
}
```

This test case creates a new room in the VoDKATV system using the operation called `createRoom`. The room is created with a random (and unique) identifier, stored in the variable `var1`, and with the description “sample description 1” (stored in the variable `var2`). After that, it checks that the room has been created by using the method `checkThatRoomWasCreated`, which receives the XML used as the input for the VoDKATV system as the first parameter, the room identifier as the second parameter, the room description as the third parameter, a label associated to the room as the fourth parameter, and the XML returned by the VoDKATV system as the fifth parameter.

The invocation to this method is checking that the newly created room returned by the VoDKATV system has the identifier stored in the variable `var1`, which is correct. However, it is also checking that the description of the room is the one stored in `var6` (“sample description”), which is not correct.

Another reason why postconditions generated may fail is because any of the instructions in a sequence fails for some reason and the final result is not as expected by the test. For example, the first test of the second run of James tries to delete a device that it previously tried to create:

```
Object var11 = null;
String var12 = getRandomMACAddress();
String var13 = "hed3_webkit";
Object var14 = null;
String var15 = "Random Device";
Long var16 = getRootTagIdInTagsTree();
String var17 = this.generateDeviceXML(
    (java.lang.String) var11, var12, var13,
    (java.lang.String) var14, var15, var16);
String var18 = this.createDevice(var17);
String var19 = this.getDeviceId(var18);
String var20 = this.deleteDevicesS(var19);
this.checkThatDeviceWasDeleted(var19, var20);
```

Nevertheless, the creation of device `createDevice` fails because the input values are wrong. In particular, `var14`, used as identifier for the room to which the device is associated is `null`. Thus, the deletion of the device fails, since it has not been created. The way of fixing the test would be to create a valid room to which the device can be associated by, for example, using the following instructions:

```
String var1 = getRandomStrId();
String var2 = "sample description 1";
Long var3 = getRootTagIdInTagsTree();
String var4 = this.generateRoomXML(var1, var2, var3);
String var5 = this.createRoom(var4);
```

And by using the new `var1` instead of `var14`.

Finally, a postcondition may try to use parameters that are not appropriate. For example, the twentieth test generated in the second run of James creates a room with a set of values, and later checks whether the room was created by using a different set of values:

```
String var1 = getRandomStrId();
String var2 = "sample description 1";
Long var3 = getRootTagIdInTagsTree();
String var4 = this.generateRoomXML(var1, var2, var3);
String var5 = this.createRoom(var4);
String var6 = "sample description 2";
this.checkRoomCreateOrUpdate(var4, var1, var6, var3, var5);
```

We can fix this test by ensuring that the creation and check are done using the same values.

```
String var1 = getRandomStrId();
String var2 = "sample description 1";
```

```

Long var3 = getRootTagIdInTagsTree();
String var4 = this.generateRoomXML(var1, var2, var3);
String var5 = this.createRoom(var4);
String var6 = "sample description 1";
this.checkRoomCreateOrUpdate(var4, var1, var6, var3, var5);

```

Number of bugs revealed by means of the extracted models. James helped a developer find one wrong behaviour. When the operation that deletes a device was invoked with an empty device identifier it produced an uncontrolled internal error caused by a `NullPointerException` in the Java code, instead of returning a “required field” error as expected.

The error generated the following XML response:

```

<devices>
  <errors>
    <error>
      <code>internal_error</code>
      <description>Internal error, check VoDKA.TV logs
    </description>
    </error>
  </errors>
</devices>

```

This is the new test case that causes this behaviour (after some manual cleanup):

```

@Test
public void generatedTest47() throws Exception {
    String var1 = getRandomMACAddress();
    String var2 = "IPHONE";
    String var3 = getRandomStrId();
    String var4 = "Random Device";
    Long var5 = null;
    Long var6 = getRootTagIdInTagsTree();
    String var7 = this.generateDeviceToCreateXML(var1, var2, var3,
                                                var4, var5);

    String var8 = this.createDevice(var7);
    String var9 = this.getDeviceId(var8);
    String var10 = this.deleteDevices(var9);
    this.checkThatDeviceWasDeleted(var9, var10);
}

```

The use of an empty identifier to invoke the operation to delete a device in this test case is just a coincidence. The reason is that the operation to create a device is invoked with a room that does not exist and, therefore, the device is not created. However, the result is not checked, so the test case continues running. The result of this operation is used to get the device identifier of the new device created, by using the operation `getDeviceId`, which is empty because no device has been created. The obtained device identifier is the string used as a parameter for the operation `deleteDevices`. Nevertheless, in order for the test to pass, it needs to be fixed, because it checks that the device was deleted successfully, but this is not true.

This bug has been fixed and the new implementation returns a “required field” error instead.

Time and computational resources needed to infer and generate the models. 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.

Developer’s rating of accuracy, quality and usefulness of the extracted models, compared to previously existing unit test suites. The developer of the tests was asked to comment and rate James on a scale from 0 to 10 for accuracy, quality, and usefulness. Their assessment for this part was:

- Accuracy: 4. *“The current version of 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 tested scenario), and test cases that have been already included in the original test suite (or in the new test cases). In addition, in the first version of James, used to carry out this pilot, some test cases do not compile. These compilation errors can be fixed automatically by well known IDEs like Eclipse, and in the end this wrong behaviour was fixed in James and all the generated test cases do compile. Even so, substantial manual work to examine and analyse all the generated JUnit test cases is necessary to filter out incorrect and duplicated test cases.”*
- Quality: 7. *“The new test cases generated by James follow the same style and guidelines used in the original Java code, as they use the same methods defined in the original test suite in the same way. Therefore, the source code of the new test cases can be understood by any person who also understands the code of the original JUnit test suite. Hence we consider that the quality of the new test cases in terms of source code quality is similar to the original test suite. However, there is a possible enhancement that would considerably improve the readability of the new test cases: the variable names used in the new test cases are called `var1`, `var2`, `var3`, etc. which makes the new test cases harder to read. One lesson learned is that we might improve this with more meaningful names for the variables, for example, if a variable is going to store the result of the method `getDeviceId()`, that variable could be called `deviceId` (or `deviceId1`, `deviceId2`, `deviceId3`, etc. if there are several invocations to that method).”*
- Usefulness: 8. *“Using James with an existing JUnit test suite helped to identify some situations that had not been tested before. In addition, the new JUnit test suite detected an unknown wrong behaviour in the system. The reason why we do not give the maximal rating here is that the structure of the original JUnit test suite had to be modified slightly so that James could work appropriately.”*

9.5 Result discussion

In the previous section, we have presented the results of some experiments that test the applicability of James to industrial systems. However, the sample size of the experiments carried out was small, thus, further validation would be required in order to obtain conclusions that are valid for the general case.

Nevertheless, in this section, we interpret the results to try to answer the research questions presented in Section 9.2; to obtain some preliminary insight about the effectiveness of the approach, and to guide future research:

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

We can conclude that it is indeed feasible to augment existing Java test suites automatically by using James, even if there are no many tests. But the results may benefit from some manual refinement of the input test suite.

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?

Our experiments show that, at least for small systems (or parts of systems), it is indeed feasible. The instrumentation increases the cost of execution and may require some manual work, but only to a certain extent. The main limitation for scalability to big systems would probably be the memory consumption required for trace collection and, for this particular implementation, the requirement of human effort both to adapt the existing tests and to validate the tests generated.

C: Do the inferred tests effectively augment the existing tests? Can this be shown by discovering new faults in the system under test?

The bug detected thanks to the new tests shows that indeed the generated tests help discovering new faults in the system.

D: Is the method assessed as accurate, quality enhancing, and useful, by the developer involved in the pilot study?

The method has been assessed as useful and its output has been evaluated as having similar quality to the input. But there is concern about the accuracy of the tests generated since a big part of them tends to exercise uninteresting aspects of the system and because many postconditions generated do not pass.

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 [3] 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.

Acknowledgements 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.

References

1. K. Androutopoulos, D. Clark, M. Harman, J. Krinke, and L. Tratt. State-based model slicing: A survey. *ACM Computing Surveys (CSUR)*, 45(4):53, 2013.
2. T. Arts, J. Hughes, J. Johansson, and U. T. Wiger. Testing telecoms software with Quviq QuickCheck. In *Erlang Workshop*, pages 2–10. ACM, 2006.
3. T. Arts, P. Lamela Seijas, and S. J. Thompson. Extracting QuickCheck specifications from EUnit test cases. In *Erlang Workshop*, pages 62–71. ACM, 2011.
4. T. Berg, B. Jonsson, and H. Raffelt. Regular inference for state machines with parameters. In *International Conference on Fundamental Approaches to Software Engineering*, pages 107–121. Springer, 2006.
5. A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli. Automatic synthesis of behavior protocols for composable web-services. In *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 141–150. ACM, 2009.
6. A. W. Biermann and J. A. Feldman. On the Synthesis of Finite-State Machines from Samples of Their Behavior. *IEEE Transactions on Computers*, 21(6):592–597, 1972.
7. A. W. Biermann and R. Krishnaswamy. Constructing programs from example computations. *IEEE Transactions on Software Engineering*, (3):141–153, 1976.
8. K. Bogdanov, N. Walkinshaw, and R. Taylor. StateChum. <http://statechum.sourceforge.net/> [last accessed 25-01-2016].
9. S. Cassel, F. Howar, B. Jonsson, and B. Steffen. Active learning for extended finite state machines. *Formal Aspects of Computing*, 28(2):233–263, 2016.
10. F. Cesarini and S. Thompson. *Erlang Programming - A Concurrent Approach to Software Development*. O’Reilly, 2009.
11. K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming*, pages 268–279, 2000.
12. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, H. Zheng, et al. Bandera: Extracting finite-state models from java source code. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 439–448. IEEE, 2000.
13. V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *International Workshop on Dynamic Systems Analysis*, pages 17–24. ACM, 2006.
14. 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.
15. J. Hatcliff, M. B. Dwyer, and H. Zheng. Slicing software for model construction. *Higher-order and symbolic computation*, 13(4):315–353, 2000.
16. M. Isberner. *Foundations of active automata learning: an algorithmic perspective*. PhD thesis, 2015.

17. F. Javed, B. R. Bryant, M. Črepinšek, M. Mernik, and A. Sprague. Context-free grammar induction using genetic programming. In *42nd annual Southeast regional conference*, pages 404–405. ACM, 2004.
18. P. Lamela Seijas. Java Erlang Bridge. <https://github.com/palaso/jeb> [last accessed 29th June 2017].
19. P. Lamela Seijas, S. Thompson, and M. Á. Francisco. Model extraction and test generation from JUnit test suites. In *International Workshop on Automation of Software Test*, pages 8–14. ACM, 2016.
20. K. J. Lang, B. A. Pearlmutter, and A. Rodney. Results of the abbadingo one dfa learning competition and new evidence driven state merging algorithm. In *ICGI'98: The 4th International Colloquium on Grammatical Inference*.
21. D. Lo, S.-C. Khoo, J. Han, and C. Liu. *Mining Software Specifications: Methodologies and Applications*. CRC Press, 2011.
22. D. Lorenzoli, L. Mariani, and M. Pezzè. Inferring state-based behavior models. In *Proceedings of the 2006 international workshop on Dynamic systems analysis*, pages 25–32. ACM, 2006.
23. A. Marchetto, P. Tonella, and F. Ricca. State-Based Testing of Ajax Web Applications. In *IEEE International Conference on Software Testing Verification and Validation*, pages 121–130. IEEE Computer Society, 2008.
24. P. Naur and B. Randell. *Software Engineering: Report on a conference sponsored by the NATO SCIENCE COMMITTEE*. Nato, 1969.
25. M. Pradel and T. R. Gross. Automatic Generation of Object Usage Specifications from Large Method Traces. In *International Conference on Automated Software Engineering*, pages 371–382. IEEE Computer Society, 2009.
26. M. Shahbaz, K. Li, and R. Groz. Learning and integration of parameterized components through testing. In *Testing of Software and Communicating Systems*, pages 319–334. Springer, 2007.
27. F. Tip. A survey of program slicing techniques. <http://www.franktip.org/pubs/jp11995.pdf>, 1994.
28. N. Walkinshaw, B. Lambeau, C. Damas, K. Bogdanov, and P. Dupont. Stamina: a competition to encourage the development and assessment of software model inference techniques. *Empirical software engineering*, 18(4):791–824, 2013.
29. N. Walkinshaw, R. Taylor, and J. Derrick. Inferring extended finite state machine models from software executions. *Empirical Software Engineering*, 21(3):811–853, 2016.
30. P. Wyard. Context free grammar induction using genetic algorithms. In *Grammatical Inference: Theory, Applications and Alternatives, IEE Colloquium on*, pages 514–518. IET, 1993.

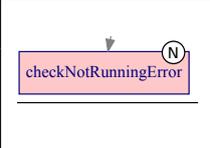
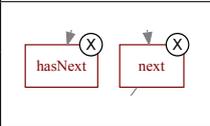
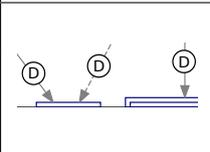
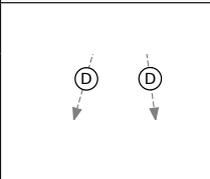
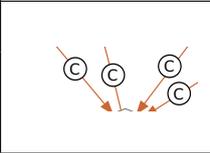
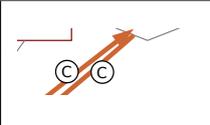
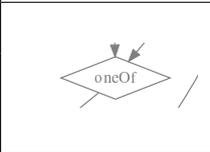
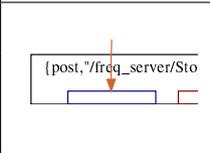
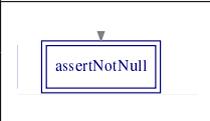
	<p>Negative instance classes</p> <p>Calls with keywords like “error” or “fail”, and their dependencies are considered negative tests, and marked in pink. In this paper, they are also marked with the letter “N” in a circle.</p>
	<p>Methods @Test, @Before, and @After</p> <p>The colour of the outline represents the places where the command was found, see Table 2 (on page 36).</p>
	<p>Arrows</p> <p>Data dependencies are represented with grey arrows (marked with a “D”). Arrows connect the methods that produce an object as result, with methods that take it as a parameter.</p>
	<p>Dashed arrows</p> <p>When an object produced as the result of a method is used as target of another method (i.e: the <code>this</code> object of the method), the dependency relationship is represented with a dashed grey arrow (marked with a “D”).</p>
	<p>Brown arrows</p> <p>Control dependencies are represented through brown arrows (marked with a “C”). These are created following the order in which the methods were originally executed in the unit tests.</p>
	<p>Loop highlighting</p> <p>Loops in control dependencies are represented with thicker arrows.</p>
	<p>oneOf diamonds</p> <p>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 <code>oneOf</code> diamond nodes.</p>
	<p>HTTP request grouping (subgraphs)</p> <p>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.</p>
	<p>Double outline</p> <p>Static methods are denoted with double outline. Methods double outline must not have an incoming dashed arrow.</p>

Table 1 Diagram symbol legend

@Before	@Test	@After	Outline colour	Circle label
No	No	No	Grey 	-
Yes	No	No	Green 	Ⓢ
No	Yes	No	Blue 	no label
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

All postconditions	Postconditions that pass				TOTAL TESTS
	0	1	2	3	
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 3 Distribution of postconditions in tests generated.

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 4 Distribution of methods that produce HTTP requests in tests generated.

Classification	Matching tests
1 interesting part	10
2 interesting parts	2
Both interesting and not	8
Only non-interesting parts	9
TOTAL	30

Table 5 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 6 Manual evaluation of positive or negative testing for interesting tests.