EasyChair Preprint

№ 414

Fine-Grained Test Minimization

Arash Vahabzadeh, Andrea Stocco and Ali Mesbah

August 9, 2018

Fine-Grained Test Minimization

Arash Vahabzadeh
University of British Columbia
Vancouver, BC, Canada
arashvhb@ece.ubc.ca

Andrea Stocco
University of British Columbia
Vancouver, BC, Canada
astocco@ece.ubc.ca

Ali Mesbah
University of British Columbia
Vancouver, BC, Canada
amesbah@ece.ubc.ca

ABSTRACT

As a software system evolves, its test suite can accumulate redundancies over time. Test minimization aims at removing redundant test cases. However, current techniques remove whole test cases from the test suite using test adequacy criteria, such as code coverage. This has two limitations, namely (1) by removing a whole test case the corresponding test assertions are also lost, which can inhibit test suite effectiveness, (2) the issue of partly redundant test cases, i.e., tests with redundant test statements, is ignored. We propose a novel approach for fine-grained test case minimization. Our analysis is based on the inference of a test suite model that enables automated test reorganization within test cases. It enables removing redundancies at the test statement level, while preserving the coverage and test assertions of the test suite. We evaluated our approach, implemented in a tool called Testler, on the test suites of 15 open source projects. Our analysis shows that over 4,639 (24%) of the tests in these test suites are partly redundant, with over 11,819 redundant test statements in total. Our results show that Testler removes 43% of the redundant test statements, reducing the number of partly redundant tests by 52%. As a result, test suite execution time is reduced by up to 37% (20% on average), while maintaining the original statement coverage, branch coverage, test assertions, and fault detection capability.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

test minimization, test reduction, test redundancy, test model

ACM Reference Format:
Arash Vahabzadeh, Andrea Stocco, and Ali Mesbah. 2018. Fine-Grained Test Minimization. In *Proceedings of 40th International Conference on Software Engineering (ICSE '18).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3180155.3180203

1 INTRODUCTION

Testing has become a widespread practice to ensure the quality and correctness of software systems. As the production code evolves, new test cases are added and existing ones are modified. Over time, a test suite can accumulate redundant test cases [3, 9, 21], without benefiting the overall test suite coverage or effectiveness.

Test suite minimization —also known as reduction— techniques aim at eliminating redundant test cases from test suites to reduce their maintenance and regression testing costs [35]. Existing reduction techniques, however, use adequacy criteria, such as code coverage, as a guideline to remove *whole* test cases with redundant coverage [48]. A potential drawback of such techniques is the deletion of a test case having similar code coverage as other test cases, but different test statements and assertions. This can severely modify the intended test suite behaviour [35], because how a production method is called and asserted influences the overall test suite effectiveness [50] and fault finding capability [34, 44].

To overcome this issue, an approach targeting more fine-grained redundancies within test statements is needed. However, reorganizing (or refactoring) test cases is a challenging task. Developers use the test suite to check that the behaviour of the system is preserved when production code is changed. In contrast, such a safety net does not exist when a test suite needs to go through internal changes. Moreover, it is not straightforward to manually re-order complex test cases in a way that retains the semantics.

In this paper, we propose a novel approach to fine-grained test minimization, while preserving the original behaviour of the test suite. Our approach analyses test cases at the *statement level* and infers a test model that captures the relationships between test statements and test states. The test model allows fine-grained test re-organizational tasks such as identifying and removing redundant test statements. As opposed to existing test reduction techniques, our fine-grained test minimization technique: (1) models the actual behaviour of the test suite by capturing the production method calls together with their inputs, instead of simply relying on code coverage, (2) removes redundant test statements, instead of removing whole test cases, and (3) preserves all test assertions in a test suite, in addition to maintaining the coverage and fault finding capability.

Our work makes the following main contributions:

- A novel test minimization technique for eliminating fine-grained redundancies in test cases that is safe and accurate.
- A test model capturing test statements and test states, along with their relationships.
- An algorithm for identifying behaviour-preserving refactorings in a test suite. It uses the test model to reorganize tests in a way to minimize redundant test statements.
- An implementation of our approach in a tool called Testler, which supports Java code and JUnit4 and is available [41].
- An empirical evaluation of Testler performed on test suites of 15 open source projects. Our results show, on average, a 43% decrease in the number of redundant statements, a 52% decrease in the number of partly redundant tests, and a test suite execution time reduction by up to 37% (20% on average).

2 A MODEL FOR ANALYZING TESTS

Our *test suite model* that enables our fine-grained test analysis is defined as follows.

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden
2018. ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180203

```
1  @Test
2  public void testAdd() {
3   Complex x = new Complex(3.0, 4.0);
4   Complex y = new Complex(5.0, 6.0);
5   Complex z = x.add(y);
6   assertEquals(8.0, z.getReal(), 1.0e-5);
7  }
8  @Test
9  public void testSubtract() {
10  Complex x = new Complex(3.0, 4.0);
11  Complex y = new Complex(5.0, 6.0);
12  Complex z = x.subtract(y);
13  assertEquals(-2.0, z.getReal(), 1.0e-5);
14 }
15 @Test
16 public void testMultiply() {
17  Complex x = new Complex(3.0, 4.0);
18  Complex y = new Complex(5.0, 6.0);
19  Complex z = x.multiply(y);
20  assertEquals(-9.0, z.getReal(), 1.0e-5);
21 }
22 @Test
23 public void testDivide() {
24  Complex dividend = new Complex(3.0, 4.0);
25  Complex divisor = new Complex(5.0, 6.0);
26  Complex q = dividend.divide(divisor);
27  assertEquals(39.0 / 61.0, q.getReal(), 1.0e-5);
28 }
```

Figure 1: Four tests from Apache Commons Math. Fine-grained redundancies are highlighted.

Definition 1 (**Test Suite Model**). *A Test Suite Model is a directed graph denoted by a triple $\langle r, V, E\rangle$ where V is a set of vertices representing test states, E is a set of directed edges representing test statements and assertions, and r denotes the root of the graph, which is the initial empty state.*

To explain our model, we use the `ComplexTest` class from the Apache Commons Math project [2], one of the experimental subjects used in our evaluation. The test class, shown in Figure 1, consists of four test cases with redundant statements (i.e., the initialization of the objects `x` and `y` of the class `Complex`, at Lines 3-4, 10-11, 17-18, and 24-25). Figure 2 shows how these redundant statements are removed by our approach.

Figure 3 illustrates the test suite model obtained for the class `ComplexTest`. Rectangles denote test states (nodes), whereas round boxes (the annotated edges) depict test statements, and dashed lines represent the compatibility between a test state and a test statement (explained in Section 2.4).

Next, we define the properties of our model, the notions of test statement and test state, as well as their relationships in the model.

2.1 Model Properties

There are a number of properties that our model needs to exhibit. First, the model should capture how the production code methods are called by the test suite. This is important to preserve the behaviour of the test suite after any refactoring activity. Second, the model should capture dependencies at the test statement level to support test reorganization. Since a test statement might have dependencies on previous statements, it is important to know how to safely move test statements within or among tests. Finally, the model should facilitate the discovery and removal of redundancies in test cases.

```
1  @Test
2  public void testAdd_Subtract_Multiply_Divide() {
3   Complex x = new Complex(3.0, 4.0);
4   Complex y = new Complex(5.0, 6.0);
5   Complex z = x.add(y);
6   assertEquals(8.0, z.getReal(), 1.0e-5);
7   Complex z_subtract = x.subtract(y);
8   assertEquals(-2.0, z_subtract.getReal(), 1.0e-5);
9   Complex z_multiply = x.multiply(y);
10  assertEquals(-9.0, z_multiply.getReal(), 1.0e-5);
11  Complex q = x.divide(y);
12  assertEquals(39.0 / 61.0, q.getReal(), 1.0e-5);
13 }
```

Figure 2: Reorganized tests of Figure 1. The redundant test statements are removed and four test cases are merged into one.

2.2 Test Statements

We use test statements as the smallest unit of computation for the test model. Using a more fine-grained unit, such as bytecode operation, would increase the model size and the analysis complexity. On the other hand, using a larger unit, such as blocks of statements, would invalidate our ability to detect and reorganize partly redundant test cases with common test statements. Therefore, we consider each test case as a sequence of *test statements* (hereafter referred to as *st*). For example, each line of the test cases in Figure 1 is a *st*. Assertions are also a particular type of *st*.

A unit test case typically creates a set of variables (e.g., objects) and assigns values to their (member) variables, then it calls the production method under test using those variables as inputs, and finally it asserts the value returned by the method. Our test model needs to capture all these three entities, namely, *variables and their values*, *production method calls*, and *test assertions*.

Definition 2 (**Variable Value**). *The value of a variable* `x` *($Val(x)$) is defined as:*

$$Val(x) = \begin{cases} primitive\_value & : Type(x) \in P \\ \{(x_i, Val(x_i)) | x_i \in Fields(x)\} & : Type(x) \notin P \end{cases}$$

$Type(x)$ denotes the type of the variable `x`, P is the set of all primitive types, and $Fields(x)$ denotes the set of all member variables of the object `x`. If the variable `x` is a primitive type, $Val(x)$ is the primitive value. Otherwise, if the variable `x` is an object, its value is a set of $(x_i, Val(x_i))$ pairs where x_i is the name of ith member variable in `x`; in Java this includes `private`, `protected` and `public` member variables of the object. For example, in Figure 1 the value of the object `x` at Line 3 is $Val(x) = \{(Complex.r, 3.0), (Complex.i, 4.0)\}$ given that the `Complex` class has two member variables of type `double` named `r` and `i`.

In order to preserve the test suite semantics, we analyze it to capture its externally observable behaviour. We refer to methods in the production code that are under test as *production methods*. The external behaviour of a test suite can be hence modelled by capturing all the production methods called along with their inputs.

Definition 3 (**Production Method Calls (PMC)**). *The Production Method Calls of a test statement st ($PMC(st)$) is a set of $(MethodName_i, InputSet_i)$ pairs, in which $MethodName_i$ is the qualified name of the called production method, and $InputSet_i$ is the*

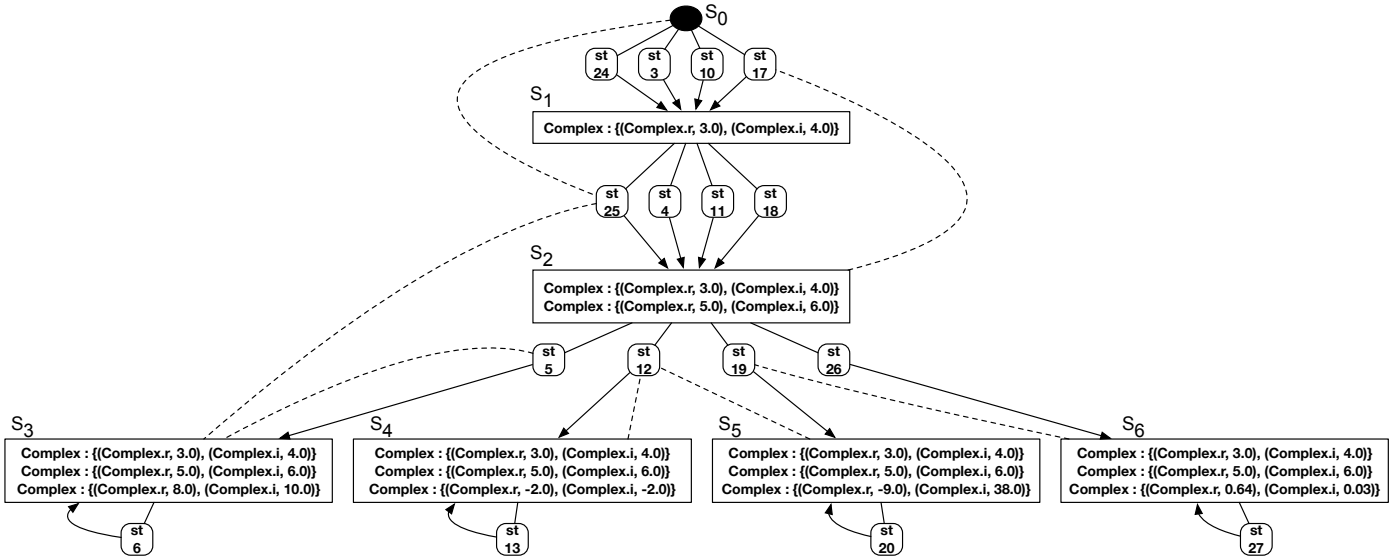

Figure 3: Extracted partial model for the running example. Nodes (rectangles) represent test states and directed edges are test statements. Each edge is labelled with all the information about the test statement; here we use line numbers to simplify the graph. Dashed lines represent state compatibility relations between test states and test statements.

ordered set of $(Type(x_j), Val(x_j))$ *pairs for each input variable* x_j *of the method starting with* `this` *object for non-static member functions.*

For example, in Figure 1, the *PMC* for the test statement at Line 5 is $\{(Complex.add, [(Type(x), Val(x)), (Type(y), Val(y))])\}$ since, as part of the test statement execution, the method `Complex.add` is called with the two inputs `x` and `y`. The *PMC* of a test statement that does not call any production methods is the empty set.

Further, our model needs to accommodate the ability of moving test statements from a source position in one test case to a destination position in another test case. In order to preserve the test's behaviour and avoid undesirable side effects, we need to keep track of the data and definition dependencies of the test statements [26]. Indeed, if we know which variables have been used as part of the execution of a test statement, we can determine whether it can be safely moved to another destination position in the test suite.

Concerning *data dependency*, we save the variables that are used by the test statements and by the assertions, defined as follows.

Definition 4 (**Used Variables of Test Statements (UVS)**). *The Used Variables of a test statement* st_j *(*$UVS(st_j)$*) is a set of* $(Type(x_i), Val(x_i))$ *pairs where each variable* x_i *is used in the execution of* st_j*.*

For example, the UVS of the statement at Line 5 of Figure 1 contains $\{(Type(x), Value(x)), (Type(y), Value(y))\}$.

For assertions, we also need to keep track of the method calls that create the value of the variables. Since assertions check the output of particular production method calls, we need to capture this information as part of our test model. For example, at Line 6 of Figure 1, the assertion checks the output of the method `add` with specific inputs. It is possible to retrieve the whole chain of method calls that the assertion evaluates as following: the assertion checks the output of the `add` method, which uses the output of two constructor calls of the `Complex` class.

Definition 5 (**Used Variables of Assertions (UVA)**). *The Used Variables of an assertion* $asrt$*,* $UVA(asrt)$*, is a set of* $(Type(x_i), Val(x_i), Meth(x_i))$ *tuples where as part of the assertion execution the variable* x_i *is used and* $Meth(x_i)$ *is the PMC of the test statement that assigns the value of* x_i*.*

For example, in Figure 1, the UVA at Line 6 is $\{(Type(z), Value(z), Meth(z))\}$, where $Meth(z) = \{(Complex.add, [(Type(x), Val(x)), (Type(y), Val(y))])\}$ since variable `z` is being used as part of the execution of the assertion and its value is created by the `add` production method call (*PMC* of the test statement at Line 5).

In addition to data dependencies, a test statement can have *definition dependencies* on previous test statements. For example, at Line 26 of Figure 1, the test statement does not depend on the value of the variable `q`. Thus, given that the data dependency of the test statement is satisfied, it is possible to replace the variable `q` with any other variable of type `Complex`. In order to be executed, the test statement needs three variables of the type `Complex` defined in the previous test statements, in addition to its data dependencies. The Defined Variables (*DV*) set of a test statement captures this definition dependency.

Definition 6 (**Defined Variables (DV)**). *The Defined Variables of a test statement* st*,* $DV(st)$*, is the set of the variable types that are referenced in* st*, which need to be defined before the execution of* st*.*

Note that for the defined variables, we abstract away the actual values by focusing on the types. Consider Figure 1 again. The DV set at Line 26 is $\{Complex, Complex, Complex\}$ since the variables `dividend`, `divisor` and `q` need to be previously defined to satisfy the definition dependency of the statement.

2.3 Test States

To perform data and definition dependency analysis, we maintain a *test state* at each test statement.

Definition 7 (**Test State**). *A Test State encompasses information pertaining to the defined variables, their values, and the PMC that created them at a specific test statement in the test case. Formally, the Test State (S_j) is a set of $(Type(x_i), Val(x_i), Meth(x_i))$ tuples for each variable x_i referable from jth test statement in the test case.*

In the Java programming language and JUnit testing framework, the test state includes information about local variables, static field of loaded classes, and member variables of the test class. For example, in Figure 1, the test state before the execution of Line 5 is $\{(Type(x), Value(x), Meth(x)), (Type(y), Value(y), Meth(y))\}$, since the two variables `x` and `y` are referable at Line 5. Following our model, $Meth(x) = \{(Complex, \{(double, 3.0), (double, 4.0)\})\}$ since `x` is created by the *PMC* `Complex` (i.e., the constructor) with two input values 3.0 and 4.0 of type `double`.

2.4 Test State Compatibility

It is possible to move a test statement st to a destination position d only if the test state at d is *compatible* with st.

Definition 8 (**Compatible State**). *A test state is compatible with a test statement if it satisfies the test statement data and definition dependencies. In this case, the test statement can be executed on the test state while preserving its behaviour. Formally, a test state (S_i) is compatible with a test statement (st_j) iff its used variables ($UVS(st_j)$) and defined variables ($DV(st_j)$) are subsets of the test state ($(UVS(st_j) \subset S_i) \wedge (DV(st_j) \subset Def(S_i))$). $Def(S_i)$ denotes the set of defined variables in the test state S_i.*

The compatibility relation for assertions is defined similarly. In Figure 3 dashed lines represent state compatibility relations between test states and test statements. For example, the statement of Line 12 ($st12$) is compatible with the states S_2, S_3, S_4, S_5, and S_6 since the read variable set at Line 12 $\{(Complex, \{(Complex.r, 3.0), (Complex.i, 4.0)\}), (Complex, \{(Complex.r, 5.0), (Complex.i, 6.0)\})\}$ is a subset of these test states. Note that we have illustrated only a subset of all compatibility relations to avoid a tangled graph.

With the notion of compatible states, we can determine possible valid reorganizations of test statements in test cases. For example, we can relocate the test statement of Line 12 ($st12$) in Figure 3 to any location after the states S_3, S_4, S_5, or S_6.

2.5 Equivalent and Redundant Test Statements

To detect redundancies in the test suite, we look at the external behaviour of each test statement to identify those that have identical behaviour. These *equivalent test statements* are identical as far as testing the production code is concerned.

Definition 9 (**Equivalent Test Statements**). *Equivalent Test Statements are the sets of test statements that have the same set of production method calls (PMC).*

To preserve the coverage of the production code, we need to execute (at least) one of the test statements in each set of equivalent test statements. All remaining equivalent test statements are *redundant test statements* and hence can be removed.

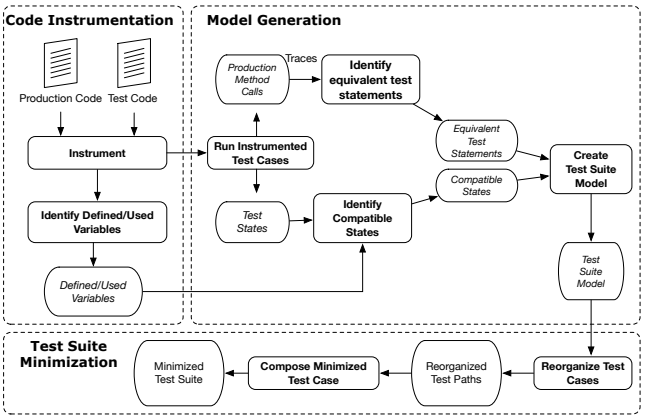

Figure 4: Overview of our approach.

For example, concerning Figure 1, the set of equivalent test statements is $\{\{st_3, st_{10}, st_{17}, st_{24}\}, \{st_4, st_{11}, st_{18}, st_{25}\}, \{st_5\}, \{st_6\}, \{st_{12}\}, \{st_{13}\}, \{st_{19}\}, \{st_{20}\}, \{st_{26}\}, \{st_{27}\}\}$. In this case, in order to maintain the coverage of the test, we only need to execute one of the test statements from the set $\{st_3, st_{10}, st_{17}, st_{24}\}$, one from the set $\{st_4, st_{11}, st_{18}, st_{25}\}$, as well as the sets with only one member.

Definition 10 (**Partly Redundant Tests**). *Test cases that have one or more redundant test statements are called partly redundant tests.*

3 APPROACH

We now describe our approach for creating the model of a test suite and reducing the redundancies given the model. Figure 4 depicts our overall approach.

3.1 Code Instrumentation

To capture the test state at each test statement level, we store the type and value of all referable variables through code instrumentation. We probe local variables, member variables of the test class, and static fields of loaded classes. To capture production method calls (*PMC*s), we instrument the production code to log the entry point, input values (including the `this` object for non-static methods), and exit points of each method. To keep track of production methods that are called by each test statement, we produce a separate call stack log for each test statement of every test case. These call stacks allow us to obtain the PMC sets directly called from test statements.

Used and Defined Variables. For each method invocation, we assume that all the input variables and all of their properties are *used* as part of the test statement execution. This also includes all referable variables, such as static variables and member variables of an object that are part of a method invocation on the object. For example, at Line 6 of Figure 1, we assume that all of the properties of the variable `z` (i.e. `z.r` and `z.i`) will be used as part of the test statement execution (even if `z.i` is not actually used). This is a conservative assumption that prioritizes the precision of detecting compatible states for test statements. In this case, our recall might be adversely influenced since we might miss some compatibility

relations, but the precision is enforced because the relations we detect are correct. To compute the *defined* variables set (see Definition 6), we check for the type of the variables that are referenced in the test statement.

3.2 Model Generation

Equivalent Test Statements. After the instrumentation phase, we execute the instrumented test cases against the instrumented production code, and use the traces to compute sets of equivalent test statements (i.e., having the same *PMC*).

Compatible States. To compute compatible states for a test statement s, we check the states in which the variables used in s have the same values. We also make sure that the test states satisfy the definition dependency (see Section 2.4). For assertions, additionally, we look for the *PMC* that defined the most recent value for the used variables. We require that the direct method calls and their inputs that an assertion checks remain the same to preserve test assertion coverage [37, 50].

With the information about equivalent test statements and compatible states, we create a graph representing our test suite model.

3.3 Test Suite Minimization

We use the inferred model to identify and remove redundant test statements. For example, by reorganizing the four test cases of Figure 1, we create the minimized test case shown in Figure 2, which has the same coverage and assertions, but with **six less statements**.

To maintain the test suite coverage, we basically need to call each production method once. Each test case in our test model is a path starting from the initial state. For example, in Figure 3, the test `testAdd` is the path $\langle st_3, st_4, st_5, st_6 \rangle$ in which st_i is ith test statement (i.e., edge). Thus, to maintain the test suite coverage, we need to find a set of paths, starting from the initial state, that visits at least one test statement from each set of equivalent test statements. To find such paths, we propose a greedy algorithm.

Test Reorganization Algorithm. Algorithm 1 shows our test reorganization algorithm. The intuition is to extend a path among the compatible states of a test statement to cover as many unique test statements and assertions as possible. To do so, we maintain a set of equivalent test statements and assertions that we need to cover (`uncoveredEqStmts`).

Starting from the initial state, we find the shortest path to the nearest test statement that is still uncovered (Line 7). We repeat this procedure from that node until path extension covering new equivalent test statements or assertions is no longer possible (inner loop Lines 12–20). This event occurs when all the equivalent test statements and all the assertions in the model have been covered. Otherwise, if there exist equivalent test statements or assertions which are still uncovered, we start from the initial state and repeat the procedure again (outer loop Lines 4–22). To find the shortest path, we use a variant of the best-fit search algorithm that also maintains the running state [36]. We track the test state at each point in the graph by maintaining a running state (`runningState`). This is essentially the test state that is computed dynamically at each point in the graph. We update the running state at each iteration of the algorithm (Line 9 and Line 15).

Algorithm 1: Test Suite Reorganization

```
Input: uncoveredEqStmts: set of uncovered equivalent test statements
Input: G: the test suite model
Output: paths: set of paths that visits at least one test statement in each set of
        equivalent test statements, and all assertions while minimizing number of test
        statements visited
1  REORGANIZE(uncoveredEqStmts, G)
2  begin
3      paths ← ∅
4      do
5          first ← G.get(init)
6          runningState ← ∅
7          first, path ← PATHTONEARESTUNCOVERED(G, first, uncoveredEqStmts,
             runningState)
8          foreach st_i ∈ path do
9              runningState ← APPLY(st_i, runningState)
10         end
11         frontier ← first
12         while frontier ≠ null do
13             frontier, newpath ← PATHTONEARESTUNCOVERED(G, frontier,
                 uncoveredEqStmts, runningState)
14             foreach st_i ∈ newpath do
15                 runningState ← APPLY(st_i, runningState)
16             end
17             updateGraph(G, frontier, runningState)
18             markAsCovered(frontier, uncoveredEqStmts)
19             path.add(newpath)
20         end
21         paths.add(path)
22     while first ≠ null
23 end
```

Each test statement operates on a compatible test state and transforms the running state to another test state. Let S_i and S_{i+1} be the test states before and after the execution of the test statement st_i. The function $apply(st_i, S_i) = S_{i+1}$ applies the effect of executing the test statement st_i on the test state S_i and gives us the changed test state S_{i+1}. Thus, we know the test state *before* and *after* the execution of each test statement in the original test execution. This enables us to compute the effects of running the test statement on each of its compatible states. Basically, we need to update the value of the used variables of the compatible test state to the values of variables in S_{i+1}. If the state S_j is compatible with the test statement st_i, then $apply(st_i, S_j) = updateValues(USV(st_i), S_{i+1}, S_j)$. We assume that the test statement could potentially change all of its used variables. We update the graph to include any new compatibility edge based on the computed running state (Line 17). To this aim, we compare the UVS and DV of test statements with the computed running state to identify the compatible test statements. The algorithm returns a set of paths (each corresponding to a reorganized/merged test case) that cover each set of equivalent test statements and all the assertions. For example, for the test suite model of Figure 3, our algorithm returns the path $\langle st_3, st_4, st_5, st_6, st_{12}, st_{13}, st_{19}, st_{20}, st_{26}, st_{27} \rangle$, which corresponds to the merged test case of Figure 2.

Composing Minimized Test Cases. Algorithm 1 gives us a set of paths that minimizes the number of test statements executed, while preserving the test suite coverage. Despite the state compatibility, however, problems may arise when moving test statements, because of variable naming and cast issues. For example, in `testDivide` of Figure 1, variables `x` and `y` have names `dividend` and `divisor`, respectively. Therefore, to generate a compilable reorganized test case, we need to rename these variables. Test statements can also define variables that have previously been defined with the same

Algorithm 2: Test Case Composition

```
   Input: statementsPath: ordered list of statements in the composed test case
   Output: statementsPath: renamed list of statements in the composed test case
 1 RenameStatements(statementsPath)
 2 begin
 3     stateBiMap⟨value, Set⟨name, type⟩⟩ ← ∅
 4     foreach stmt ∈ statementsPath do
 5         mapPreq⟨name, value⟩ ← getNameValuePreqVarsInStatement(stmt)
 6         renameMap⟨oldName, newName⟩ ← ∅
 7         castMap⟨varName, oldType, newType⟩ ← ∅
 8         foreach (varName, varValue) ∈ mapPreq do
 9             varNamesInState ← stateBiMap[varValue]
10             if varName ≠ varNamesInState then
11                 renameMap[varName] ← generateName(varNamesInState)
12             end
13             castMap ← checkForTypes(stateBiMap, stmt)
14         end
15         leftHandSideVars⟨Name⟩ ← getVarsInLeftHandSide(stmt)
16         foreach varName ∈ mapPreq do
17             varNameInState ← stateBiMap[varName]
18             if varNameInState ≠ null then
19                 renameMap[varName] ← generateName(varName)
20             end
21         end
22         stmt ← renameStatement(stmt, renameMap, castMap)
23         updateStateMap(stateBiMap, stmt)
24     end
25 end
```

name. For example, in the reorganized test case, the variable z was already defined in three out of four test cases. Thus, we need to rename such variable definitions to avoid name clashes. Further, test statements can use member variables and member functions of the source test class; thus we also need to include those in the destination test class (note that this is not needed when test statements relocate within a test class). At last, due to the polymorphism in object oriented programs, we might need to cast a variable to its sub or super class, if the static type of two variables having the same value differ in the source and the destination state.

Algorithm 2 shows the pseudocode of the algorithm for composing reorganized test paths. To maintain the state, we use a bidirectional map from variable values to variable names and their types. As we go through the test statements in the reorganized test case path, we check, for each test statement, if we have the value for each variable in the test statement. If such a value exists, but has a different name in the state, we rename the variable in the test statement to the name of the variable in the state (Lines 8–15). If the type of the variable is different, we cast the variable to the destination type. If there are name duplicates, we rename them (Lines 16–22). Finally, we update the bidirectional state map with the changed values from test statement execution (Line 23).

3.4 Preserving Test Suite Behaviour

Assume that we reorganize a set of test cases x into the reorganized set of test cases y, we show that y preserves the fault revealing behaviour of x. $PMC(x)$ denotes the set of production method calls that the set of test cases x call with their inputs. Since $PMC(x) = PMC(y)$, each production method m_i that is called as part of the execution of x, will be called with the same inputs in y. Hence, both the coverage and the **implicit oracles** of x are preserved. We also retain the **explicit oracles** of x, because our approach keeps all the corresponding test assertions in the reorganized test cases. Assume that in x, assertion as_i checks the return value of the production method m_i with the input in_i. Let as_j be the same assertion as_i that is included in y. Since $UVA(as_i) = UVA(as_j)$, assertion as_j will check the return value of m_i with the same input in_i. If a fault f in m_i affects the return value of $m_i(in_i)$ and is detectable by as_i, it is also detectable by as_j. □

3.5 Implementation

We implemented our approach in a tool called Testler, which is publicly available [41]. The tool is written in Java. It supports analyzing Java programs with JUnit4 tests. However, our overall approach is general and applicable to other programming languages and testing frameworks. Testler takes as input the path to a Java project. It instruments the test and production code, and runs the instrumented test code against the instrumented production code to obtain traces. Then, Testler uses the traces to create a test model, detects and reorganizes partly redundant test cases, and generates a new minimized test suite.

4 EVALUATION

To assess the real-world relevance and efficacy of our approach, we address the following research questions:

RQ_1 (prevalence): How prevalent are partly redundant tests in practice?

RQ_2 (reduction): What is the redundancy reduction, in terms of tests and test statements, achievable with Testler?

RQ_3 (execution time): What is the reduction in the execution time of the test suites reorganized by Testler?

RQ_4 (code coverage and fault detection): Does Testler preserve the code coverage and fault detection of the test suite?

RQ_5 (performance): What is the runtime of Testler?

4.1 Subject Systems

We include 15 subject systems in our study. Our selection criteria for the subject systems was that they should use the Maven build system and to have executable JUnit4 test cases (as required by our tool's implementation), to be of various sizes in terms of lines of code, and to span different domains. We ran Testler to infer the test models on the test suites of the subject systems. For projects that come with different modules, we repeated the analysis on a per-module basis. Table 1 provides characteristics of the subject systems, including their names, lines of Java production code counted using `cloc` [8], lines of test code, number of test cases, as well as number and percentage of partly redundant tests.

4.2 Procedure and Results

Prevalence (RQ_1). To assess the prevalence of fine-grained redundancies, we measured the number of test cases that have at least one common equivalent test statement with another test case, which we call **partly redundant tests**. To do so, we inferred the test models by running Testler on the test suites of the subject systems. For each subject, we analyzed the inferred test suite model to identify classes of equivalent test statements (see Section 2.5).

The fifth and sixth columns of Table 1 show the number and percentage of partly redundant tests in the subject systems. Our results show that 4,639 (24%) out of the total number of 19,350 test

Table 1: Subject systems and their characteristics

	Lines of Code (K)		Test Cases		
	Prod.	**Test**	**#**	**Par. Redundant**	**%**
Collections	12.3	20.3	459	110	24
Math	45.2	59.1	3,990	1,354	34
Lang	26.6	41.6	2,344	604	26
Email	2.7	3.0	141	21	15
AssertJ	6.4	24.7	4,620	947	20
CheckStyle	16.6	27.0	1,865	164	9
PMD	42.2	14.3	465	192	41
LambdaJ	3.6	4.8	260	82	32
Java-library	24.3	12.2	611	86	14
JFreeChart	98.5	17.3	187	111	59
Accumulo	192.9	13.8	987	289	29
XmlSecurity	40.0	32.3	941	190	20
Crunch	24.0	7.2	452	78	17
Tika	60.5	17.1	553	75	14
Log4J	46.2	25.0	1,475	336	23
Total/Average	642.0	319.6	19,350	4,639	24

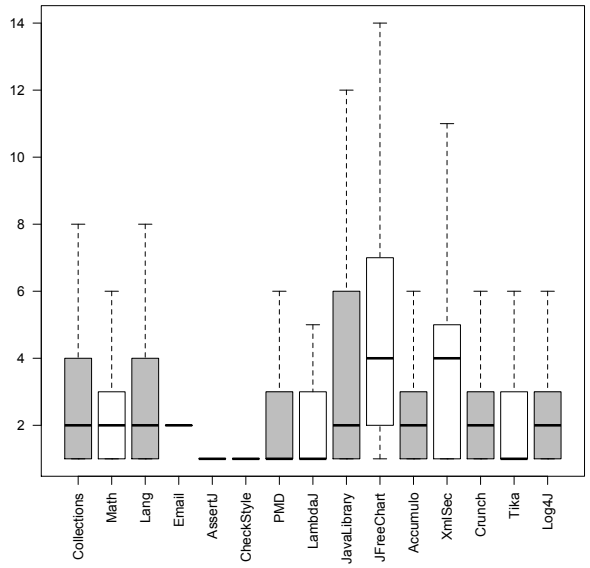

Figure 5: Distribution of redundant test statements per test case in each subject system.

cases in the test suites of our subject systems are partly redundant. Redundancies on individual subject systems range from 9% (CheckStyle) to 59% (JFreeChart).

Figure 5 shows boxplots about the distribution of redundant test statements in each subject system. We observe that most of the tests of the studied subjects share between 1–7 common test statements. This empirical data motivates further the need for an approach capable of detecting fine-grained redundancies in tests.

Reduction (RQ_2). To assess the efficacy of our approach in reducing test statement redundancies, we ran Testler on the subject systems. Testler reorganizes redundant test cases to avoid repeated production method calls, thus reducing the number of redundant test statements. The first macro-column of Table 2 (Redundancy) presents the number of partly redundant **test cases** *before* running Testler, the subset *reduced* by Testler, and the reduction percentage. Further, the table shows the number of partly redundant **test statements** *before* running Testler, the subset *reduced* by Testler, and the achieved reduction percentage.

On average, Testler was able to remove 43% of the redundant test statements, which also resulted in a reduction of 52% in the number of test cases. Across all test suites, 4,639 partly-redundant test cases were reorganized into 2,236 (4,639-2,403) tests, performed by removing 5,041 redundant test statements. XmlSecurity is our best case with 82% test reduction, obtained by discarding 98% redundant test statements. The lowest reduction was obtained for AssertJ (27%), by removing 40% redundant test statements. All test cases that were reorganized passed successfully, with no errors or failures. Our results confirm the design choice of our algorithm (see Section 3.1), which prioritizes the precision of redundancy detection over the detriment of the recall. We discuss why Testler cannot eliminate all the partial redundancies in Section 5.

Execution time (RQ_3). To assess the effects of reducing redundancy on test suite execution time, we measure the execution time of the test suites *before* and *after* the reorganization by Testler. To mitigate the variability effect of non-deterministic tests, we performed the measurements 10 times and we report the averages.

The second macro-column of Table 2 (Execution Time) shows the execution time of the original test suite, the execution time of the reorganized test suite, and the percentage of reduction. Percentage-wise, Testler was able to reduce test execution time across all test suites by about 20%, with the lowest being Collections and PMD (5%) and the highest LambdaJ (37%). For nine subjects (Collections, Lang, Email, AssertJ, CheckStyle, PMD, LambdaJ, Java-library, JFreeChart), the test suites ran in less than one minute. The absolute execution time reductions for those "fast" test suites range between 0.211 s of JFreeChart (-10%) and 5.326 s of CheckStyle (-20%). The other six test suites (Math, Accumulo, XMLSecurity, Crunch, Tika, Log4J) have executions that range between 1.4 and 4.4 minutes. For those "slower" test suites, the absolute execution time reductions range between 10.305 s of Math (-12%) and 63.644 s of Log4J (-24%).

Code Coverage and Fault Detection (RQ_4). We measured the statement and branch coverage for each test suite before and after reorganization, using `EclEmma` [10]. The third macro-column of Table 2 (Code Coverage) shows these coverage numbers.

To measure fault detection rate, we mutated each subject system using `PIT` [32] against the test suite before and after our refactoring. The fourth macro-column of Table 2 (Fault Detection) shows the mutation scores obtained. We also counted the number of test assertions in the original and the reorganized test suites, which remained the same in all subject systems.

Overall, our results show that (1) test assertions remain intact, (2) the statement and branch coverage are preserved, and (3) the fault detection rate is unchanged, for all the subject systems.

Performance (RQ_5). To assess the performance of running Testler, we measured the execution time on a macOS machine, equipped

Table 2: Results

	Redundancy						Execution Time			Code Coverage				Fault Detection		Analysis Time
	Par. Red. Tests			Red. Test Stmts			Seconds		%	Stmt		Branch		Mut. Score		CI/MG/TR (Tot)
	Before (#)	Reduced (#)	Reduction (%)	Before (#)	Reduced (#)	Reduction (%)	Original	Reorganized	Reduction	Before (%)	Reduced (%)	Before (%)	Reduced (%)	Before (%)	Reduced (%)	Time (seconds)
Collections	110	52	47	210	165	79	17.359	16.440	5	84.7	84.7	72.3	72.3	40.0	40.0	71/70/183 (324)
Math	1,354	946	70	4,052	1,623	40	83.109	72.804	12	92.7	92.7	85.7	85.7	79.0	79.0	101/401/1103 (1,605)
Lang	604	335	55	1,533	563	37	11.053	10.376	6	92.9	92.9	57.5	57.5	66.0	66.0	38/43/372 (453)
Email	21	10	48	38	12	32	7.143	5.933	17	54.6	54.6	50.9	50.9	48.0	48.0	10/12/3 (25)
AssertJ	947	253	27	516	207	40	3.389	2.794	18	95.6	95.6	92.2	92.2	63.0	63.0	244/37/383 (664)
CheckStyle	164	77	47	258	122	47	26.361	21.035	20	95.4	95.4	96.8	96.8	17.0	17.0	172/625/216 (1,013)
PMD	192	122	64	346	203	59	7.550	7.145	5	90.1	90.1	52.8	52.8	25.0	25.0	367/110/32 (508)
LambdaJ	82	42	51	109	55	50	1.468	0.924	37	85.2	85.2	52.2	52.2	85.0	85.0	12/6/13 (31)
Java-library	86	37	43	338	71	21	2.062	1.851	10	96.0	96.0	60.7	60.7	54.0	54.0	58/97/215 (370)
JFreeChart	111	56	50	1,657	374	23	2.752	2.485	10	95.5	95.5	47.6	47.6	24.0	24.0	45/52/205 (302)
Accumulo	289	93	32	683	140	20	65.883	50.794	23	94.7	94.7	71.5	71.5	20.0	20.0	58/333/106 (497)
XmlSecurity	190	156	82	1,102	1,079	98	68.590	57.590	16	78.7	78.7	54.0	54.0	68.0	68.0	42/179/142 (363)
Crunch	78	52	67	125	83	66	161.812	137.657	15	94.5	94.5	74.4	74.4	31.0	31.0	27/123/20 (170)
Tika	75	45	60	141	54	38	78.970	54.350	31	88.4	88.4	57.5	57.5	35.0	35.0	50/673/348 (1,071)
Log4J	336	127	38	711	290	41	261.987	198.343	24	80.5	80.5	38.7	38.7	13.0	13.0	48/490/100 (638)
Total/Average	4,639	2,403	52	11,819	5,041	43	799.488	640.521	20	88.0	88.0	64.3	64.3	44.5	44.5	8,034/536

with a 2.3GHz Intel Core i7 and 16 GB of memory. Table 2 (Analysis Time) shows the execution time in seconds, pertaining to the three phases of our algorithm: code instrumentation (CI), model generation (MG), and the test suite reorganization (TR), and total (Tot).

In total, Testler took 8,034 seconds (2.2 hours) to analyze all the 15 subjects test suites; 536 seconds (8.9 minutes) on average. Email and Math were the extreme cases with 25 and 1,605 seconds, respectively. This is not surprising as Testler found the least (21) and the most (1,354) number of redundant test cases in Email and Math, respectively. On average, code instrumentation (CI) required approximately 1.5 minutes, the model generation (MG) 3.6 minutes, and the reorganization (TR) 3.8 minutes. Considering the analysis time in relation with the size of the test suites, Testler was fastest on LambdaJ (0.12 s/test case) and slowest on Tika (1.94 s/test case). On average, across all systems, the runtime is 0.42 seconds (8,034/19,350) for each test case to be analyzed, reorganized, and recomposed.

5 DISCUSSION

In this section, we discuss some of our evaluation findings, tool design decisions and limitations, as well as threats to validity of our study.

Automation. Our results confirm that over time (1) test suites tend to accumulate a considerable number of partly redundant tests, and (2) fine-grained redundancies may pertain to test statements in tests across the whole test suite, and (3) our technique can safely remove nearly half of these redundancies automatically. Manually finding and reorganizing redundant tests to create a reduced test suite that still preserves the coverage could be in fact quite challenging. Our own experience in reorganizing tests, which we were required to do while devising Testler, corroborates the difficulty of the task. For example, for the `ComplexTest` class of Math, which is composed of 139 tests, the authors spent approximately one hour to manually detect the redundant statements within that *single* class, and reorganize the tests in a way to retain the coverage; analyzing and reorganizing the remaining 3,989 test classes would be practically infeasible to perform manually.

Test Suite Evolution. As long as the SUT stays deterministic, Testler preserves all the test states and assertions also during software evolution, because the model is created and updated based on the actual behaviour of the test suite. An important feature of Testler is that the reduced test suite does not need to be updated if there are no changes made to the tests, as part of a commit. Testler saves the inferred test model so that the whole analysis does not need to be repeated every time a new test is added or an existing one is changed. This means that the test model is only partially updated when the software evolves. As discussed in our evaluations (RQ_5), the test analysis process took less than half a second to complete for each test case.

Readability. We are aware that merging tests might affect the overall readability of the test suite. In our experiments (RQ_1), Testler merged test cases having at least one common equivalent test statement with another test case, because we were interested in assessing the maximal capability of our technique at finding and eliminating fine-grained redundancies (at the cost of producing larger merged test classes). As far as readability is concerned, in practice the tester

can customize the granularity of the analysis to optimize the balance between readability and redundancy reduction. That said, the purpose of our work is not to replace the existing test suite. TESTLER keeps both the original and minimized versions of the test suite and maintains links between them. This enables the user to trace back any minimized test case (or its failure) to the test case in the original test suite for better comprehension.

Test Execution Reduction and Overhead. Industrial projects often go through numerous changes daily, and the whole test suite is executed after each change. For instance at Google, on a typical workday, around 40,000 changes are committed [33]. Our results show that by removing fine-grained redundancies, the test execution time can be minimized by 20% on average, which was achieved by removing around 12% of the initial 19,350 test cases in our evaluation. A 20% reduction in test execution time would add up to a substantial test runtime reduction quickly. In our evaluation, the 20% reduction corresponds to about 11 seconds. Hypothetically speaking, if a minimized test suite is executed 40,000 times a day, this would save 122 hours per day. Thus, we believe the startup overhead associated with our analysis can be amortized quickly when considering high commit rates in practice.

Accuracy and Efficacy. Our evaluation results show that TESTLER removed around half of redundant test statements automatically and safely. All the test suites that were reorganized passed successfully and the coverage and fault detection rates were maintained. Therefore, we believe TESTLER is accurate in its minimization task.

Regarding the efficacy of TESTLER, we investigated why *all* redundant test statements were not reduced. We enumerate some of the main reasons next, which pertain to both the inherent nature of JUnit tests and some of our design decisions in developing TESTLER. Our analysis does not have a fully sound handling of the following features.

First, TESTLER cannot reorganize test cases that terminate abruptly as, for instance, those that check that an exception is thrown (e.g., using the JUnit annotation `@Test(expected=SomeException.class)`). In our experiments, a similar behaviour was also simulated by means of the `return` and `fail` statements. Second, although the use of inheritance in test code is debatable [24], most of our subject systems use inheritance in their test code. We chose not to reorganize test cases in test classes that are subclassed by another test class, since in this case the subclass might override some of the test cases and render the reorganized test cases useless. Third, TESTLER does not reorganize parameterized test cases, since in this case, the test case will be run with different inputs and can only be merged with another test case that has exactly the same inputs. Fourth, some tests use custom test runners to run their test cases. For example, in one case, test cases would be retried several times with a custom runner until they would pass. In this scenario, we can only reorganize test classes that have the same custom runner. Fifth, we chose not to reorganize test statements inside conditionals such as `if`s, `for`-loops, and `try-catch`s. Sixth, since we do not store the variable identity as part of our test state (Definition 7), we do not support reorganizing test cases with `assertSame` assertions. Finally, we do not reorganize flaky (non-deterministic) tests, multithreaded tests, or tests that have read/write dependencies on external resources such as files or databases.

Side Effects or Test Dependencies. In our evaluation, we did not witness any issues with side effects of running tests or test dependencies. By creating our test model from the dynamic execution of the test cases, we consider potential side effects and dependencies of test statements and test states, using the state compatibility notion, when reorganizing tests.

Applications. TESTLER can be used by developers to find and minimize redundant test statements. Our test suite model can also be utilized for other test analysis activities, for instance, for test bug [42] or test smell [43] detection. One of the tasks performed during test refactoring is to reorganize test cases to remove eager and lazy test smells [43]; in this case, our model can help with the refactoring task, since it is not straightforward to manually reorganize test cases in a way that preserves the behaviour of the test suite. For instance, our test model can be used to identify small test cases that have common test statements and merge them to remove lazy test smells. Our test model also facilitates going the other direction: it can be used to reorganize large test cases into smaller ones, to remove the eager test smell, while keeping the incurred redundancy at a minimum.

Although we evaluated TESTLER on desktop unit tests, we believe our technique can be even more effective in other domains of testing, such as UI, mobile, or web, where the test execution time may be longer. For example, tests developed for the web domain using frameworks such as Selenium have generally a much higher running cost than unit tests that test low-level Java methods. This is due to the overhead imposed by the interaction with the GUI, the browser and the network latency. In this context, we expect TESTLER to be especially beneficial, because a minimized test suite would contain less calls to the browser's APIs and server-side requests, resulting in a substantial decrease in test execution time.

Relation to Test Reduction Techniques. Traditional test suite minimization seeks to eliminate redundant test cases in order to reduce the number of tests to run [48]. Our technique seeks to eliminate redundant statements in the whole test suite in order to reduce the number of test statements to execute (as our results show, this also results in a reduction of the number of test cases). More importantly, a distinguishing characteristic of our work is that we retain all the test assertions, which are known to be highly correlated with test suite effectiveness [50].

Relation to Test Prioritization/Selection Techniques. Test prioritization seeks to order test cases in such a way that early fault detection is maximized, whereas test selection seeks to identify the test cases that are relevant to the most recent changes [48]. Our approach is orthogonal to those techniques, and can be combined with them. Minimized test cases can still be prioritized. On the other hand, test selection is change-driven. Although we do not target software evolution in this paper, our technique can also be used to apply change-related fine-grained minimization. Also, as the test cases that we merge together have common test statements, it is likely that they test the same component and thus can be selected together as part of a selection strategy.

Parallel/Virtualized Test Execution. A number of techniques address the problem of long running test suites by running tests in virtualized [4], or cloud-based parallel [14, 25] environments. Any of these techniques require complete test independency, a

requirement hard to meet in many real-world test suites including those of our subject systems, which we ran sequentially. However, such techniques can still be applied to test suites that Testler minimizes.

Threats to Validity. Using a limited number of subject systems in our evaluation poses an external validity in terms of generalizability of our results. We tried to mitigate this threat by choosing 15 subject systems with various sizes, domains, and tests, although we need more subject systems to fully address the generalization threat. As reported in a recent empirical study of 20 large Java systems [4], it is often not necessary to reset the application prior to test execution. Our results confirm this finding as we did not witnessed any side effects. However, in some types of applications it might be necessary to reset the application after each test execution session. With respect to reproducibility of our results, the source code of Testler and all subject systems are available online [41], making the evaluation repeatable and our results reproducible.

6 RELATED WORK

There is a large body of work on test suite reduction (minimization) and test selection techniques [5–7, 19, 20, 22, 23, 29, 30, 34, 39, 40, 44, 47, 48], which have been proposed for removing redundant test cases. These techniques use different coverage criteria, such as statement or branch coverage to detect redundancies at the whole test case level. Although it is possible to use coverage criteria in our approach, we chose to preserve the test suite behaviour and find redundant parts of test cases that call the same production methods with exactly the same input. As opposed to these techniques that remove whole test cases, in this paper we tackle the problem of partly redundant tests, and we reorganize test cases by removing only redundant test statements. This allows us to preserve the original test suite's coverage and retain all test assertions, without affecting the fault finding capability.

Many techniques are proposed for regression test selection [11], and use different levels of granularity for tracking dependencies, such as file dependency [15] and class dependency [31], to detect affected test cases, as part of a change to production code. More recently, techniques have been proposed that combine test reduction, test selection, and test prioritization techniques [28, 38, 49]. However, none operates at the fine-grained test statement level.

Our test state representation is closely related to the heap representation used by *Gyori et al.* [18]. They store the portions of the concrete heap accessible from static fields of test classes. On the other hand, in order to support test reorganization, we need to include local variables as well as member variables used in the test class. Our state also includes information about the production method calls that were responsible for the creation of a specific value in the test state.

Some approaches entirely focus on reducing the test suite execution time [4, 16, 27]. For instance, a recent work by *Bell and Kaiser* [4] uses unit test virtualization to reduce the time required for the execution of a test suite, while leaving all the redundant statements. Our work, on the contrary, focuses on identifying and removing redundant test statements, which also results in test execution time reduction of the merged test suites. Moreover, in most of our subjects, there are test dependencies and it is not always possible to reset the system state prior to execution of tests.

Alipour et al. [1] present an approach that reduces a test suite by compromising a certain amount of coverage while preserving the overall fault-finding ability. Our technique reduces the test suite by eliminating redundant test statements while preserving both the original coverage and fault-finding ability.

Fang et al. [12] use assertion fingerprints to detect similar test cases that can be refactored into one single test case. They perform static analysis on test code and, by analyzing the control flow graph, they compute branch count, merge count, and exceptional successor count for each assertion. Based on these attributes, they detect refactoring candidates. Our approach finds refactoring candidates based on common redundant statements in test cases.

Guerra et al. [17] visually represent test cases with a graphical notation to help developers with the refactoring activity. Our approach makes sure that reorganizing test statements in the test suite preserves its behaviour by closely examining the production methods called from test cases.

Xuan et al. [45] split test cases into smaller fragments to enhance dynamic analysis, while *Xuan and Monperrus* [46] perform test purification to improve fault localization. In these papers the goal is to improve dynamic analysis by splitting test cases into smaller units, whereas we aim at reducing redundancies in the test code through test case reorganization.

Fraser and Wotawa [13] merge test cases generated by a model checker by comparing the state of the application for different tests and merging only those for which a common prefix exists. *Devaki et al.* [9] merge web applications GUI test cases to reduce test execution time. They define the state of the program as a combination of browser's DOM and database state. These approaches can only interleave chunks of test steps that result in the same state, whereas our approach is capable of reorganizing and interleaving all valid refactorings *among and across* unit test cases.

7 CONCLUSIONS

In this paper, we proposed a test suite model that facilitates test code analysis at the test statement level. We used the proposed model to present an automated technique and tool, called Testler, for minimizing fine-grained statement redundancies in test cases, while preserving the behaviour, coverage, test assertions, and fault detection of the test suite. We empirically evaluated our technique on 15 subject systems and overall, Testler was able to reduce the number of partly redundant test cases up to 52% and test execution time by up to 37%, while preserving the original test suite coverage and production method call behaviour.

For future work, we plan to investigate the application of fine-grained test analysis on test selection techniques. We also intend to run Testler on more subject systems and extend the tool to support web and mobile test suites.

REFERENCES

[1] Mohammad Amin Alipour, August Shi, Rahul Gopinath, Darko Marinov, and Alex Groce. 2016. Evaluating Non-adequate Test-case Reduction. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 16–26. https://doi.org/10.1145/2970276.2970361

[2] Apache Commons Math 2018. Lightweight, self-contained mathematics and statistics components. http://commons.apache.org/proper/commons-math. (2018).

[3] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley. 2012. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *Proceedings of the 28th International Conference on Software Maintenance (ICSM '12)*. IEEE Press, Piscataway, NJ, 56–65.

[4] Jonathan Bell and Gail Kaiser. 2014. Unit Test Virtualization with VMVM. In *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*. ACM, New York, NY, USA, 550–561. https://doi.org/10.1145/2568225.2568248

[5] J. Black, E. Melachrinoudis, and D. Kaeli. 2004. Bi-criteria models for all-uses test suite reduction. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*. IEEE Computer Society, Washington, DC, USA, 106–115.

[6] T.Y. Chen and M.F. Lau. 1998. A new heuristic for test suite reduction. *Information and Software Technology* 40, 5 (1998), 347–354. https://doi.org/10.1016/S0950-5849(98)00050-0

[7] T.Y. Chen and M.F. Lau. 1998. A simulation study on some heuristics for test suite reduction. *Information and Software Technology* 40, 13 (1998), 777–787. https://doi.org/10.1016/S0950-5849(98)00094-9

[8] Cloc 2018. Counts blank lines, comment lines, and physical lines of source code in many programming languages. https://github.com/AlDanial/cloc. (2018).

[9] Pranavadatta Devaki, Suresh Thummalapenta, Nimit Singhania, and Saurabh Sinha. 2013. Efficient and Flexible GUI Test Execution via Test Merging. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA '13)*. ACM, New York, NY, USA, 34–44. https://doi.org/10.1145/2483760.2483781

[10] EclEmma 2018. Java Code Coverage for Eclipse. http://www.eclemma.org. (2018).

[11] Emelie Engström, Per Runeson, and Mats Skoglund. 2010. A Systematic Review on Regression Test Selection Techniques. *Information and Software Technology* 52, 1 (Jan. 2010), 14–30. https://doi.org/10.1016/j.infsof.2009.07.001

[12] Zheng Felix Fang and Patrick Lam. 2015. Identifying Test Refactoring Candidates with Assertion Fingerprints. In *Proceedings of the Principles and Practices of Programming on The Java Platform (PPPJ '15)*. ACM, New York, NY, USA, 125–137. https://doi.org/10.1145/2807426.2807437

[13] Gordon Fraser and Franz Wotawa. 2007. Redundancy Based Test-suite Reduction. In *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering (FASE '07)*. Springer-Verlag, Berlin, Heidelberg, 291–305. http://dl.acm.org/citation.cfm?id=1759394.1759425

[14] Alessio Gambi, Sebastian Kappler, Johannes Lampel, and Andreas Zeller. 2017. CUT: Automatic Unit Testing in the Cloud. In *Proceedings of the 2017 International Symposium on Software Testing and Analysis (ISSTA '17)*. ACM, New York, NY, USA, 364–367. https://doi.org/10.1145/3092703.3098222

[15] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA '15)*. ACM, New York, NY, USA, 211–222. https://doi.org/10.1145/2771783.2771784

[16] Alex Groce, Mohammed Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. 2014. Cause Reduction for Quick Testing. In *Proceedings of the 7th IEEE International Conference on Software Testing, Verification, and Validation (ICST '14)*. IEEE Computer Society, Washington, DC, USA, 243–252. https://doi.org/10.1109/ICST.2014.37

[17] Eduardo Martins Guerra and Clovis Torres Fernandes. 2007. Refactoring Test Code Safely. In *Proceedings of the International Conference on Software Engineering Advances (ICSEA '07)*. IEEE, Piscataway, NJ, 44–44.

[18] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable Testing: Detecting State-polluting Tests to Prevent Test Dependency. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA '15)*. ACM, New York, NY, USA, 223–233.

[19] Dan Hao, Lu Zhang, Xingxia Wu, Hong Mei, and Gregg Rothermel. 2012. On-demand Test Suite Reduction. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 738–748. http://dl.acm.org/citation.cfm?id=2337223.2337310

[20] Mary Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. 1993. A Methodology for Controlling the Size of a Test Suite. *ACM Transactions on Software Engineering and Methodologies* 2, 3 (July 1993), 270–285. https://doi.org/10.1145/152388.152391

[21] Benedikt Hauptmann, Elmar Juergens, and Volkmar Woinke. 2015. Generating Refactoring Proposals to Remove Clones from Automated System Tests. In *Proceedings of the 23rd International Conference on Program Comprehension (ICPC '15)*. IEEE Press, Piscataway, NJ, USA, 115–124. http://dl.acm.org/citation.cfm?id=2820282.2820298

[22] Dennis Jeffrey and Neelam Gupta. 2007. Improving Fault Detection Capability by Selectively Retaining Test Cases During Test Suite Reduction. *IEEE Trans. Softw. Eng.* 33, 2 (Feb. 2007), 108–123. https://doi.org/10.1109/TSE.2007.18

[23] James A. Jones and Mary Jean Harrold. 2001. Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '01)*. IEEE Computer Society, Washington, DC, USA, 92–101. https://doi.org/10.1109/ICSM.2001.972715

[24] Petri Kainulainen. 2014. Three Reasons Why We Should Not Use Inheritance In Our Tests. https://www.petrikainulainen.net/programming/unit-testing/3-reasons-why-we-should-not-use-inheritance-in-our-tests. (2014).

[25] Sebastian Kappler. 2016. Finding and Breaking Test Dependencies to Speed Up Test Execution. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '16)*. ACM, New York, NY, USA, 1136–1138. https://doi.org/10.1145/2950290.2983974

[26] Ken Kennedy and John R. Allen. 2002. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[27] Shadi Abdul Khalek and Sarfraz Khurshid. 2011. Efficiently Running Test Suites Using Abstract Undo Operations. In *Proceedings of the 2011 IEEE 22Nd International Symposium on Software Reliability Engineering (ISSRE '11)*. IEEE Computer Society, Washington, DC, USA, 110–119. https://doi.org/10.1109/ISSRE.2011.20

[28] Bogdan Korel, Luay Tahat, and Boris Vaysburg. 2002. Model based regression test reduction using dependence analysis. In *Proceedings of the International Conference on Software Maintenance (ICSM '02)*. IEEE Computer Society, Piscataway, NJ, 214–223.

[29] Martina Marré and Antonia Bertolino. 2003. Using Spanning Sets for Coverage Testing. *IEEE Transactions on Software Engineering* 29, 11 (Nov. 2003), 974–984. https://doi.org/10.1109/TSE.2003.1245299

[30] Scott McMaster and Atif Memon. 2008. Call-Stack Coverage for GUI Test Suite Reduction. *IEEE Transactions on Software Engineering* 34, 1 (Jan. 2008), 99–115. https://doi.org/10.1109/TSE.2007.70756

[31] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling Regression Testing to Large Software Systems. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '04/FSE '12)*. ACM, New York, NY, USA, 241–251. https://doi.org/10.1145/1029894.1029928

[32] PIT 2018. PIT Mutation Testing. http://pitest.org. (2018).

[33] Rachel Potvin and Josh Levenberg. 2016. Why Google Stores Billions of Lines of Code in a Single Repository. *Commun. ACM* 59, 7 (jun 2016), 78–87. https://doi.org/10.1145/2854146

[34] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie Hong. 1998. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings of the International Conference on Software Maintenance (ICSM '98)*. IEEE Computer Society, Piscataway, NJ, 34–43. https://doi.org/10.1109/ICSM.1998.738487

[35] Gregg Rothermel, Mary Jean Harrold, Jeffery Von Ronne, and Christie Hong. 2002. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability* 12, 4 (2002), 219–249.

[36] Stuart J. Russell and Peter Norvig. 2003. *Artificial Intelligence: A Modern Approach* (2 ed.). Pearson Education.

[37] David Schuler and Andreas Zeller. 2011. Assessing Oracle Quality with Checked Coverage. In *Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation (ICST '11)*. IEEE Computer Society, Washington, DC, USA, 90–99. https://doi.org/10.1109/ICST.2011.32

[38] August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. 2015. Comparing and Combining Test-suite Reduction and Regression Test Selection. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '15)*. ACM, New York, NY, USA, 237–247. https://doi.org/10.1145/2786805.2786878

[39] Adam M. Smith, Joshua Geiger, Gregory M. Kapfhammer, and Mary Lou Soffa. 2007. Test Suite Reduction and Prioritization with Call Trees. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*. ACM, New York, NY, USA, 539–540. https://doi.org/10.1145/1321631.1321733

[40] Sriraman Tallam and Neelam Gupta. 2005. A Concept Analysis Inspired Greedy Algorithm for Test Suite Minimization. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '05)*. ACM, New York, NY, USA, 35–42. https://doi.org/10.1145/1108792.1108802

[41] Testler 2018. Refactoring Java test suites to eliminate fine-grained redundancies. https://github.com/saltlab/Testler. (2018).

[42] Arash Vahabzadeh, Amin Milani Fard, and Ali Mesbah. 2015. An Empirical Study of Bugs in Test Code. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME '15)*. IEEE Computer Society, Piscataway, NJ, 101–110.

[43] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok. 2002. Refactoring Test Code. In *Extreme Programming Perspectives*. Addison-Wesley, 141–152. http://www.cwi.nl/~arie/papers/xp2001.pdf

[44] W. Eric Wong, Joseph R. Horgan, Saul London, and Aditya P. Mathur. 1995. Effect of Test Set Minimization on Fault Detection Effectiveness. In *Proceedings of the 17th International Conference on Software Engineering (ICSE '95)*. ACM, New York, NY, USA, 41–50. https://doi.org/10.1145/225014.225018

[45] Jifeng Xuan, Benoit Cornu, Matias Martinez, Benoit Baudry, Lionel Seinturier, and Martin Monperrus. 2016. B-Refactoring: Automatic test code refactoring to improve dynamic analysis. *Information and Software Technology* 76 (2016), 65 – 80. https://doi.org/10.1016/j.infsof.2016.04.016

[46] Jifeng Xuan and Martin Monperrus. 2014. Test Case Purification for Improving Fault Localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14)*. ACM, New York, NY, USA, 52–63. https://doi.org/10.1145/2635868.2635906

[47] Xue ying Ma, Zhen feng He, Bin kui Sheng, and Cheng qing Ye. 2005. A genetic algorithm for test-suite reduction. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics (SMC '05)*. IEEE, Piscataway, NJ,

133–139.

[48] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Software Testing, Verification and Reliability* 22, 2 (March 2012), 67–120. http://dx.doi.org/10.1002/stv.430

[49] Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. 2013. Faster Mutation Testing Inspired by Test Prioritization and Reduction. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA '13)*. ACM, New York, NY, USA, 235–245. https://doi.org/10.1145/2483760.2483782

[50] Yucheng Zhang and Ali Mesbah. 2015. Assertions Are Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '15)*. ACM, New York, NY, USA, 214–224. https://doi.org/10.1145/2786805.2786858