Fine-Grained Test Minimization

Arash Vahabzadeh, Andrea Stocco and Ali Mesbah

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

August 9, 2018
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

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.
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.

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:

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

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 $Val(x) = \{(\text{Complex}, r, 3.0), (\text{Complex}, i, 4.0)\}$ 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 $(\text{MethodName}_i, \text{InputSet}_i)$ pairs, in which MethodName$_i$ is the qualified name of the called production method, and InputSet$_i$ is the
ordered set of \((\text{Type}(x_i), \text{Val}(x_i))\) pairs for each input variable \(x_i\) 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 \(\{\text{Complex.add}, [(\text{Type}(x), \text{Val}(x)), (\text{Type}(y), \text{Val}(y))]\}\) since, as part of the test statement execution, the method \text{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 \((\text{Type}(x_i), \text{Val}(x_i))\) 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 \(\{(\text{Type}(x), \text{Val}(x)), (\text{Type}(y), \text{Val}(y))\}\).

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 \text{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 \text{add} method, which uses the output of two constructor calls of the \text{Complex} class.

**Definition 5 (Used Variables of Assertions (UVA)).** The Used Variables of an assertion \(\text{assert}\), \text{UVA}\((\text{assert})\), is a set of \((\text{Type}(x_i), \text{Val}(x_i)), \text{Meth}(x_i))\ tuples where as part of the assertion execution the variable \(x_i\) is used and \text{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 \(\{(\text{Type}(z), \text{Val}(z)), \text{Meth}(z) = \{(\text{Complex.add}, [(\text{Type}(x), \text{Val}(x)), (\text{Type}(y), \text{Val}(y))]\})\}\) 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 \text{Complex}. In order to be executed, the test statement needs three variables of the type \text{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 \(st\), \text{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 \(\{\text{Complex}, \text{Complex}, \text{Complex}\}\) 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 ($\text{Type}(x_i), \text{Val}(x_i), \text{Meth}(x_i)$) tuples for each variable $x_i$ referable from $j$th 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 $\{(\text{Type}(x), \text{Value}(x), \text{Meth}(x))\}$, since the two variables $x$ and $y$ are referable at Line 5. Following our model, $\text{Meth}(x) = \{(\text{Complex}, (\text{double}, 3.0)), (\text{double}, 4.0))\}$ 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 state 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 ($S_j$) is compatible with a test statement ($st_j$) if its used variables $\text{UVS}(st_j)$ and defined variables $\text{DV}(st_j)$ are subsets of the test state $\text{UVS}(st_j) \subseteq S_j \land \text{DV}(st_j) \subseteq \text{Def}(S_j))$. $\text{Def}(S_j)$ denotes the set of defined variables in the test state $S_j$.

The compatibility relation for assertions is defined similarly. In Figure 3, the compatibility between two test statements is defined as $\text{UVS}(st_j) \subseteq S_j \land \text{DV}(st_j) \subseteq \text{Def}(S_j))$. For example, concerning Figure 1, the set of equivalent test statements is $\{(st_1, st_{10}, st_{15}, st_{24}), (st_4, st_{13}, st_{18}, st_{25}), (st_5, st_{19}), (st_{12}), (st_{13}), (st_{19}), (st_{20}), (st_{26}), (st_{27})\}$. 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 $\{st_3, st_{10}, st_{15}, st_{24}\}$, one from the set $\{st_5, st_{11}, st_{18}, st_{25}\}$, 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 (PMCs), 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 Defini-
tion 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 state-
ment $s$, 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 com-
patible 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 $⟨s_3, s_4, s_5, s_6⟩$ in which $s_k$ 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 re-
organization 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).

Each test statement operates on a compatible test state and trans-
forms 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 $s_i$.
The function $apply(s_i, S_i) = S_{i+1}$ applies the effect of executing
the test statement $s_i$ 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
$s_i$, then $apply(s_i, S_j) = updateValues(UVS(s_j), S_{i+1})$. We as-
sume that the test statement could potentially change all of its
used variables. We update the graph to include any new compati-
ble 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 re-
organized/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 $⟨s_3, s_4, s_5, s_6,
s_12, s_{13}, s_{19}, s_{20}, s_{26}, s_{27}⟩$, 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, re-
spectively. 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 1: Test Suite Reorganization**

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>REORGANIZE(uncoveredEqStmts, $G$)</td>
</tr>
<tr>
<td>2</td>
<td>begin</td>
</tr>
<tr>
<td>3</td>
<td>paths ← 0</td>
</tr>
<tr>
<td>4</td>
<td>do</td>
</tr>
<tr>
<td>5</td>
<td>first ← $G$.getInit()</td>
</tr>
<tr>
<td>6</td>
<td>runningState ← ∅</td>
</tr>
<tr>
<td>7</td>
<td>first, path ← PATHTOНЕARESTUNCOVERED($G$, first, uncoveredEqStmts, runningState)</td>
</tr>
<tr>
<td>8</td>
<td>foreach $s_i$ ∈ path do</td>
</tr>
<tr>
<td>9</td>
<td>runningState ← APPLY($s_i$, runningState)</td>
</tr>
<tr>
<td>10</td>
<td>end</td>
</tr>
<tr>
<td>11</td>
<td>frontier ← first</td>
</tr>
<tr>
<td>12</td>
<td>while frontier # null do</td>
</tr>
<tr>
<td>13</td>
<td>frontier, newPath ← PATHTOНЕARESTUNCOVERED($G$, frontier, uncoveredEqStmts, runningState)</td>
</tr>
<tr>
<td>14</td>
<td>foreach $s_i$ ∈ newPath do</td>
</tr>
<tr>
<td>15</td>
<td>runningState ← APPLY($s_i$, runningState)</td>
</tr>
<tr>
<td>16</td>
<td>end</td>
</tr>
<tr>
<td>17</td>
<td>updateGraph($G$, frontier, runningState)</td>
</tr>
<tr>
<td>18</td>
<td>makeUnCovered(frontier, uncoveredEqStmts)</td>
</tr>
<tr>
<td>19</td>
<td>path.add(newPath)</td>
</tr>
<tr>
<td>20</td>
<td>end</td>
</tr>
<tr>
<td>21</td>
<td>paths.add(path)</td>
</tr>
<tr>
<td>22</td>
<td>while first # null</td>
</tr>
<tr>
<td>23</td>
<td>end</td>
</tr>
</tbody>
</table>

---
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 \textit{implicit oracles} of \( x \) are preserved. We also retain the \textit{explicit oracles} of \( x \), because our approach keeps all the corresponding test assertions in the reorganized test cases. Assume that in \( x \), assertion \( a_{s_i} \) checks the return value of the production method \( m_i \) with the input \( i_{n_i} \). Let \( a_{s_j} \) be the same assertion \( a_{s_i} \) that is included in \( y \). Since \( UV(A(s_i)) = UV(A(s_j)) \), assertion \( a_{s_j} \) will check the return value of \( m_i \) with the same input \( i_{n_j} \). If a fault \( f \) in \( m_i \) affects the return value of \( m_i(i_{n_i}) \) and is detectable by \( a_{s_i} \), it is also detectable by \( a_{s_j} \).

### 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:

**RQ1 (prevalence):** How prevalent are partly redundant tests in practice?

**RQ2 (reduction):** What is the redundancy reduction, in terms of tests and test statements, achievable with Testler?

**RQ3 (execution time):** What is the reduction in the execution time of the test suites reorganized by Testler?

**RQ4 (code coverage and fault detection):** Does Testler preserve the code coverage and fault detection of the test suite?

**RQ5 (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 jLoc [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 (RQ1).** 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 cases of the subject systems are partly redundant tests.

<table>
<thead>
<tr>
<th>RQ1: How prevalent are partly redundant tests in practice?</th>
<th>RQ2: What is the redundancy reduction, in terms of tests and test statements, achievable with Testler?</th>
<th>RQ3: What is the reduction in the execution time of the test suites reorganized by Testler?</th>
<th>RQ4: Does Testler preserve the code coverage and fault detection of the test suite?</th>
<th>RQ5: What is the runtime of Testler?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Algorithm 2: Test Case Composition

```plaintext
Algorithm 2: Test Case Composition

<table>
<thead>
<tr>
<th>Algorithm 2: Test Case Composition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> statementsPath: ordered list of statements in the composed test case</td>
</tr>
<tr>
<td><strong>Output:</strong> statementsPath: renamed list of statements in the composed test case</td>
</tr>
<tr>
<td>1 RenameStatements(statem</td>
</tr>
<tr>
<td>2 begin</td>
</tr>
<tr>
<td>3 StateBiMap(value, Set(name, type)) ← Ø</td>
</tr>
<tr>
<td>4 foreach stmt ∈ statementsPath do</td>
</tr>
<tr>
<td>5   mapPreq(stmt, name, value) ← GetInputValuePairVarStmt(stmt)</td>
</tr>
<tr>
<td>6   renameMap(oldName, newName) ← Ø</td>
</tr>
<tr>
<td>7   castMap(varName, oldType, newName) ← Ø</td>
</tr>
<tr>
<td>8   foreach varName ∈ varValue in mapPreq do</td>
</tr>
<tr>
<td>9     varNamesInState ← stateBiMap[varValue]</td>
</tr>
<tr>
<td>10    foreach varName ∈ varNamesInState do</td>
</tr>
<tr>
<td>11      renameMap[varName] ← GenerateName(varNamesInState)</td>
</tr>
<tr>
<td>12     castMap[varName] ← CheckForTypes(stateBiMap, stmt)</td>
</tr>
<tr>
<td>13 end</td>
</tr>
<tr>
<td>14 end</td>
</tr>
<tr>
<td>15 end</td>
</tr>
<tr>
<td>16 foreach varName ∈ mapPreq do</td>
</tr>
<tr>
<td>17     varNamesInState ← stateBiMap[varName]</td>
</tr>
<tr>
<td>18     foreach varName ∈ varNamesInState do</td>
</tr>
<tr>
<td>19       renameMap[varName] ← GenerateName(varNamesInState)</td>
</tr>
<tr>
<td>20     end</td>
</tr>
<tr>
<td>21 end</td>
</tr>
<tr>
<td>22 end</td>
</tr>
<tr>
<td>23 updateStateBiMap(stateBiMap, stmt)</td>
</tr>
<tr>
<td>24 end</td>
</tr>
<tr>
<td>25 end</td>
</tr>
</tbody>
</table>
```
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 (RQ2). 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 (RQ3). 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 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, XMLHttpRequest, 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 (RQ4). We measured the statement and branch coverage for each test suite before and after reorganization, using Ec1Emma [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 (RQ5). To assess the performance of running Testler, we measured the execution time on a macOS machine, equipped

---

**Table 1: Subject systems and their characteristics**

<table>
<thead>
<tr>
<th>System</th>
<th>LOCS (Prod)</th>
<th>LOCST (Test)</th>
<th># Par. Red</th>
<th>Redundancy %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collections</td>
<td>12.3</td>
<td>20.3</td>
<td>459</td>
<td>110</td>
</tr>
<tr>
<td>Math</td>
<td>45.2</td>
<td>59.1</td>
<td>3,990</td>
<td>1,354</td>
</tr>
<tr>
<td>Lang</td>
<td>26.6</td>
<td>41.6</td>
<td>2,344</td>
<td>604</td>
</tr>
<tr>
<td>Email</td>
<td>2.7</td>
<td>3.0</td>
<td>141</td>
<td>21</td>
</tr>
<tr>
<td>AssertJ</td>
<td>6.4</td>
<td>24.7</td>
<td>4,620</td>
<td>947</td>
</tr>
<tr>
<td>CheckStyle</td>
<td>16.6</td>
<td>27.0</td>
<td>1,865</td>
<td>274</td>
</tr>
<tr>
<td>PMD</td>
<td>42.2</td>
<td>14.3</td>
<td>465</td>
<td>192</td>
</tr>
<tr>
<td>LambdaJ</td>
<td>3.6</td>
<td>4.8</td>
<td>260</td>
<td>82</td>
</tr>
<tr>
<td>Java-library</td>
<td>24.3</td>
<td>12.2</td>
<td>611</td>
<td>86</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>98.5</td>
<td>17.3</td>
<td>187</td>
<td>111</td>
</tr>
<tr>
<td>Accumulo</td>
<td>192.9</td>
<td>13.8</td>
<td>987</td>
<td>289</td>
</tr>
<tr>
<td>XmlSecurity</td>
<td>40.0</td>
<td>32.3</td>
<td>941</td>
<td>190</td>
</tr>
<tr>
<td>Crunch</td>
<td>24.0</td>
<td>7.2</td>
<td>452</td>
<td>78</td>
</tr>
<tr>
<td>Tika</td>
<td>60.5</td>
<td>17.1</td>
<td>553</td>
<td>75</td>
</tr>
<tr>
<td>Log4j</td>
<td>46.2</td>
<td>25.0</td>
<td>1,475</td>
<td>336</td>
</tr>
</tbody>
</table>

Total/Average 642.0 319.6 19,350 4,639 24
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.

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.

Table 2: Results

<table>
<thead>
<tr>
<th>Redundancy</th>
<th>Execution Time</th>
<th>Code Coverage</th>
<th>Fault Detection</th>
<th>Analysis Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Par. Red. Tests</td>
<td>Red. Test Stmt</td>
<td>Seconds</td>
<td>%</td>
</tr>
<tr>
<td>Before (%)</td>
<td>Reduced (%)</td>
<td>Before (%)</td>
<td>Reduced (%)</td>
<td>Original</td>
</tr>
<tr>
<td>Collections</td>
<td>110 52 47</td>
<td>210 165 79</td>
<td>17.359</td>
<td>16.440</td>
</tr>
<tr>
<td>Math</td>
<td>1,354 946 70</td>
<td>4,052 1,623 40</td>
<td>83.109</td>
<td>72.804</td>
</tr>
<tr>
<td>Email</td>
<td>604 335 55</td>
<td>1,533 563 37</td>
<td>11.053</td>
<td>10.376</td>
</tr>
<tr>
<td>AssertJ</td>
<td>947 253 27</td>
<td>516 207 40</td>
<td>3.389</td>
<td>2.794</td>
</tr>
<tr>
<td>CheckStyle</td>
<td>164 77 47</td>
<td>258 122 47</td>
<td>26.361</td>
<td>21.035</td>
</tr>
<tr>
<td>PMD</td>
<td>192 122 64</td>
<td>346 203 59</td>
<td>7.550</td>
<td>7.145</td>
</tr>
<tr>
<td>LambdaJ</td>
<td>82 42 51</td>
<td>109 55 50</td>
<td>1.468</td>
<td>0.924</td>
</tr>
<tr>
<td>Java-library</td>
<td>86 37 43</td>
<td>338 71 21</td>
<td>2.062</td>
<td>1.851</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>111 56 50</td>
<td>1,657 374 23</td>
<td>2.752</td>
<td>2.485</td>
</tr>
<tr>
<td>Accumulo</td>
<td>289 93 32</td>
<td>683 140 20</td>
<td>65.883</td>
<td>50.794</td>
</tr>
<tr>
<td>XmlSecurity</td>
<td>190 156 82</td>
<td>1,102 107 98</td>
<td>68.590</td>
<td>57.590</td>
</tr>
<tr>
<td>Crunch</td>
<td>78 52 67</td>
<td>125 83 66</td>
<td>161.812</td>
<td>137.657</td>
</tr>
<tr>
<td>Tika</td>
<td>75 45 60</td>
<td>141 54 38</td>
<td>78.970</td>
<td>54.350</td>
</tr>
<tr>
<td>Log4j</td>
<td>336 127 38</td>
<td>711 290 41</td>
<td>261.987</td>
<td>198.345</td>
</tr>
</tbody>
</table>

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.
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`, `for-loops`, and `try-catches`. 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 witness 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 test cases, 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 code through test case reorganization.

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 refactoring 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


