Removing Redundant Statements in Amplified Test Cases

Wessel Oosterbroek
Delft University of Technology
w.oosterbroek@student.tudelft.nl

Carolin Brandt
Delft University of Technology
c.e.brandt@tudelft.nl

Andy Zaidman
Delft University of Technology
a.e.zaidman@tudelft.nl

***Abstract*—Test amplification generates new tests by modifying existing, manually written tests. Up until now, this process preserves statements that were relevant for the original test case but are no longer needed for the behavior of the new test case. These unnecessary statements impact the readability of the tests in question. As a part of the effort to make amplified test cases more readable, we investigate dynamic slicing, taint analysis and static analysis as approaches to remove redundant statements. We design and evaluate a static analysis approach that we implemented as part of the test amplification tool DSpot. Our empirical evaluation on 274 amplified test cases shows that the implemented approach works well: while being rudimentary, it is able to remove a significant portion of the redundant statements in the amplified test cases. While the removal of the statements themselves is fast, verifying that the tests still work as intended through mutation testing is still resource-intensive.**

I. Introduction

An important aspect of software development is verifying that the created software works as intended. One of the ways to do this is through developer testing, an approach where developers write tests that check parts of the code they engineered [1]. Manually creating and maintaining test suites can take a lot of time and effort [2]. One option to reduce the time it takes to create these test suites is to automatically generate tests, reducing the time developers spend on writing tests. A tool designed to help accomplish this task is DSpot [3]: it *amplifies* test cases by taking an existing test and returning a set of new test cases that improve overall coverage [4]. While the designers of DSpot envision a world where the tool can generate pull requests with test cases ready for developers to merge [5], the amplified test cases prove cumbersome to use: use of vague identifiers, the inclusion of no longer used statements, and assertions that are generally weak. As such, all generated tests have to be thoroughly checked and refactored before a developer can benefit from them [6].

In this paper, we aim to combat redundant statements in amplified test cases. These statements are left over from the existing test case used for amplification, but are not necessary for the behavior of the new test cases. Overall, we investigate:

Main RQ: *How can redundant statements be detected and removed from amplified test cases created by DSpot?*

This research was partially funded by the Dutch science foundation NWO through the Vici "TestShift" grant (No. VI.C.182.032)

To answer this research question we raise two sub-questions:

RQ1: *What redundant statements does DSpot include in the amplified test cases it generates?*

As a first step to identify and find suitable solutions for deleting redundant statements, we conduct an analysis of several test cases created by DSpot in Section II.

With our understanding of how redundant statements are introduced in the amplified test cases, we set out to investigate techniques to detect and remove these redundant statements. In particular, we look into slicing, taint analysis, and static analysis. For each of them, we discuss their advantages and drawbacks, as well the feasibility of implementing them into DSpot. Finally, we settle on a lightweight static analysis approach, and investigate Section III:

RQ2: *How does the implemented solution perform, i.e., how many of the redundant statements are removed?*

In a small-scale study on a variety of projects and tests, we analyze our solution and compare it to the default DSpot implementation. In Section IV we will analyze the results of the conducted study, highlighting that the main concerns for our proposed approach are accuracy and run-time.

II. Redundant Statements

In the context of this paper, redundant statements are statements whose removal does not affect the mutation coverage of a particular test in any way. Important to highlight is that our definition does not include all statements that could be considered unnecessary, e.g., statements that could potentially be combined with other statements for readability are not considered in this paper.

Instead, our focus is on statements that are part of the original test case that formed the input to the DSpot amplification, but that are no longer relevant for the amplified test case. By removing these statements we intend to improve the readability of the test cases in question.

Contrary to redundant statements in the context of a normal program, statements that do not change variables or objects in a test case might still be interesting. Consider a test for a function that when given as input an integer returns the absolute value of that integer. The test might give as input a positive integer, and then verifies that the function returns the exact same integer. If that function call was made in the context of a normal program one could consider calling the function to be redundant, however in the context of this unit test it is not as we are verifying the behavior of the function in question even if the integer itself was not changed.

```java
1 public void booleanAttributesEmptyStringValues2() {
2   Document doc = Jsoup.parse("<div hidden>");
3   Attributes as = doc.body().child(0).attributes();
4   as.get("hidden");
5   Attribute first = as.iterator().next();
6   first.getKey();
7   first.getValue();
8   first.hasDeclaredValue();
9   assertFalse(doc.getAllElements().isEmpty(); }
```

Listing 1: An amplified test case created with DSpot.

```java
public void booleanAttributesEmptyStringValues2() {
  Document doc = Jsoup.parse("<div hidden>");
  assertFalse(doc.getAllElements().isEmpty(); }
```

Listing 2: The test case from Listing 1 without redundant statements.

A. *Types of Redundant Statements in Amplified Test Cases*

To get a better understanding of the types of redundant statements created by DSpot, we manually analyzed 30 amplified test cases from the JSoup Project [7], and found that there are three main types of redundant statements:

a) Declarations of unnecessary variables: The first type refers to variables that are either not used in any statement or variables for which all statements, that involve the object, are not relevant for the test case and are therefore redundant. These statements and the variables themselves could be removed from the test case, the `first` object declared on line 5 in Listing 1 is an example of an unnecessary variable.

b) Elements from old assertions: By default DSpot sometimes keeps elements in from assertions that were part of the original test case. The assertion itself is removed but the call to a method to retrieve a value within the assert statement is not. An example can be found in Listing 1, on line 6 the `first.getKey()` statement originates from an assertion in the original test case. These statements are in our observation, often redundant as they usually do not have any side effects and are only retrieving a value, i.e., they are often getters.

c) Statements with side effects: Last are redundant statements that have side effects, but are not relevant, e.g., changing the surname of a Person object while the assert statements are only concerned with the age of the person.

Answer to RQ1: There are three main types of redundant statements that appear most often in the tests created by DSpot. Declarations of unnecessary variables, elements from old assertions and statements with side effects. We expect that both declarations of unnecessary variables and elements from old assertions should be straightforward to detect and remove, while removing statements that have side effects will be more challenging.

III. Detecting Redundant Statements

In this section, we discuss three methods to detect the redundant statements described in Section II. This section will give insights into what the possible advantages and disadvantages of each method are, in addition to explaining the static analysis approach we contributed to DSpot.

It is important to note that when editing a test case we can verify that the test still performs as expected, something that would not be possible for a regular program. After minimizing, the test case should still compile, run and catch the same number of mutants as the original test case. Verifying that the test case still compiles and catches the same mutants as the original does nevertheless not fully guarantee that it behaves the same way as the original amplified test. There might be differences in the way it runs or even catches mutants. However, DSpot uses PIT mutation testing to select amplified test cases, meaning that if the minimized test still catches the same mutants, they are seen as equally valuable by DSpot.

A. *Dynamic Slicers*

Slicing is a technique to simplify programs, removing parts of a program that have no effect on the statements or variables we are interested in [8]. It has been researched quite extensively and finds main use-cases in debugging, program analysis and re-engineering, and allows ignoring parts of the program that are of no interest [8].

There is a difference between static slicing, with no assumptions about the input a program will receive, and dynamic slicing, in which only one execution path of the program and a specific set of input variables is considered. The result of a dynamic slicer solely concerns the execution of the program, which results in dynamic slices often being much smaller [8].

To our knowledge there is no prior work about removing redundant statements from test cases using program slicing. AlAbwaini et al. talk about removing redundant code using program slicing in the context of a normal program [9]. Their approach consists of using program analysis techniques to find the effective variables, i.e., the variables that influence the outcome of a program, followed by running a slicer on each of these variables looking at which parts of the code affected them. Combining these slices allows for the removal of redundant code from the program. However, this becomes more difficult on larger programs because it becomes harder to find the effective variables [9].

As a test case defines the execution path and input values to the program, we have the opportunity to use a dynamic slicer. A dynamic slicer should be able to identify most statements that do not affect the assert statements in a test case. However, there are no dynamic Java slicers available that support Java 1.8 or above, which is necessary for integration into DSpot and compatibility with modern Java programs. JavaSlicer only supports JDK versions up to 1.7 [10]. An interesting slicer, which was made public too late to be considered in this project, is Slicer4J [11], [12]. Slicer4J is based on a dynamic Android slicer called Mandoline and supports modern

Java applications, which makes it an interesting candidate for potential integration into DSpot.

B. Dynamic Taint Analysis

The goal of dynamic taint analysis is to find out which computations are affected by tainted sources such as user-input [13]. It has a wide variety of use cases and can even be used in test generation [13]. Taint analysis could also be an option to detect redundant statements, by marking all inputs of a test case and seeing which ones affect the assert statements. Bell and Kaiser describe this use case as a method to detect brittle assertions in their paper about Phosphor, a dynamic taint analysis tool for Java [14]. To the best of our knowledge this is the only dynamic taint analysis tool for Java that supports general use cases as well as default JVMs [14]. However, while this is useful to detect redundant input variables and objects, it does not tell much about statements that have side-effects, as we are only looking at which input variables affect the assert statements, but not at how other statements, that use input variables, affect the assert statements.

C. Static Code Analysis

Another option is to use static analysis to look at a test case and try to infer which statements are redundant and which are not. For example, all declarations of variables directly or indirectly used by the assert statement are guaranteed to be needed for the test to still compile and thus can not be redundant. On the other hand, variables that do not get used at all, even not indirectly, by the assert statements are likely to be redundant. An advantage of this approach is that it is fast, taking almost no time to remove the redundant statements.

To detect redundant statements from amplified test cases, we investigate dynamic slicers, taint analysis and static analysis. While this selection is non-exhaustive, it should give an idea of options to detect redundant statements.

D. Algorithm

Our contribution to DSpot consists of a static analysis algorithm that tries to remove as many of the redundant statements as possible. The actual implementation uses some parts from the PitMutantMinimizer by Danglot found in DSpot [3], which tries to remove assertions that do not improve the mutation score of an amplified test.

Our algorithm to delete redundant statements consists of three separate steps, which all take the original amplified test case as input. In each step, the algorithm tries to delete statements from the test case while becoming more conservative in which statements are deleted in each subsequent step. After each step, the resulting test case is syntactically compared to the original amplified test case, if there is no difference between the two, the algorithm returns the original test case. If there is a difference, PIT [15] will be used to verify that the test case still catches the same mutants as before. In the case that it does, the algorithm returns the minimized test case, if it does not, the algorithm continues with the next step. If all steps fail the algorithm will return the original amplified test.

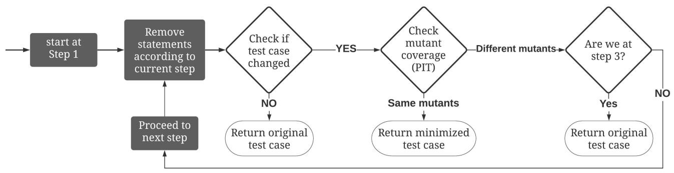
start at Step 1
Remove statements according to current step
Check if test case changed
YES
Check mutant coverage (PIT)
Different mutants
Are we at step 3?
NO
NO
Same mutants
Yes
Proceed to next step
Return original test case
Return minimized test case
Return original test case

Fig. 1: Overview of our approach described in Section III-D

Figure 1 presents a flow chart of this process. The individual steps consist of the following:

- **Step one:** Delete all statements, except for the assertions and statements that are needed to compile.
- **Step two:** Remove all statements that do not interact with the assert statements, where an interaction refers to the statements containing variables that are needed by the assert statements directly or indirectly. Additionally, remove `loops` and variables that only interact with the assert statements when they are declared. The rationale behind this comes from the fact that a lot of the unnecessary variables use needed variables when being declared, e.g., a string is set equal to the name of an object. Listing 3 shows an example on line 4. The `String name` is declared using the `doc` object that is relevant for the test case, however the string `name` itself is not. The same logic applies to `loops`, which we will only consider if relevant objects are used inside the body of said `loop`, thus the `for loop` on line 5 in Listing 3 is removed.
- **Step three:** Remove all the statements that do not directly or indirectly interact with variables used in the assert statements.

```
1 public void exampleTest() {
2   Document doc = Jsoup.parse(in, "UTF-8");
3   Elements templates = doc.body().getElementsByTag(
        "template");
4   String name = doc.nodeName();
5   for (Element template : templates) {
6     boolean equals = name.equals(""); }
7   assertFalse(templates.equals(null)); }
```

Listing 3: An example test containing a for loop and object that only interacts when the assert statement when they are declared.

An example of this process is shown in Listings 4 through 7. Listing 4 presents an amplified test case, which was slightly modified for the purposes of this explanation. In step one, all variables and statements that are not needed for compilation are removed, the resulting test case is shown in Listing 5. However this test case fails as we removed the `a2.setValue(characters)` and `a2.setKey("three")` statements which were needed for the assertions to pass. Listing 6 shows the result of step two, this test case still fails due to the `String key = a2.setKey("three")` statement being removed, as it falls under a variable that only interacts with assertions when declared. Listing 7 shows the final passing test case after step three, only removing the statement `Attribute a1 = new Attribute("one", "")`, this test case compiles and covers the same mutants as the original.

```java
public void hasValue2() {
  String characters = "#{>%";
  Attribute a1 = new Attribute("one", "");
  Attribute a2 = new Attribute("two", null);
  String key =  a2.setKey("three");
  a2.setValue(characters);
  assertEquals("three=\"#{>%\"", a2.toString(); }
```

Listing 4: A (modified) amplified test case created with DSpot.

```java
public void hasValue2() {
  Attribute a2 = new Attribute("two", null);
  assertEquals("three=\"#{>%\"", a2.toString(); }
```

Listing 5: The result after applying step one on the amplified test.

```java
public void hasValue2() {
  String characters = "#{>%";
  Attribute a2 = new Attribute("two", null);
  a2.setValue(characters);
  assertEquals("three=\"#{>%\"", a2.toString(); }
```

Listing 6: The result after applying step two on the amplified test.

```java
public void hasValue2() {
  String characters = "#{>%";
  Attribute a2 = new Attribute("two", null);
  String key = a2.setKey("three");
  a2.setValue(characters);
  assertEquals("three=\"#{>%\"", a2.toString(); }
```

Listing 7: The result after applying step three on the amplified test.

This minimizer was implemented into the DSpot prettifier module, which does not allow for easy removal of elements that were in the assert statements of the original test case without modifying its structure to a larger extent than adding a new minimizer. To remove these statements it is necessary to either provide the original test case using a parameter or to edit the amplification process of DSpot itself to no longer include these statements. Moreover, we are using DSpot to run PIT, meaning that the exact same parameters are used as when amplifying using mutation testing.

Our solution consists of a lightweight static analysis algorithm, which has the advantage of being straightforward to implement compared to dynamic slicing and taint analysis. We expect that it should be able to remove a significant number of redundant statements. We verify the removal through mutation analysis.

IV. Empirical Evaluation

In this section, we will answer research question **RQ2** by evaluating the performance of the implemented algorithm, described in Section III-D. To that end we performed a qualitative as well as a quantitative study. In the qualitative study we focus on what statements get removed or not by manually inspecting 46 minimized test cases. In the quantitative study we investigate the overall performance by measuring the difference in statements before and after minimization. We created 274 amplified tests, which originate from and improve the coverage of 15 test classes from the JSoup, Stream-Lib and Twilio-Java projects [7], [16], [17]. Three of these classes containing 46 tests, from the JSoup project, were manually checked for redundant statements before and after running the minimizer.

To reproduce the results discussed in this paper we have created a replication package [18].

A. *Qualitative Analysis*

We analyzed 46 test cases and found that our approach is able to remove 28% (32 out of 113) of all redundant statements in these tests. Note that these test cases still include elements from old assertions (see Section II-A), as 65 out of the 81 not-removed statements were of this type, we expect removing those statements would significantly increase the number of statements removed. Listing 8 shows a test case where lines 5-7, which are all elements from old assertions, are redundant as they only retrieve values and have no side-effects. Manually analyzing the minimized test cases shows that there are quite a few test cases in which only a few of the redundant statements get removed, while others are minimized as much as possible. This is caused by the algorithm removing all but necessary statements in step one, meaning that if this first minimization step succeeds the test case is likely to be fully minimized.

All redundant statements of the type unnecessary variable were removed, however they represent a small part of all removed redundant statements. While the majority of removed statements consist of the other two types, described in Section II, their removal is not guaranteed and depends on the test structure and usage of declared variables.

```java
1  public void html5() {
2  Attributes a = new Attributes();
3  a.put("Tot", "a&p");
4  a.put("Hello", "There");
5  a.size();
6  a.hasKey("Tot");
7  a.hasKey("Hello");
8  assertEquals(-758045610, a.hashCode(); }
```

Listing 8: Minimized test case with elements from old assertions.

B. *Quantitative Analysis*

Figure 2 shows the results of minimizing 274 amplified tests from 4 projects, showing a reduction in the number of statements in each test case after minimizing. In these results the assert statements themselves are included, thus making the minimum number of statements in each test one. An interesting observation that can be made when looking at Figure 2 is that by far the largest number of statements is removed in step one, in step two a couple statements are removed, while in step three almost no statements are removed. Step three removing few statements is the result of those statements already being removed in either steps one or two. There were a few test cases where the first two steps failed and there were statements that could be removed in step three. While it did not result in any statements being removed in the test cases selected for this study, one might see slightly different results when minimizing other test cases.

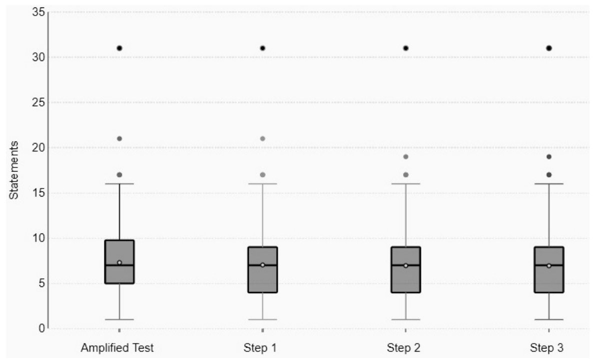

Fig. 2: Box plots showing the number of statements in 274 tests before and after minimizing with each step.

C. *Run-time*

An important thing to note is that removing these redundant statements takes a significant amount of time due to the mutation coverage that has to be determined after changing a test using PIT. While the median number of PIT runs is two and the average lower than two per test, the time spent on calculating mutation coverage accounts for the overwhelming majority of the time spent on minimizing these test cases. Removing the statements themselves only takes a few milliseconds, while it can take up to five minutes to run our complete approach, including the mutation analysis.

Answer to RQ2: Our results show that the implemented approach works well: while being rudimentary it is able to remove a significant portion of the redundant statements in the amplified test cases. Unnecessary variables are always removed, while the removal of statements with side-effects and elements of old assertions depends on the structure of the test case. A problem with removing redundant statements is that using mutation coverage to verify that the tests still work takes a significant amount of time.

D. *Threats to Validity*

1) External validity: The variety of projects and tests selected to evaluate our minimizer on is limited. For one they are all open source projects, with most tests coming from the JSoup project. While we do not expect major differences in the number of removed statements for other projects given that they are likely to have a similar style of unit tests. Moreover, we suspect that the performance of our approach will suffer with more complicated tests. Since step one of the removal process relies on removing all statements in a test case, except for those that are needed to compile, this step might work less well on longer and more complicated test cases. We expect the types of redundant statements to be closely linked to the test amplification of DSpot, which is to our knowledge the only tool for Java JUnit test amplification. Future replications of our study are needed to reinforce our findings and to determine whether they generalize to other test amplification approaches.

2) Internal validity: We use mutation analysis to establish that the redundant statement reduction keeps the behavior of the minimized test cases. While we have run PIT with all mutation operators enabled, it might still be that equivalent mutants disturb our behavior checking process. Future work should investigate this further.

V. Conclusion & Future work

In this paper, we looked at redundant statements in amplified test cases created by DSpot, as well as possible options for removing them from these test cases. While taint analysis or a (dynamic) slicer would be powerful, we opted for a more lightweight approach using static analysis of the test case. In a 3-step process we remove redundant statements from amplified test cases while maintaining their mutation score. Running the minimizer significantly reduced the average number of statements in the analyzed amplified test cases. We hypothesize that this makes them easier to understand.

Further research could focus on implementing a more fine-grained approach to remove redundant statements, such as the discussed slicers or taint analysis. Alternatively, a more fine-grained static analysis approach could be considered. Another consideration is the run-time of these algorithms, if one verifies the run-time of approaches using mutation testing one will always suffer heavy performance impact.

References

[1] G. Meszaros, *XUnit Test Patterns: Refactoring Test Code*. Pearson Education, 2007.
[2] M. Beller, G. Gousios, A. Panichella, S. Proksch, S. Amann, and A. Zaidman, "Developer testing in the IDE: patterns, beliefs, and behavior," *IEEE Trans. Software Eng.*, vol. 45, no. 3, pp. 261–284, 2019.
[3] "Dspot," https://github.com/STAMP-project/dspot, 2021.
[4] B. Danglot, O. Vera-Perez, Z. Yu, A. Zaidman, M. Monperrus, and B. Baudry, "A snowballing literature study on test amplification," *J. Syst. Softw.*, vol. 157, 2019.
[5] B. Danglot, O. L. Vera-Pérez, B. Baudry, and M. Monperrus, "Automatic test improvement with DSpot: a study with ten mature open-source projects," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2603–2635, 2019.
[6] C. Brandt and A. Zaidman, "Developer-centric test amplification: The interplay between automatic generation and human exploration."
[7] "Jsoup," https://github.com/jhy/jsoup, 2021.
[8] M. Harman and R. Hierons, "An overview of program slicing," *software focus*, vol. 2, no. 3, pp. 85–92, 2001.
[9] N. AlAbwaini, A. Aldaaje, T. Jaber, M. Abdallah, and A. Tamimi, "Using program slicing to detect the dead code," in *Int'l Conf on Computer Science and Information Technology*. IEEE, 2018, pp. 230–233.
[10] "Javaslicer," https://github.com/backes/javaslicer, 2016.
[11] K. Ahmed, M. Lis, and J. Rubin, "Mandoline: Dynamic slicing of android applications with trace-based alias analysis," in *Proc. Int'l Conf on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 105–115.
[12] "Slicer4j," https://github.com/resess/Mandoline, 2021.
[13] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *IEEE Symposium on Security and Privacy*, 2010, pp. 317–331.
[14] J. Bell and G. Kaiser, "Dynamic taint tracking for java with phosphor," in *Proc. Int'l Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2015, pp. 409–413.
[15] "Pit," https://pitest.org.
[16] "stream-lib," https://github.com/addthis/stream-lib, 2019.
[17] "twilio-java," https://github.com/twilio/twilio-java, 2021.
[18] W. Oosterbroek, C. Brandt, and A. Zaidman, "Replication package for "removing redundant statements in amplified test cases"," https://doi.org/10.6084/m9.figshare.14910486, 2021.