Removing redundant statements in amplified test cases

Wessel Oosterbroek, Carolin Brandt, Andy Zaidman

Delft University of Technology

Abstract

Amplified test cases created by DSpot and TestCube often contain unnecessary statements that impact the readability of the tests in question. As a part of the effort to make these amplified test cases more developer-friendly, we investigate (dynamic) slicing, taint analysis and static analysis as approaches to remove redundant statements. In addition, we evaluate a simple static analysis approach that we implemented into DSpot. 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. A problem with removing redundant statements is the fact that it, at least for the approaches we discuss in this paper, will take a significant amount of time depending on the size and quality of the original tests. While the removal of the statements themselves is relatively fast, especially when using our implemented static analysis approach, verifying that the tests still work as intended through mutation testing is resource-intensive.

1 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 unit testing, an approach where developers write tests that exercise specific and small parts of the code to be tested usually focusing on the effects of a single method (Hunt & Thomas, 2003).

However, manually creating and maintaining test suites can take a lot of time and effort, especially since sometimes a lot of unit tests are needed to reach high coverage on a relatively small part of the overall program.

One of the options to reduce the time it takes to create these test suites is to automatically generate tests instead of having developers creating them manually, reducing the time developers would have to spend on writing tests. One of the tools that has been designed to help accomplishing this task is DSpot (*DSpot*, 2021). A tool that amplifies test cases by taking as input an existing test case and giving as output new tests to improve overall coverage, the designers of DSpot envision a world where the tool can generate pull requests with test cases ready for developers to merge (Danglot, Vera-Pérez, Baudry, & Monperrus, 2019). However, while tools like DSpot have the potential advantage of reducing the time developers would have to spend on writing tests, we do not see them widely used in practice as of yet (Brandt & Zaidman, 2021). According to Brandt and Zaidman, it appears to be the case that, while these tools are capable of automatically creating relevant tests, they are also cumbersome to use. DSpot generates tests that often use vague identifiers, include no longer used statements and have weak assertions. All generated tests have to be thoroughly checked before a developer could add them to an existing test suite (Brandt & Zaidman, 2021). Therefore we aim to make this automatic test generation more developer-friendly:

> A tool is developer-friendly if it prioritizes the needs of its users (the developers) and provides functions and features in a way that lets the user perform their task effectively and efficiently (Brandt & Zaidman, 2021, p. 2).

More specifically the goal of this paper is to improve TestCube, an Intellij plugin that uses DSpot to amplify tests, by making the generated test cases easier to read and more understandable for developers (*TestCube*, 2021). TestCube mutates test cases through DSpot, generating new tests that improve code coverage. To increase understandability the generated tests each contain only one assertion (*TestCube*, 2021; Brandt & Zaidman, 2021). While there are many ways the amplified tests created by TestCube could be improved, such as by generating better identifiers and using stronger assertions (Brandt & Zaidman, 2021), in this paper, we will investigate removing no longer used statements from the amplified test cases.

Main RQ *How can no longer used statements be detected and removed from amplified test cases created by TestCube?*

To answer this research question it is important to get an idea of what exactly these statements are and how they potentially could be removed, thus this main research question raises the following sub-questions:

RQ1 *What no longer used statements do DSpot and TestCube include in the amplified test cases they generate?*
The goal here is to find out when exactly and how often DSpot generates test cases that have no longer used statements in them, as a first step to identify and find suitable solutions for deleting these unused statements. To answer this research-question we will explain the needed terminology and show an analysis of several test cases created by DSpot in Section 2.

RQ2 *What options to detect no longer used statements currently exist?*
Consequently, this paper aims to contribute a list of options that could be used to detect these no longer used statements in amplified test cases. We investigate (backwards) slicing, but also consider techniques such as static analysis and taint analysis.

RQ3 *How well do these options perform when compared to one another, keeping in mind the final use case: implementation into TestCube or DSpot?*
In addition, the potential advantages and drawbacks of said options will be discussed, the practicality of the listed options is also important as the goal is to implement one of them into DSpot or TestCube. We aim to investigate the feasibility of incorporating the options into DSpot and reason about the difference in performance between each of the options. We will discuss this research question together with **RQ2** in Section 3.

RQ4 *How does the implemented solution perform, how many of the no longer used statements are removed?*
The implemented solution will be analysed and compared to the default TestCube implementation running it on a variety of projects and tests. The main concerns here are accuracy and run-time, especially when considering the goal of making TestCube more developer-friendly, the latter is rather important. In Section 4 we will analyse the results of the conducted study.

2 Redundant Statements

In the context of this paper no longer used statements are statements whose removal does not affect the mutation coverage of a particular test in any way, also referred to as redundant statements from now on. We will discuss more about why mutation coverage is a proper way to identify these statements in section 3. This thus does not include all statements one could consider unnecessary, e.g. statements that could potentially be combined with other statements for readability are not considered in this paper.

Instead, the focus is on statements that were part of the original test case that was being amplified by DSpot, but that are no longer relevant for the amplified test case. By removing these statements we intent 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 verify 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 behaviour of the function in question even if the integer itself was not changed.

Moreover, redundant statements or code should not be confused with dead code, which is never executed at run-time and hence always unused (Haas et al., 2020). For our purposes, we will say that dead code is redundant, but that does not mean that all redundant code is dead code.

One example of redundant, but not dead code in an amplified test case are unnecessary variables. Those variables were present in the original test case, but are no longer used in the amplified one. Listing 1 presents an amplified test case with multiple redundant statements, the `first` and `attributes` objects and all associated statements are not needed in this test case. In Listing 2 the same test case is shown but with all redundant statements removed, it still has the same mutation score according to PIT, a tool used to calculate mutation coverage (Coles et al., 2016). We will refer to this process of removing redundant statements as minimizing throughout this paper.

```
1  public void booleanAttributesAreEmptyStringValues_assSep5() throws Exception {
2      Document doc = Jsoup.parse("<div hidden>");
3      Attributes attributes = doc.body().child(0).attributes();
4      attributes.get("hidden");
5      Attribute first = attributes.iterator().next();
6      first.getKey();
7      first.getValue();
8      first.hasDeclaredValue();
9      Assertions.assertFalse(((Collection) (((Document) (doc)).getAllElements())
           ).isEmpty());
10 }
```

Listing 1: An amplified test case created with the TestCube Intellij plugin.

```
1  public void booleanAttributesAreEmptyStringValues_assSep5() throws Exception {
2      Document doc = Jsoup.parse("<div hidden>");
3      Assertions.assertFalse(((Collection) (((Document) (doc)).getAllElements())
           ).isEmpty());
4  }
```

Listing 2: An amplified test case created with the TestCube Intellij plugin, with unused statements removed.

2.1 Types

To get a better understanding of the types of redundant statements created by DSpot we manually analysed over 30 amplified test cases from the JSoup Project (*JSoup*, 2021), and found that there are three main types of redundant statements that appear most often:

- 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 such unnecessary variables.

- Elements from old assertions
  Then there are elements from old assertions, by default DSpot sometimes keeps elements in from assertions that were part of the original test case, if those assertions are not at the end of the 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 often redundant as they usually do not have any side effects and are only retrieving a value, i.e. they are often getters.

- Statements with side effects
  Last are redundant statements that do have side effects, those are statements that do influence objects or variables that are (indirectly) used by the assert statements but are not relevant either, e.g. changing the surname of a Person object while the assert statements in the test are only concerned with the age of said person. We suspect that removing statements of this kind will be the most challenging.

Note that as mentioned earlier in this section, not all code that is redundant in the context of a normal program is also redundant in the context of a test case. This means that, at least for all methods we will discuss in this paper, after minimizing it needs to be verified that the test still works as expected using PIT, in general this takes a substantial amount of time that fluctuates significantly depending on the quality of the test and size of the program (Coles, 2021). In Section 4 we will further analyse the run-time of our implemented approach.

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 do have side effects will be more challenging.

3 Detecting Redundant Statements

In this section we will discuss three methods to detect and delete the redundant statements as described in Section 2, while we have specifically focused on dynamic slicers and doing static analysis on the test cases themselves, we will also briefly go into using taint analysis to detect redundant statements. 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. In Section 4 we will investigate the performance of this implemented approach.

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 when talking about 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 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 by default DSpot uses PIT mutation testing as a way to select amplified test cases, with there also being options for other selectors such as line coverage. Meaning that if it can be guaranteed that the minimized test still catches the same mutants the basis on which DSpot selected the test case remains unchanged. Note that this is only true if we use mutation testing in the same way as when the tests were created by DSpot. Otherwise, while likely being a good heuristic in practice, we can not make definitive statements about preserving the reason a test case was selected.

3.1 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 (Harman & Hierons, 2001). It has been researched quite extensively and finds main use-cases in debugging, program analysis and re-engineering (Harman & Hierons, 2001). Slicing allows ignoring parts of the program that are of no interest (Harman & Hierons, 2001).

Then there is a difference between static slicing, where we do not make any 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; All others are ignored. The result of a dynamic slicer solely concerns that execution of the program, which results in dynamic slices often being much smaller and specific as we only look at one possible way of executing the program in contrast to static slicing, in which every path is considered (Harman & Hierons, 2001).

To our knowledge there are no papers about removing redundant statements from test cases using program slicing. There are papers, such as an interesting one published in 2018, that talk about removing redundant code using program slicing in the context of a normal program (AlAbwaini et al., 2018). 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, they note that their approach becomes more difficult to use on larger programs because it becomes harder to find the effective variables (AlAbwaini et al., 2018). Moreover, the authors state that verifying the behaviour of the reduced program is done by seeing if it still has the same behaviour and output as the original, which seems problematic as that is an undecidable problem for arbitrary programs (Reus, 2016).

However removing redundant statements in test cases is notably different, as the fact that a test should still run and catch the same number of mutants can be exploited to verify that it still works correctly. In addition, 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, a practical problem is that 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. There is JavaSlicer, however it only supports JDK versions up to 1.7 (*JavaSlicer*, 2016). Another interesting slicer, which was made public too late to be considered in this research project, is Slicer4J (Ahmed, Lis, & Rubin, 2021; *Slicer4J*, 2021). Slicer4J is based on a dynamic Android slicer called Mandoline and supports modern java applications, which makes it an interesting candidate to look at for potential integration into DSpot or TestCube.

3.2 Dynamic Taint Analysis

The goal of dynamic taint analysis is to find out which computations are affected by tainted sources such as user-input (Schwartz, Avgerinos, & Brumley, 2010). It has a wide variety of use cases and can even be used in test generation (Schwartz et al., 2010). Again there is the same difference between static and dynamic taint analysis as described for slicers in the previous section.

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 in the test case. This use case is even described as a method to detect brittle assertions in a paper from 2015 about Phosphor, a dynamic taint analysis tool for Java (Bell & Kaiser, 2015). 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 (Bell & Kaiser, 2015).

However, while this is useful to detect redundant input variables and objects it does not tell much about statements that do have side-effects as we are only looking at which input variables affect the assert statement, but not at how other statements, that use input variables, affect the assert statements. While this is another interesting problem, it was not practical and time-efficient to implement Phosphor into DSpot.

3.3 Static Code Analysis

Another option is to use static analysis to look at a test case and try to infer which statements are potentially redundant and which are not. For example, all declarations of variables (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.

3.4 Algorithm

Our contribution to DSpot consists of a simple 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 Benjamin Danglot found in DSpot (*DSpot*, 2021), which tries to remove assertions that do not improve the mutation score of an amplified test.

The algorithm to delete the redundant statements contains three separate steps, to all of them the original amplified test case is given as input. In these steps the algorithm deletes certain statements from the test case, while becoming more conservative in which statements are deleted after each step. After a step the new test case is checked against the original amplified test case, if there is no difference between the two, the algorithm moves on to the next test case immediately. If there is a difference PIT will be used to verify that the test case still catches the same mutants as expected. In the case that it does the algorithm changes the current test case and moves on to the next test, if it does not the algorithm continues with the next step. If all steps fail the algorithm will return the original amplified test and continue with the next test case. Figure 1 presents a flow chart of this process.

Note that step one subsumes step two and step two subsumes step three, the individual steps consist of the following:

- **Step one:** Deleting all statements in the test, except for the assertions and statements that are needed to compile the test case.
- **Step two:** Removes 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, `loops` and variables that only interact with the assert statements when they are declared are removed as well, 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.

  In Listing 3 on line 4 an example is shown. 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:** Removes all the statements that do not directly or indirectly interact with variables used in the assert statements.

```
1  public void exampleTest() throws Exception {
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      }
8      Assertions.assertFalse(templates.equals(null));
9  }
```

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

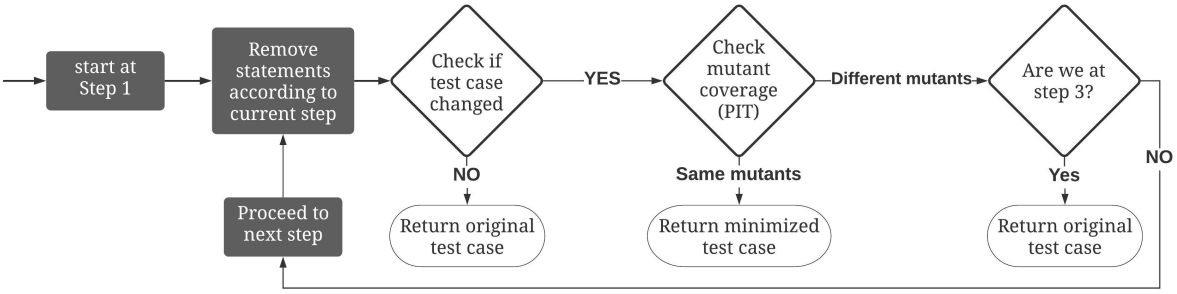

Figure 1: Overview of the implemented static analysis approach described in Section 3.4, this process is repeated for every test case that has to be minimized.

An example of this process is shown in Listings 4, 5, 6 and 7. Listing 4 presents an original 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. In Listening 6 the result of step two can be found, this test case still does not work due to the `String key = a2.setKey("three")` statement being removed, as it falls under a variable that only interacts with assertions when declared. The last Listening 7 shows the final working 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.

```
    public void hasValue_mg56_assSep142() throws Exception {
        String characters = "3a(!.#b{[Iz>YSe|%xHd";
        Attribute a1 = new Attribute("one", "");
        Attribute a2 = new Attribute("two", null);
        String key =  a2.setKey("three");
        a2.setValue(characters);
        Assertions.assertEquals("three=\"3a(!.#b{[Iz>YSe|%xHd\"", ((Attribute)
            (a2)).toString());
    }
```

Listing 4: A (modified) amplified test case created with the TestCube Intellij plugin.

```
    public void hasValue_mg56_assSep142() throws Exception {
        Attribute a2 = new Attribute("two", null);
        Assertions.assertEquals("three=\"3a(!.#b{[Iz>YSe|%xHd\"", ((Attribute)
            (a2)).toString());
    }
```

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

```
    public void hasValue_mg56_assSep142() throws Exception {
        String characters = "3a(!.#b{[Iz>YSe|%xHd";
        Attribute a2 = new Attribute("two", null);
        a2.setValue(characters);
        Assertions.assertEquals("three=\"3a(!.#b{[Iz>YSe|%xHd\"", ((Attribute)
            (a2)).toString());
    }
```

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

```
    public void hasValue_mg56_assSep142() throws Exception {
        String characters = "3a(!.#b{[Iz>YSe|%xHd";
        Attribute a2 = new Attribute("two", null);
        String key = a2.setKey("three");
        a2.setValue(characters);
        Assertions.assertEquals("three=\"3a(!.#b{[Iz>YSe|%xHd\"", ((Attribute)
            (a2)).toString());
    }
}
```

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

We created these steps as a result of analysing the tests created by DSpot, which mainly indicated a lot of unused objects declarations (and statements related to those objects) as well tests that did not use any of the statements relevant in the original test case (other than necessary object declarations). Therefore we expect to be able to remove a significant number of the redundant statements with this rudimentary approach.

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 longer include these statements. More about this can be found in section 5. Moreover, we are using DSpot to run PIT, meaning that the exact same parameters are used as when amplifying using mutation testing.

Answer to RQ2: To the best of our knowledge there exists no related work concerning the removal of redundant statements from test cases. Therefore we investigate three ways to delete redundant statements from test cases: (dynamic) slicers, taint analysis and static code analysis. The suggested approaches are inspired by existing tools and use-cases, i.e. brittle test detection using taint analysis and deleting redundant statements from code using slicers. While the list of approaches is non-exhaustive, it should give a good idea of how it might be possible to delete redundant statements.

Answer to RQ3: While dynamic slicers and taint analysis tools are worthwhile looking at, our implemented solution consists of a static analysis algorithm. The implemented solution has the advantage of being straightforward to implement compared to dynamic slicers and taint analysis. We expect that it should be able to remove a significant number of redundant statements, by removing statements that are likely to be redundant and verifying their removal using PIT.

4 Analysis

In this section we will answer research question four by discussing the performance of the implemented algorithm, described in Section 3.4. To that end we have performed a qualitative as well as a quantitative study. In the qualitative section we will focus on what lines do get removed and which do not by manually going over 34 test cases. While in the quantitative section we will investigate the overall performance by measuring the difference in statements before and after minimization, to this end we have created 274 amplified tests, which originate from 15 test classes that we amplified, using tests from the JSoup, Stream-Lib and Twilio-Java projects (*JSoup*, 2021; *stream-lib*, 2019; *twilio-java*, 2021). Two of these classes containing 34 tests, from the JSoup project, were manually checked for redundant statements before and after running the minimizer.

4.1 Qualitative Analysis

We analysed 34 test cases and found that our approach is able to remove 47% (73 out of 154) of all redundant statements in these tests. Note that these test cases still include elements from old assertions (See Section 2.1), removing those statements would significantly increase the number of statements removed. Listing 8 shows a test case where lines 6-12, which are all elements from old assertions, are redundant as they only retrieve values and have no side-effects. Additionally, manually analysing the minimized test cases shows that there are quite a few test cases in which not even close to all of the redundant statements get removed, while others are minimized as much as possible.

This is caused by the algorithm removing all but strictly necessary statements in step one, meaning that if this first minimization step succeeds the test case is likely to be fully minimized. As DSpot often generates test cases that only use these necessary object declarations, by calling an arbitrary method on an object, this occurs in quite a few test cases. On the other hand, there are also examples of their not being any redundant statements removed in a test case, in Listing 9 a test case is shown that still includes a redundant `for loop`, because step two of the minimization fails. The reason for this is line 6, in which the `DSPOTinvoc5` object is declared, this line is marked as redundant in step two due to only interacting with needed variables (i.e. the `a` object) when being declared, however in this case the statement `a.put("data -name", "3")` does have side-effects causing the test case to fail. This shows that the removal of redundant statements is dependent on the structure and assertions in the test case.

All redundant statements of the type unnecessary variable were removed, however that only contributed a relatively small part of all removed redundant statements. While the majority of removed statements consist of the other two types, described in Section 2, their removal is not guaranteed and depends on the test structure and usage of declared variables.

```
1   public void html_assSep5() throws Exception {
2       Attributes a = new Attributes();
3       a.put("Tot", "a&p");
4       a.put("Hello", "There");
5       a.put("data-name", "Jsoup");
6       a.size();
7       a.hasKey("Tot");
8       a.hasKey("Hello");
9       a.hasKey("data-name");
10      a.hasKey("tot");
11      a.hasKeyIgnoreCase("tot");
12      a.getIgnoreCase("hEllo");
13      Assertions.assertEquals(-758045610, ((int) (((Attributes) (a)).
            hashCode())));
14  }
```

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

```java
public void testIteratorHasNext_rv161_assSep269() throws Exception {
    String __DSPOT_key_74 = "vk:PDcJ+3%i%04t]/|:I";
    Attributes a = new Attributes();
    a.put("Tot", "1");
    a.put("Hello", "2");
    Attributes __DSPOT_invoc_5 = a.put("data-name", "3");
    int seen = 0;
    for (Attribute attribute : a) {
        seen++;
        String.valueOf(seen);
        attribute.getValue();
    }
    __DSPOT_invoc_5.removeIgnoreCase(__DSPOT_key_74);
    Assertions.assertEquals(" Tot=\"1\" Hello=\"2\" data-name=\"3\"", ((
        Attributes) (a)).toString());
}
```

Listing 9: Minimized test case with where step two failed, leaving in a lot of redundant statements.

4.2 Quantitative Analysis

In Figure 2 the results of minimizing the 274 amplified tests can be found, showing a clear reduction in the average and median 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.

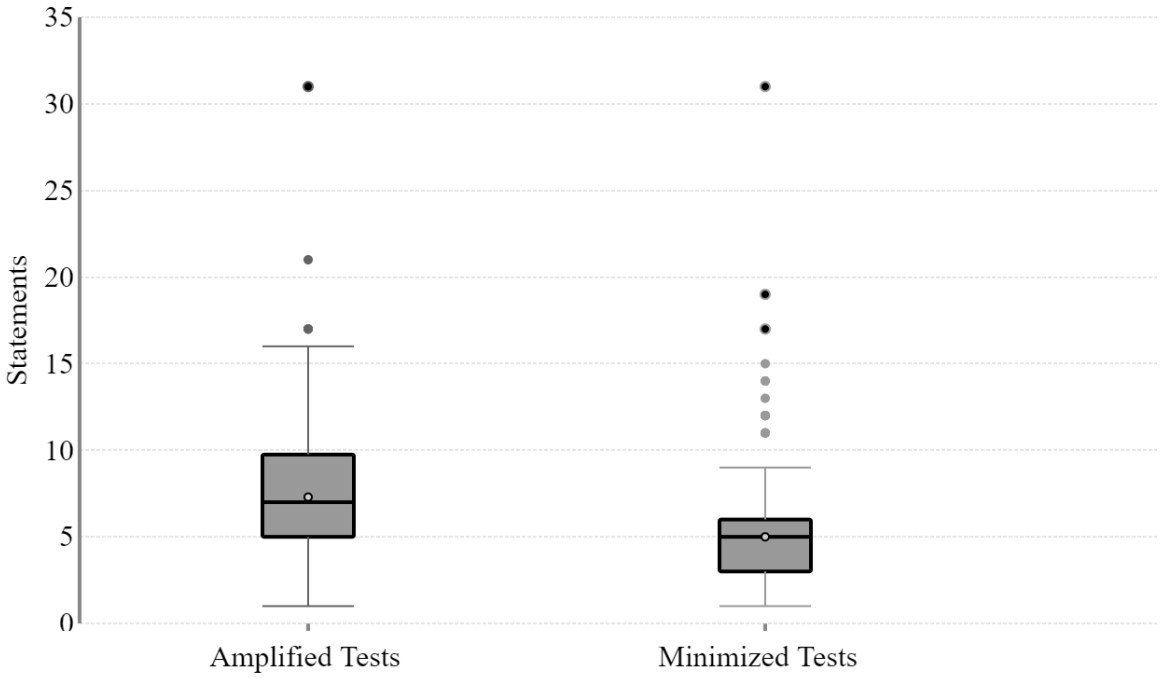

Figure 2: Box plots showing the number of statements in 274 tests before and after minimizing using the approach described in section 3.4.

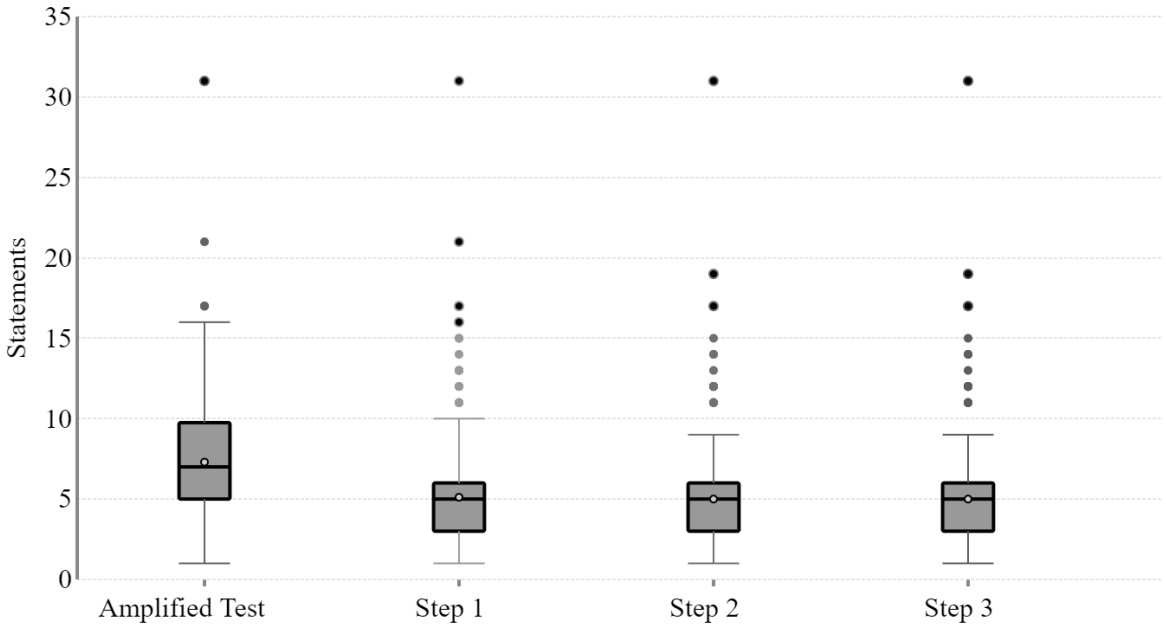

Figure 3: Box plots showing the number of statements in 274 tests before and after minimizing using the approach described in section 3.4.

An interesting observation can be made when looking at Figure 3 which shows the minimization process taking place step by step. By far the largest number of statements is removed in step one, in step two a couple statements are removed, while in step three no statements were removed at all.

Step three not removing any statements is the result of those statements already being removed in either steps one or two. There were not any test cases where the first two steps failed and there being 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 selecting other test cases.

4.3 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 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 minimize a test case.

Answer to RQ4: 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 the fact that it using mutation coverage to verify that the tests still work takes a significant amount of time. In fact, the overwhelming majority of time spent during minimization is used for calculating mutation coverage.

5 Discussion

When looking at the results more in-depth it is often the case that there are either one of two things happening: either the test case did use (almost) none of the statements in the original test, which means that most of those statements get removed in step one of the algorithm, or test case does use statements from the original test case. In the second case only the objects which are not related at all get deleted, which often means a reduction of only one or two lines at most.

What we are doing here is taking a very small subset of all possible combinations of all statements in a test case, namely all of those described in Section 3.4. We could add a step to remove `for loops` and run PIT to check if the test case works as intended, in which Listing 9 would have been minimized correctly. However, that comes at the cost of doing more PIT runs for every test case. The extreme being testing every possible combination, which should result in all redundant statements being removed from each test, but comes at the cost of being resource-intensive.

Moreover, mutation testing is slow, removing the statements themselves takes milliseconds, but running PIT for every test can take a significant amount of time. This issue is amplified because it is sometimes needed to run PIT through DSpot multiple times, checking if the test works after each step.

Another interesting thing to note is that we did not specifically consider block statements. This is important because DSpot generates try-catch blocks when asserting exceptions. Detecting redundant statements inside a try-catch block requires a approach than the ones discussed in this paper, as the assert statement does not contain any information about the used variables and the assertion could be thrown during any statement.

5.1 Responsible Research

It has to be noted that the variety of projects and tests selected to test the minimizer on is fairly limited. For one they are all open source projects, with most tests coming from the JSoup project. Moreover, we suspect that the performance of our approach will suffer the more complicated the tests become. 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. However, we do not expect major differences in the number of removed statements between regular applications, given that they use a similar style of unit tests.

5.2 Reproducibility

To reproduce the results discussed in this paper we have created a replication package that can be accessed through https://doi.org/10.5281/zenodo.5032346. It includes details on how to generate tests and use the minimize as well as all data (raw amplified tests) before and after minimizing. The repository with a fork of DSpot containing our minimizer can be found at https://github.com/WortelSoep/dspot.

6 Conclusion & Future work

In this paper we have looked at the nature of the 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 likely be a great fit, we have opted for a simpler, more rudimentary approach using static analysis of the test case. In a process of three steps we attempt to remove as many redundant statements from a test case as possible and our results show that this approach works well due to the nature of redundant statements in tests created by DSpot. 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 and read.

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 as well. 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 when using dynamic slicing. One might be able to skip this step, while it is true that it seems unlikely that the issue of redundant statements being actually useful in a test case will go away, it might not be that relevant but further research should look at this.

References

Ahmed, K., Lis, M., & Rubin, J. (2021). Mandoline: Dynamic slicing of android applications with trace-based alias analysis. In *2021 14th ieee conference on software testing, verification and validation (icst)* (pp. 105–115).

AlAbwaini, N., Aldaaje, A., Jaber, T., Abdallah, M., & Tamimi, A. (2018). Using program slicing to detect the dead code. In *2018 8th international conference on computer science and information technology (csit)* (pp. 230–233).

Bell, J., & Kaiser, G. (2015). Dynamic taint tracking for java with phosphor (demo). In *Proceedings of the 2015 international symposium on software testing and analysis* (p. 409â413). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi-org.tudelft.idm.oclc.org/10.1145/2771783.2784768 doi: 10.1145/2771783.2784768

Brandt, C., & Zaidman, A. (2021). Developer-friendly test amplification. Unpublished.

Coles, H. (2021). *Pitest.* Retrieved 2021-06-07, from https://pitest.org/

Coles, H., Laurent, T., Henard, C., Papadakis, M., & Ventresque, A. (2016). PIT: a practical mutation testing tool for java. In *Proceedings of the 25th international symposium on software testing and analysis* (pp. 449–452).

Danglot, B., Vera-Pérez, O. L., Baudry, B., & Monperrus, M. (2019). Automatic test improvement with DSpot: a study with ten mature open-source projects. *Empirical Software Engineering*, *24*(4), 2603–2635.

Dspot. (2021). https://github.com/STAMP-project/dspot. GitHub.

Haas, R., Niedermayr, R., Roehm, T., & Apel, S. (2020). Is static analysis able to identify unnecessary source code? *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *29*(1), 1–23.

Harman, M., & Hierons, R. (2001). An overview of program slicing. *software focus*, *2*(3), 85–92.

Hunt, A., & Thomas, D. (2003). *Pragmatic unit testing in java with junit.* The Pragmatic Bookshelf.

Javaslicer. (2016). https://github.com/backes/javaslicer. GitHub.

Jsoup. (2021). https://github.com/jhy/jsoup. GitHub.

Reus, B. (2016). More undecidable problems. In *Limits of computation: From a programming perspective* (pp. 97–112). Cham: Springer International Publishing. Retrieved from https://doi.org/10.1007/978-3-319-27889-6_9 doi: 10.1007/978-3-319-27889-6_9

Schwartz, E. J., Avgerinos, T., & Brumley, D. (2010). All you ever wanted to know about dynamic

taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 ieee symposium on security and privacy* (p. 317-331). doi: 10.1109/SP.2010.26
Slicer4j. (2021). https://github.com/resess/Mandoline. GitHub.
stream-lib. (2019). https://github.com/addthis/stream-lib. GitHub.
Testcube. (2021). https://github.com/TestShiftProject/test-cube. GitHub.
twilio-java. (2021). https://github.com/twilio/twilio-java. GitHub.