MEDIC: A Static Analysis Framework for Equivalent Mutant Identification

Marinos Kintis[a,*], Nicos Malevris[a]

[a]*Athens University of Economics and Business, Department of Informatics, 76 Patission Street, 10434, Athens, Hellas*

Abstract

Context: The equivalent mutant problem is a well-known impediment to the adoption of mutation testing in practice. In consequence of its undecidable nature, a complete automated solution is unattainable. To worsen the situation, the manual analysis of the generated mutants of a program under test is prohibitive due to their vast number and the complexity of determining their equivalence.

Objective: This paper focuses on the automated identification of equivalent and partially equivalent mutants, i.e. mutants that are equivalent to the original program for a specific subset of paths. To this end, the utilisation of a series of previously proposed data flow patterns is investigated. This study also examines the cross-language nature of these patterns and the killability of the detected partially equivalent mutants.

Method: A tool, named MEDIC (Mutants' Equivalence DIsCovery), incorporating the aforementioned patterns was developed. Its efficiency and effectiveness were evaluated based on a set of manually analysed mutants from real-world programs, written in the Java programming language. Furthermore, MEDIC was employed to test subjects written in the JavaScript programming language.

Results: MEDIC managed to detect **56** per cent of the examined equivalent mutants in **125** seconds, providing strong evidence regarding both its effectiveness and efficiency. Additionally, MEDIC was able to identify equivalent mutants in the JavaScript test subjects, lending colour to the cross-language nature of the implemented patterns. Finally, the identified partially equivalent mutant set consisted largely of killable mutants, 16 per cent of which were stubborn ones.

*Corresponding author

Email addresses: `kintism@aueb.gr` (Marinos Kintis), `ngm@aueb.gr` (Nicos Malevris)

Conclusion: It can be concluded that pattern-based equivalent mutant identification forms a viable approach for combating the equivalent mutant problem. MEDIC automatically detected a considerable number of the manually identified equivalent mutants and was successfully applied to test subjects in all examined programming languages.

Keywords: mutation testing, equivalent mutant detection, stubborn mutant detection, static analysis, data flow analysis

1. Introduction

Mutation testing [1, 2] is a well-studied technique that has been applied to many software artefacts and testing levels [3]. Mutation works by inserting specific types of faults in a program under test, which is called the original program, thus creating many different program versions, which are termed mutants. The inserted faults are in essence simple syntactic changes that are based on predefined rules, which are known as mutation operators. Mutants denote faults that programmers typically make or certain characteristics of a program that should be exercised, e.g. each branch [4]. It has been empirically found that a carefully selected set of mutation operators will lead to a more thorough testing process [5, 6]. Additionally, several research studies have compared the effectiveness of mutation testing with various testing criteria, e.g. data flow criteria [7, 8, 9, 10, 11], and concluded that mutation is more effective.

Concisely, mutation entails four phases: (a) the mutant generation phase; (b) the test data generation phase; (c) the mutant execution phase; and (d) the equivalent detection phase. The mutant generation phase refers to the generation of the mutants of the program under test. This step has been fully automated by tools that apply specific mutation operators to the program under test. An example of such a tool is the muJava testing framework [12] for the Java programming language, which is utilised in the present study. The next phase (b) is concerned with the generation of appropriate test data that distinguish the behaviour of the original program from the one of its mutants. Such a test case is said to kill the corresponding mutants and these mutants are said to be killable, i.e. they can be killed by a test case. Although the automated test data generation for mutation still remains an open issue, partial solutions have been proposed in the literature, e.g. [13, 14, 15].

At the third stage (c) of the mutation testing process, the mutants and the original program are executed with the generated test cases to discover the killed ones. Unfortunately, not all mutants can be killed: some mutants are semantically identical to the original program, even though they are syntactically different. These mutants, termed equivalent mutants, are the primary concern of the last phase (d) of mutation testing. At this particular stage, the mutants that have not been previously killed, termed alive mutants, must be manually inspected to determine whether they are equivalent ones or the available test data are inadequate for killing them, thus new data must be created. Previous research on this area has introduced several semi-automated solutions to the equivalent mutant problem, e.g. [16, 17, 18, 19, 20]. It should be mentioned that the last three phases of mutation are repeated until all killable mutants have been killed or a predefined stopping point has been reached.

This study focuses on the automated identification of equivalent mutants and empirically investigates a set of previously proposed data flow patterns [21], whose presence in the source code of the program under test will lead to the generation of equivalent and partially equivalent mutants. A partially equivalent mutant is equivalent to the original program for specific paths. As a consequence, these paths cannot produce killing test cases for the corresponding mutant [21]. Despite this fact, partially equivalent mutants can be killable. Since these patterns are determined statically, they can be incorporated into mutation testing frameworks to weed out a portion of the total equivalent mutants *before* their generation, thus reducing the manual cost of mutation.

The contributions of the present paper can be summarised in the following points:

1. An automated framework, named MEDIC (Mutants' Equivalence DIsCovery), which implements the aforementioned data flow patterns and can detect equivalent mutants in different programming languages. MEDIC is the first tool, to the authors' knowledge, that is reported to achieve such a task.
2. An empirical study, based on 1287 manually analysed mutants from real-world projects, which investigates MEDIC's detection power, performance and cross-language nature.
3. Insights regarding the nature of partially equivalent mutants and the difficulty in killing them.

The rest of the paper is organised as follows: Section 2 furnishes a more detailed view of the equivalent mutant problem and the underlying data flow patterns. Sections 3 and 4 present the implementation details of MEDIC, along with the conducted empirical study and the obtained experimental results, while Section 5 discusses the pertaining threats to validity. Finally, Sections 6 and 7 describe similar research studies and conclude this work, respectively.

2. Background

This section provides information regarding the equivalent mutant problem and succinctly describes the implemented data flow patterns in conjunction with appropriate examples, belonging to the examined test subjects.

2.1. The Equivalent Mutant Problem

Detecting equivalent mutants is an arduous task, primarily due to the undecidable nature of the underlying problem [16, 22], which imposes their manual analysis. This manual effort has been estimated to require approximately 15 minutes per equivalent mutant [20]. Taking this fact into consideration, along with the estimated number of equivalent mutants per program which ranges between 10 and 40 per cent of the generated ones [3], it can be easily concluded that the required human effort may become unbearable even for small projects. Thus, the need for (semi-)automated solutions becomes apparent. Although such solutions are not a panacea, their therapeutic properties to the practical application of mutation are indisputable. To this end, this paper introduces MEDIC, an automated framework that is capable of detecting more than **90** equivalent mutants in just **125** seconds, as Section 4 presents.

2.2. Problematic Data Flow Patterns

Recently, several data flow patterns that detect source code locations capable of generating equivalent mutants has been introduced [21]. These patterns examine the data flow information of the program under test. The underlying analysis is based on the *Static Single Assignment* (SSA) intermediate representation [23, 24]; a form utilised by modern compilers for the internal representation of the input program.

The basic characteristic of the SSA representation is that each variable of the program under consideration is assigned exactly once. Essentially, for each assignment to a variable, that variable is uniquely named and all its uses, reached by that assignment, are appropriately renamed [25]. An example of this representation is illustrated in Figure 1. The first part of the figure presents a portion of the source code of the `uncapitalize` method of the `Commons` test subject and the second part its SSA representation, as obtained by the T. J. Watson Libraries for Analysis (WALA) framework [26], a program analysis tool utilised by MEDIC. The lines of each part of the figure are numbered in such a way that directly relates the source code with its SSA representation: the SSA representation of line `1` of the source code (part *a*) is line `1` of part *b*, for line `5` of part *a*, the corresponding line is part *b*'s line `5` and so on. By examining this figure, it becomes clear that each variable is uniquely named in the SSA representation based on its assignments in the source code and that its uses are also appropriately renamed. This brief description of the figure's contents is sufficient for the purposes of this introduction. Section 3 elaborates on this example and describes the WALA and MEDIC frameworks in more detail.

The examined patterns can lead to the detection of equivalent and partially equivalent mutants [21]. A partially equivalent mutant is a mutant that is equivalent to the original program for only a specific set of paths, whereas an equivalent one is equivalent to the original program for all paths. The identification of partially equivalent mutants is considered valuable due to the discovery of the subset of paths that will not yield a killing test case for the corresponding mutant. The present work empirically studies the concept of partially equivalent mutants and discusses the corresponding findings in Section 4.

2.2.1. Use-Def (UD) Category of Patterns

The Use-Def category of patterns targets equivalent mutants that are due to mutation operators that insert the post-increment (e.g. `var++`) or decrement

```
     ...
 1:  int strLen = str.length();
 2:  StringBuffer buffer = new StringBuffer(strLen);
 3:  boolean uncapitalizeNext = true;
 4:  for (int i = 0; i < strLen; i++) {
 5:      char ch = str.charAt(i);
 6:      if (isDelimiter(ch, delimiters)) {
 7:          buffer.append(ch);
 8:          uncapitalizeNext = true;
 9:      }
10:      else if (uncapitalizeNext) {
11:          buffer.append(Character.toLowerCase(ch));
12:          uncapitalizeNext = false;
13:      }
14:      ...
15:  }
     ...
```

(a) Example Code fragment.

```
     ...
 1:  v12 = v1.length()                   → defines: strLen, uses: str
 2:  v13 = new StringBuffer(v12)         → defines: buffer, uses: strLen
 4:  v30 = phi v15:#1, v28               → defines: uncapitalizeNext
     v31 = phi v10:#0, v29               → defines: i
     conditional branch(lt) v31,v12      → uses: i and strLen
 5:  v17 = v1.charAt(v31)                → defines: ch, uses: i and str
 6:  v19 = isDelimiter(v17, v2)          → uses: ch and delimiters
     conditional branch(eq) v19,v10:#0   → uses: v19 (internal) and v10
 7:  v21 = v13.append(v17)               → uses: ch and buffer
10:  conditional branch(eq) v30,v10:#0   → uses: uncapitalizeNext and v10
11:  v23 = Character.toLowerCase(v17)    → uses: ch
     v25 = v13.append(v23)               → uses: buffer and v23 (internal)
14:  v28 = phi v15:#1, v10:#0, v30       → defines: uncapitalizeNext
     v29 = v31 + v15:#1                  → defines: i
     ...
```

(b) Its SSA representation.

Figure 1: Code fragment of the `Commons` test subject and its SSA representation.

(e.g. `var−−`) arithmetic operators or other similar operators. These mutation operators are included in many mutation testing systems, e.g. the muJava framework for the Java programming language [12] and the Proteum system for the C programming language [27]. The basic characteristic of such operators is that they do not change the value of the affected variable at the evaluation of the affected expression. The variable is used unchanged and the next time it is referenced, it will be incremented or decremented by one. Therefore, if the examined use cannot reach another use, the induced change is bound to be indiscernible, leading to the generation of equivalent mutants. To prevent this situation, the patterns belonging to this category search for uses of variables that do not reach another use. More precisely, they search for uses of variables that (1) reach definitions before reaching other uses and/or (2) they reach no other use. These problematic situations are bound to generate two equivalent mutants. Consider, for instance, the use of variable `ch` at line 11 of Figure 1. This variable constitutes a valid target for the Arithmetic Operator Insertion Short-cut (AOIS) mutation operator of the muJava framework. Furthermore, note that the use of line 11 will either reach another use only *after* a definition of the same variable or will not reach another use (in the case the `for` block exits). Taking these facts into account, it becomes clear that the application of the post-increment and decrement operators at this variable, i.e. the application of the AOIS mutation operator, will result in two equivalent mutants. Analogous cases are handled by the patterns of this category, which are described subsequently.

SameLine-UD Problematic Pattern. The SameLine-UD problematic pattern detects solely equivalent mutants that affect a variable that is being used and defined at the same statement. In this particular case, the changes induced by the post-increment and decrement operators can never be distinguished. Figure 2a presents an example from the `classify` method of the `Triangle` test subject: the application of AOIS at line 29 will generate two equivalent mutants, denoted by the symbol Δ, affecting variable `trian`. The SameLine-UD pattern discovers this problematic source code location by identifying the use and the definition of `trian` at the corresponding line, as highlighted in the figure.

SameBB-UD Problematic Pattern. This pattern also detects solely equivalent mutants, but in this case the use of the variable and its definition belong to the same basic block. Thus, SameBB-UD searches for basic blocks that contain uses and definitions of the same variable and the use precedes the definition with no intermediate uses. The application of AOIS to such a case will inevitably lead to the generation of equivalent mutants. An example of this problematic situation belonging to the `wrap` method of the `Commons` test subject, is depicted in Figure 2b, where variable `offset`, which constitutes a valid target for AOIS, is used at line 56 and is later defined at line 57. This situation is detected by SameBB-UD, leading to the discovery of the two equivalent mutants depicted in the figure.

DifferentBB-UD Problematic Pattern. The DifferentBB-UD pattern detects partially equivalent mutants and in specific cases equivalent ones. This pattern searches for uses and definitions of variables between different basic blocks. More precisely, it searches for uses of variables at one basic block (the using basic block) that can reach a definition of that variable at another basic block (the defining basic block), without other intermediate uses or definitions. Such being the case, a path connecting the using and defining basic blocks cannot yield a killing test case for the considered mutants (ones generated by the AOIS mutation operator). Thus, these mutants can be identified as partially equivalent

```
    ...
28: if (a == c) {
29:     trian = trian + 2;
    Δ       ... trian++ ...
    Δ       ... trian-- ...
30: }
    ...
```

(a) `Triangle`: SameLine-UD – Equivalent

```
    ...
55: else {
56:     ... (str.substring( offset ));
    Δ                ... (offset++));
    Δ                ... (offset--));
57:     offset = inputLineLength;
58: }
    ...
```

(b) `Commons`: SameBB-UD – Equivalent

```
    ...
30: while (ABS(diff) > mEpsilon) {
31:     if (diff < 0) {
32:         m = x;      // BB:4
33:         x = ( M + x) / 2;
    Δ          ... M++ ...
    Δ          ... M-- ...
34:     }
35:     else if (diff > 0) {
36:         M = x;   // BB:6
37:         x = (m + x) / 2;
38:     }
39:     diff = x * x - N;
40: }
    ...
```

(c) `Bisect`: DifferentBB-UD – Partially Eq.

```
117: for (; i < length; i++) {
118:     char c = name.charAt(i);
119:     if (...) {
120:         ...
122:     }
123:     else if (...) {
124:         ...
125:     }
126:     else {
127:         result.append( c );
     Δ                 ... (c++);
     Δ                 ... (c--);
128:     }
129: }
     ...
```

(d) `XStream`: DifferentBB-UD – Equivalent

Figure 2: Problematic situations that are detected by the Use-Def (UD) category of patterns. Each part presents a code fragment of a studied test subject, the name of the discovering pattern and the type of the detected mutants. (Mutants are denoted by the Δ symbol.)

for this specific path. It should be noted that this fact does not mean that the mutants are equivalent, it means that they cannot be killed by targeting that particular path. An instance of this case is illustrated in Figure 2c. This figure presents a code fragment of the `sqrt` method of the `Bisect` test subject. It can been seen that variable `M` is used at basic block 4 and is defined at basic block 6. Thus, a path that contains these basic blocks with that particular order and no other uses of `M` cannot yield a killing test case for the mutants depicted in the figure. Apart from detecting partially equivalent mutants, DifferentBB-UD can also detect equivalent ones: in this case, (1) the previously stated conditions must hold for all paths connecting the using and defining basic blocks and (2) no path should exist that connects the using basic block with another one that uses the respective variable and does not include an intermediate basic block that defines the corresponding variable. It should be mentioned that this last condition was not included in the original definition of this pattern and was added to address a corner case belonging to a studied test subject. Figure 2d, which depicts a code fragment of the `decodeName` method of the `XStream` test subject, presents an instance of equivalent mutant detection. The problematic pattern resides in lines 127 and 118. At the former, variable `c` is used and at the latter it is defined. The application of AOIS at this specific code location will lead to the generation of two equivalent mutants, which are discovered by the DifferentBB-UD pattern. Another example of equivalent detection based on this pattern was presented at the beginning of this subsection.

The aforementioned patterns search for uses of variables that can reach a definition. As a special case, the Use-Ret (UR) category of patterns, described below, searches for uses that do not reach another use.

2.2.2. Use-Ret (UR) Category of Patterns

As mentioned earlier, this family of patterns searches for uses of variables that do not reach another use. Thus, the application of the AOIS mutation operator at these specific code locations will lead to the generation of equivalent or partially equivalent mutants. This category includes three patterns that are analogous to the previously presented ones.

SameLine-UR Problematic Pattern. This pattern targets equivalent mutants that relate to problematic situations where a variable is used at an exiting statement, e.g. a `return` statement. An example of such a case is present in Figure 3a, which illustrates a code fragment of the `dividedBy` method of the `Joda-time` test subject. It can be seen that variable `divisor` is used at the `return` statement of line 190; the application of AOIS at this particular line will result in two equivalent mutants (depicted in the figure) that are detected by this pattern.

SameBB-UR Problematic Pattern. The SameBB-UR pattern is analogous to the SameBB-UD one, but instead of searching for uses of a variable that coexist with definitions of the same variable inside a basic block, it searches for basic blocks that contain exiting statements and include uses of variables that do not reach other uses. An instance of this situation is illustrated in Figure 3b, which presents a code fragment of the `removeSource` method of the `Pamvotis` test subject. It can be seen that the highlighted use of variable `position` does not reach another use before the `return` statement of line 85. Thus, the application of the AOIS mutation operator will generate two equivalent mutants that are discovered by this pattern.

DifferentBB-UR Problematic Pattern. This pattern constitutes a special case of the DifferentBB-UD one in that it searches for uses of variables at a basic block that can reach an exiting basic block without any intermediate uses or definitions. In this particular circumstance, the application of the AOIS mutation operator can lead to the generation of partially equivalent mutants or equivalent ones. An example of the former case is depicted in Figure 3c. This code fragment belongs to the `isDelimiter` method of the `Commons` test subject.

```
...
187: if (divisor == 1) {
188:     return this;
189: }
190: return mins(getVal()/ divisor );
  Δ                ... /divisor++);
  Δ                ... /divisor−−);
...
```

(a) `Joda-time`: SameLine-UR – Equivalent

```
...
83: if (position != -1) {
84:     ... rmElementAt( position );
  Δ                ... (position++);
  Δ                ... (position−−);
85:     return true;
86: } else {
...
88: }
```

(b) `Pamvotis`: SameBB-UR – Equivalent

```
...
214: for (... ; i < isize; i++) {
215:     if ( ch == delimiters[i]) {
  Δ        (ch++ ...)
  Δ        (ch−− ...)
216:         return true;
217:     }
218: }
219: return false;
...
```

(c) `Commons`: DifferentBB-UR – Partially Eq.

```
...
49: else {
50:     if (tr == 3 && b + c>a) {
  Δ            ... && b++ ...
  Δ            ... && b−− ...
51:         return ISOSCELES;
52:     }
53: }
...
55: return INVALID;
```

(d) `Triangle`: DifferentBB-UR – Equivalent

Figure 3: Problematic situations that are detected by the Use-Ret (UR) category of patterns. Each part presents a code fragment of a studied test subject, the name of the discovering pattern and the type of the detected mutants. (Mutants are denoted by the Δ symbol.)

It can be seen that the only use of variable `ch` is at line 215 which lies inside a `for` block. Consequently, any path that does not contain more than one loops, i.e. more than one uses of this variable, cannot produce killing test cases for the AOIS mutants. The DifferentBB-UR pattern identifies this fact and marks this code location as problematic and the corresponding mutants as partially equivalent. Figure 3d presents an instance of a code location that will generate equivalent mutants. This figure illustrates a code fragment of the `classify` method of the `Triangle` test subject. It can be seen that variable `b` is used at line 50 and has no other uses after this line. Thus, the mutants created by the insertion of the post-increment and decrement arithmetic operators can never be killed. The DifferentBB-UR pattern discovers this problematic situation.

2.2.3. Def-Def (DD) Category of Patterns

The Def-Def category of patterns targets problematic situations that are caused by definitions of variables instead of uses. This constitutes the main difference between this category and the aforementioned ones. The only pattern that belongs to this family is the DD problematic pattern.

DD Problematic Pattern. This pattern detects problematic situations that arise from the existence of two consecutive definitions of a variable, belonging to different basic blocks, with no intermediate uses. Such being the case, any mutation operator that changes the first definition is bound to generate partially equivalent mutants or equivalent ones. In the former situation, the two definitions must be reached by at least one path with no intermediate uses – these paths cannot produce a killing test case for the respective mutants. In the latter case, every path reaching the first definition must reach a second one (with no intermediate uses), thus, the change induced by mutation operators affecting the first definition cannot be discerned. Examples of such mutation operators are: the AODU (Arithmetic Operator Deletion Unary) and the AORB (Arithmetic Operator Replacement Binary) mutation operators of the muJava framework. An instance of a code location that will generate partially equivalent mutants is presented in Figure 4a. This code fragment belongs to the `removeSource` method of the `Pamvotis` test subject. It can be seen that variable `position` is defined twice, at lines 76 and 79 respectively, and there is no use of that variable prior to the second definition. Due to the fact that these two definitions can be reached by at least one path and there is no use between them, it can be concluded that mutants affecting the definition of line 76 are partially equivalent. The depicted mutant, produced by the AODU mutation operator which deletes unary arithmetic operators, falls into this category. The DD pattern can also detect equivalent mutants: Figure 4b illustrates an example belonging to the `addNode` method of the `Pamvotis` test subject. In this code fragment, variable `nAifsd`, which is defined at line 1089, is later redefined at every `case` block and

```
    ...
76: int position = -1;
    Δ          ... = 1;
77: for (int i = 0; ... ; i++) {
78:     if (...) {
79:         position = i;
80:         break;
81:     }
82: }
    ...
```

(a) `Pamvotis`: DD – Partially Eq.

```
1089: float nAifsd = sifs + 2 * slot;
      Δ                  ... / slot;
1090: switch (ac) {
1091: case 1 :
1092: {   ...
1095:     nAifsd = sifs + aifs1 * slot;
1096:     ...
1097: }
      ...
1115: default :
1116: {   ...
1119:     nAifsd = sifs + aifs0 * slot;
1120:     ...
1121: }
```

(b) `Pamvotis`: DD – Equivalent

Figure 4: Problematic situations that are detected by the Def-Def (DD) category of patterns. Each part presents a code fragment of a studied test subject, the name of the discovering pattern and the type of the detected mutants. (Mutants are denoted by the Δ symbol.)

at the `default` block of the `switch` statement, without any intermediate uses. Consequently, every path reaching the definition of line 1089 will definitely reach a second one. The DD pattern identifies this fact and reports that the mutants affecting the right expression of this definition are equivalent ones. An instance of such a mutant, generated by the AORB mutation operator which replaces binary arithmetic operators, is presented in the figure.

2.2.4. Def-Ret (DR) Category of Patterns

This category, which constitutes a special case of the aforementioned one, searches for definitions of variables that do not reach a use. In such a case, the induced change of the mutants affecting that particular definition is indistinguishable. Unfortunately, this implies the existence of unused variables in the program under test, which is or should be scarce in a real-world application. The empirical assessment of these patterns showed no instances of equivalent mutant detection (see Section 4 for more details), thus, their description is not accompanied by appropriate source code examples.

SameBB-DR Problematic Pattern. The SameBB-DR pattern searches for definitions of variables at a basic block that contains (1) an exiting statement and (2) no use of those variables between the considered definition and the exiting statement. It is obvious that such a case will generate equivalent mutants. The empirical evaluation of this pattern revealed an additional requirement: (3) the variable whose definition is affected must be local, i.e. it should not be possible to access that variable after the exit of the corresponding method. Consider for example Figure 5. This figure presents a code fragment of the `Bisect` test subject. According to the original definition of the SameBB-DR pattern, the highlighted definition of variable `mResult` will be detected as a problematic code location that will generate equivalent mutants. Upon closer inspection, it becomes apparent that these mutants can be easily killed by examining the value of `mResult` after the end of the respective method since it is not a local variable. The newly added requirement refines SameBB-DR's definition and rectifies the aforementioned situation.

DifferentBB-DR Problematic Pattern. The DifferentBB-DR pattern searches for definitions of variables at one basic block that reach an exiting basic block, without any intermediate uses. This pattern can detect both equivalent and partially equivalent mutants. In the first case, every path reaching the considered

```
    ...
41: r = x;
42: mResult = r;        // mResult is not a local variable
    Δ      ... = -r;
43: return r;
```

Figure 5: A detected problematic situation based on the original definition of SameBB-DR. The depicted mutant (denoted by the Δ symbol) is killable because `mResult` is not a local variable. The refined definition of this pattern rectifies this situation.

definition must also reach an exiting basic block (with no intermediate uses) and in the second one, at least one such path must exist between the defining and exiting basic blocks. The empirical evaluation of this pattern revealed no cases of equivalent mutant detection, similarly to the SameBB-DR one, although it did reveal instances of partially equivalent mutants.

3. MEDIC – Mutants' Equivalence DIsCovery

The primary focus of this paper is the empirical evaluation of the problematic patterns described in the previous section. To fulfil this, an automated framework, named MEDIC (Mutant's Equivalence DIsCovery), incorporating these patterns has been implemented. MEDIC utilises the Static Single Assignment (SSA) form [23, 24] of a program under test to perform its analysis. As mentioned at the beginning of Section 2, the SSA form is an internal representation of a program whose basic characteristic is that every variable has exactly one definition. MEDIC obtains such a representation by leveraging the T. J. Watson Libraries for Analysis (WALA) framework [26].

3.1. T. J. Watson Libraries for Analysis (WALA)

The WALA framework [26] is a static analysis tool for the Java and the JavaScript programming languages. It can perform various analyses, control and data flow alike. The following information is obtained by the application of WALA to the program under test:

- **SSA form of the artefact under test.** WALA transforms the source code of the input program into SSA instructions which form its SSA representation. An instance of this transformation was depicted in Figure 1, which presents a code fragment of the `Commons` test subject, along with the generated SSA instructions. WALA applies several optimisations to the transformation process. Most notably, it constructs pruned SSA forms [28] and employs copy propagation [29]. Note that these optimisations are not a prerequisite for the application of MEDIC. In fact, their employment prevents MEDIC from discovering all problematic source code locations that the implemented patterns would have, in theory [1].
- **Control Flow Graph of the SSA form.** WALA divides the produced SSA instructions into basic blocks, creating an SSA-based control flow graph. MEDIC utilises this graph to examine the requirements of each implemented pattern.
- **Definitions and uses per SSA instruction.** Typically, each SSA instruction has a single definition and one or more uses. These definitions and uses refer to variables in the SSA representation of the program under

[1]This results in MEDIC's missing 7 equivalent mutants, cf. Table 3.

test and do not necessarily map to ones in its source code. By examining the second part of Figure 1, it becomes evident that even though the uses of the SSA form match the ones of the corresponding code fragment (first part of the figure), the definitions do not. For instance, the definitions of lines 3 and 4 of the source code do not directly correspond to appropriate ones in the SSA form. Instead, these definitions appear as phi statements, or more precisely as ϕ-functions, in the SSA representation. A ϕ-function is a special form of an assignment which is added to the SSA representation of a program when there are variables in its source code that are defined along two converging paths [30]. In other words, two or more definitions of a variable in the source code can reach the same use of that variable via different paths. Such a case is present in Figure 1 where variable `i` is defined at the initialisation expression of the `for` statement (`i = 0`) and it is redefined at its increment expression (`i++`). These two definitions both reach the use of `i` at the termination expression of the `for` statement (`i < strLen`) via different paths. Thus, WALA adds an appropriate ϕ-function ($v31$ `= phi` $v10$`:#0,`$v29$) before that particular use in the produced SSA representation. A similar situation pertaining to the definitions of variable `uncapitalizeNext` (lines 3, 8, 12) is treated analogously, as shown in the figure.

3.2. MEDIC's Implementation Details

The previously described data form the basis for the operation of MEDIC. In order to increase MEDIC's extensibility, these data are not used in their raw format. Instead, a custom data model has been developed that encapsulates all the necessary information for MEDIC's operation. The main advantage of such an approach is that MEDIC is decoupled from the internal mechanics of WALA and can leverage other program analysis frameworks that can potentially support different programming languages. Since the data to be modelled were graph-centric, i.e. interconnected basic blocks containing uses and definitions of variables, the utilisation of graph databases was deemed appropriate. In particular, MEDIC utilises the Neo4j graph database [31]. Graph databases model the underlying data based on *nodes* and *relationships* which can have *properties*. A property augments the corresponding model with additional information pertinent to the respective node or relationship. MEDIC defines two kinds of nodes in its data model, the `Basic Block` nodes and the `Variable` ones, and introduces three types of relationships, the `goes` relationship that connects two `Basic Block` nodes and the `uses` and `defines` relationships that connect a `Basic Block` node with a `Variable` one. Figure 6 presents an illustrative example of the aforementioned data model that belongs to the `Bisect` test subject. By examining this figure, it becomes apparent that the developed model clearly captures both control and data flow information of basic block `bb2`. This figure also presents the properties that individual nodes and relationships can possess. For instance, a `Basic Block` node can contain only one property, the `id` property, whereas a `Variable` node can contain more. Regarding the cor-

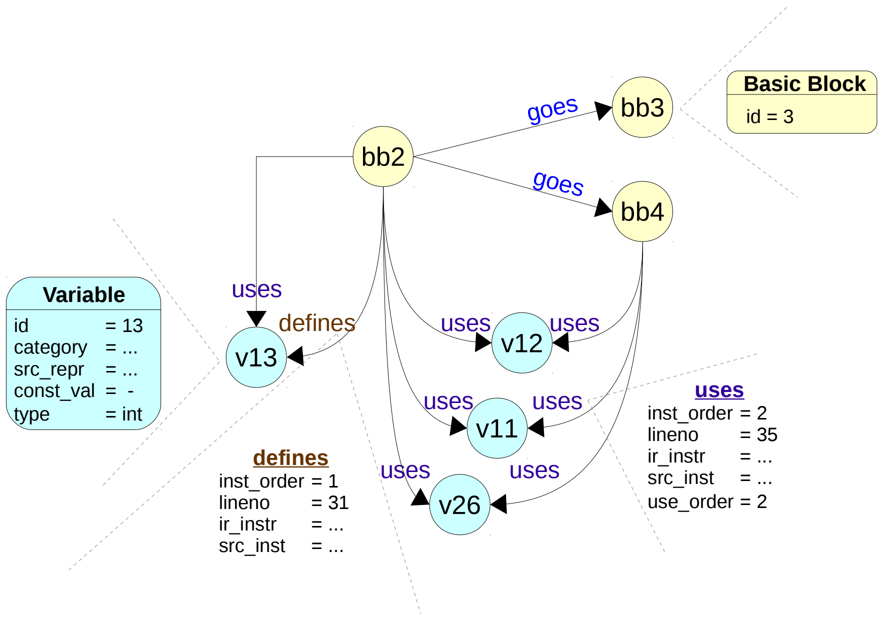

Figure 6: Example of MEDIC's underlying data model: The model is represented as a graph which has two types of nodes (`Basic block` and `Variable`) and three kinds of edges/relationships (`goes`, `uses`, and `defines`); each node and relationship can contain properties that augment the modelled information.

responding relationships, `uses` and `defines` include similar properties and the `goes` relationship includes none.

The information described by the previously presented data model suffices for the application of MEDIC, i.e. MEDIC examines this information to discover problematic code locations in the source code of the program under test that can generate equivalent or partially equivalent mutants. Note that even though this analysis is based on the SSA representation of the examined program, the discovered code locations refer to its source code.

In the following subsections, the generic algorithm that MEDIC employs for equivalent and partially equivalent mutant identification is described and the specific instantiation of this algorithm for the case of the SameLine-UD problematic pattern is detailed.

3.2.1. *MEDIC's Generic Detection Algorithm*

MEDIC implements all the problematic patterns detailed in Section 2 based on their formal definitions [21]. Algorithm 1 presents the pseudocode of MEDIC's generic algorithm for equivalent and partially equivalent mutant identification. The algorithm operates on a Neo4j database (denoted by *db*) which is based on

Algorithm 1 MEDIC's generic algorithm for equivalent and partially equivalent mutant identification.

```
   Let $p$ represent a problematic data flow pattern
   Let $q$ represent a Cypher query for $p$
   Let $db$ represent a Neo4j database modelling WALA's output

1: function EquivalentMutantDiagnosis($q$, $p$, $db$)
2:     $qres \leftarrow execute(q, db)$
3:     foreach $r \in qres$ do
4:         if isValidProblematicSituation(r,p) then
5:             return describeProblematicSituation(r,p)
6:         end if
7:     end for
8: end function
```

MEDIC's data model and encapsulates WALA's output for the program under test. MEDIC queries this database utilising the input query (denoted by q) and examines whether the obtained results satisfy the conditions imposed by the corresponding data flow pattern (denoted by p). MEDIC's detection algorithm includes three generic parts that are instantiated differently based on the employed data flow pattern:

- **Input query.** The first step in MEDIC's identification process is the execution of the specified query on the target database (line 2 of Algorithm 1). This query is written in Cypher, Neo4j's graph query language. Cypher is a declarative, SQL-inspired language that describes patterns of connected nodes and relationships in a graph. In Cypher's syntax, nodes are represented by pairs of parentheses and relationships by pairs of dashes. An example of a Cypher query that is based on MEDIC's data model and returns all variables that are used in basic block 2 of a program under test is illustrated in Figure 7. The presented query has three clauses: a `MATCH` clause that searches for the specified pattern in the underlying graph; a `WHERE` clause that filters the previously matched results based on the *id* property of the matched `Basic Block` nodes; and a `RETURN` clause that returns the value of the `src_repr`[2] property of all matched `Variable` nodes. It should be mentioned that an appropriate Cypher query has been created and incorporated into MEDIC for each implemented data flow pattern.

- **Evaluation of input query's results.** The next generic step in MEDIC's detection algorithm is the evaluation of the results obtained by the execution of the input query (line 4 of Algorithm 1). More precisely, it is examined, per obtained result, whether it describes a problematic source

[2]The `src_repr` property of a `Variable` node refers to the source code name of the modelled variable.

```
MATCH (bb:BasicBlock)-[:uses]->(var:Variable)
WHERE bb.id = 2
RETURN var.src_repr
```

Figure 7: An example Cypher query that is based on MEDIC's data model: Returns all variables that are used in basic block 2 of the program under test.

code location that could generate equivalent or partially equivalent mutants. Each considered data flow pattern necessitates several conditions, which have been formally defined [21], in order to detect equivalent and partially equivalent mutants. Thus, the first objective of this step is to examine whether these conditions hold. Additionally, certain corner cases that were revealed during the problematic patterns' empirical evaluation are also handled in this step. For instance, in the case of the UD and UR categories of patterns some code locations were identified as problematic even though they were not valid targets for the examined mutation operators (e.g. a `boolean` variable erroneously considered a valid target for AOIS). MEDIC filters out these corner cases and reports only valid problematic source code locations that could generate equivalent and partially equivalent mutants.

- **Description of problematic situations.** The final step in MEDIC's generic algorithm pertains to the description of the discovered problematic situations (line 5 of Algorithm 1). Recall that MEDIC's data model contains information regarding both the SSA representation of the program under test and its source code. These data, which can be returned by the `RETURN` clauses of the corresponding Cypher queries, are utilised by MEDIC in order to describe the detected problematic source code locations. Since problematic situations that belong to different data flow patterns require different amounts of information in order to be adequately described, MEDIC incorporates appropriate descriptive functions per problematic pattern. For instance, in order to report a problematic source code location that is detected by the SameLine-UD pattern, MEDIC utilises the name of the examined variable and the number of the source code line of the statement that includes the problematic use.

3.2.2. A Concrete Example

The previous subsection presented the generic algorithm that forms the basis for MEDIC's operation. This subsection delineates the concrete algorithm that MEDIC employs to detect problematic situations that belong to the SameLine-UD data flow pattern. The detection algorithms of the remaining problematic patterns are implemented in an analogous fashion.

As mentioned earlier, MEDIC's identification algorithm consists of three generic parts that are instantiated differently per data flow pattern. Figure 8 depicts the instantiation of these parts for the SameLine-UD problematic pattern. The first part of the figure presents the corresponding input query; the

```
MATCH (var1:Variable)<-[use:uses]-(:BasicBlock)-[def:defines]->(var2:Variable)
WHERE var1.src_repr = var2.src_repr AND use.inst_order = def.inst_order
RETURN var1.src_repr as vname, use.lineno as uline, use.src_inst as src_inst
```

(a) The input query q utilised by MEDIC for the SameLine-UD problematic pattern.

```
1: function isValidProblematicSituation($r$, 'SameLine-UD')
2:     $invalid\_cases \leftarrow [r.get('vname').concat('++'),\ \dots\ ]$
3:     foreach $invalid\_case \in invalid\_cases$ do
4:         if $r.get('src\_inst').contains(invalid\_case)$ then
5:             return $False$
6:         end if
7:     end for
8:     return $True$
9: end function
```

(b) The evaluation algorithm utilised by MEDIC for the purposes of SameLine-UD.

```
function describeProblematicSituation($r$, 'SameLine-UD')
    $des \leftarrow$ 'Problematic use and def of variable '
    $var\_name \leftarrow r.get('vname')$
    $lineno \leftarrow r.get('uline')$
    return $des.concat(var\_name).concat('\ at\ ').concat(lineno)$
end function
```

(c) The function that MEDIC utilises to describe the problematic situations that belong to the SameLine-UD pattern.

Figure 8: The instantiated parts of MEDIC's generic algorithm for the purposes of the SameLine-UD data flow pattern (see also Algorithm 1).

second part, the evaluation algorithm; and the third one, the function that reports the detected problematic source code locations. Next, these parts are presented in greater detail:

- **Input query.** As can be seen from the first part of Figure 8, the input query for the SameLine-UD problematic pattern consists of three clauses. The `MATCH` clause matches definitions and uses of variables inside the same basic blocks. The `WHERE` clause filters these matches according to the `src_repr` property of the *var1* and *var2* matched nodes and the `inst_order` property of the `use` and `def` matched relationships. The first condition of this clause ensures that the matched definition and use refer to the same variable and the second one that they belong to the same statement. Finally, the `RETURN` clause returns per matched result: the name of the corresponding variable (as `vname`); the number of the source code line that contains the problematic use and definition (as `uline`); and, the corresponding source code statement (as `src_inst`). Note that these data will be utilised to describe the discovered problematic situations of this pattern.

- **Evaluation of input query's results.** The second part of Figure 8 depicts the evaluation algorithm that examines whether the results obtained by the execution of the input query are valid problematic situations based on the SameLine-UD pattern. It should be mentioned that the `WHERE` clause of the input query covers the conditions imposed by SameLine-UD[3] – the presented algorithm handles certain corner cases that were revealed during the empirical evaluation of this pattern. For instance, the `i++` source code expression is transformed into an assignment that uses and defines the same variable in the SSA form of the program under test. Thus, it is matched by the corresponding input query. By further examining this expression, it becomes clear that it is not problematic based on the considered data flow pattern. Such invalid cases are stored in the *invalid_cases* variable of the algorithm (line 2 of the second part of Figure 8). At line 4 of the algorithm, the source code statement (accessible via the `src_inst` property of matched result r) is examined in order to determine whether it contains an invalid case. If it does, the corresponding matched result is discarded; in the opposite situation, it is returned as a valid problematic source code location that will generate equivalent mutants.

- **Description of problematic situations.** The final part of MEDIC's generic algorithm corresponds to the description of the discovered problematic situations. In the case of SameLine-UD, MEDIC utilises the name of the variable that is involved in the problematic use and definition and the number of the corresponding source code line, as can be seen from the last part of Figure 8. Recall that this information is returned by the `RETURN` clause of the respective input query. To exemplify, a problematic source code location that is detected by the SameLine-UD pattern could be described by the following: `Problematic use and def of variable` *`b`* `at line 10`. It should be mentioned that this description is intended to be human-friendly; the same information can be utilised in various ways by MEDIC or other automated frameworks.

4. Empirical Study

The primary goal of this paper is to provide insights regarding MEDIC's effectiveness and efficiency in detecting equivalent mutants. Additionally, it is examined whether MEDIC can be applied to more than one programming languages and the nature of partially equivalent mutants is investigated. This study is the first one, to the authors' knowledge, that provides evidence for automated

[3]Note that this is not the case for all problematic patterns.

stubborn mutant detection and also equivalent mutant detection in the context of different programming languages. This section begins by outlining the posed research questions and continues by detailing the corresponding empirical study and the obtained results.

4.1. Research Questions

The research questions that this study attempts to answer are summarised in the following:

RQ1. How effective is MEDIC in detecting equivalent mutants? How efficient is this process?

RQ2. Can MEDIC detect equivalent mutants in different programming languages?

RQ3. What is the nature of partially equivalent mutants? Do they tend to be killable? How easily can they be killed?

The first research question examines both the effectiveness and efficiency of MEDIC. It is important to quantify these two properties for any automated framework in order to investigate the tool's practicality. The second research question is relevant to the applicability of MEDIC and examines its cross-language nature. The final research objective aims at providing insights into the nature of partially equivalent mutants. More precisely, it explores their killability, i.e. whether or not they can be killed, and the difficulty in performing this task.

4.2. Experimental Procedure

In order to answer the above-mentioned research questions an empirical study was conducted. This study was based on test subjects of various size and complexity that are implemented in different programming languages, namely the Java and JavaScript programming languages. These two languages were chosen because they are natively supported by WALA and their application domains differ substantially.

4.2.1. RQ1: Experimental Design

To address the first research question, a set of manually analysed mutants was created. This set resulted from the application of the muJava mutation testing framework (version 3) [12, 32] to specific classes of the Java test subjects. Note that these test subjects are the ones utilised in a previous study that manually evaluated the examined data flow patterns [21]. The main difference between the present study and the aforementioned one is that this study evaluates the examined patterns based on an automated framework whereas the previous one applied them manually. Additionally, this study considers approximately twice the number of manually identified equivalent mutants.

			Manual Analysis	
Program	**Short Description**	**LOC**	**Killable**	**Equivalent**
`Bisect`	Square root calculation	36	118	17
`Commons`	Various utilities	19,583	110	28
`Joda-time`	Date and time utilities	25,909	42	4
`Pamvotis`	WLAN simulator	5,149	411	47
`Triangle`	Triangle classification	32	314	40
`XStream`	XML object serialization	16,791	127	29
TOTAL		**67,500**	**1122**	**165**

Table 1: Details of the Java test subjects utilised to answer RQ1. The first part of the table describes the corresponding test subjects and the second one presents the results of the manual analysis of the considered mutants.

Table 1 describes the Java test subjects in more detail. The table is divided into two parts. The first part presents information regarding the application domain of the studied programs and the corresponding number of source code lines. In total, six test subjects were considered, ranging from small programs to real-world libraries. The second part of the table presents the results of the manual analysis of the examined methods' mutants, i.e. the number of the equivalent and killable ones. Note that this set of mutants was based on a previous one that was utilised for the manual evaluation of the respective data flow patterns [21]. From the table, it can be seen that 1122 mutants were identified as killable and 165 as equivalent; the evaluation of MEDIC's effectiveness and efficiency is based on this set of equivalent mutants, hereafter referred to as the *manually identified set*. It must be noted that this set is one of the largest manually identified sets of equivalent mutants in the literature (cf. Table 4 in the study of Yao et al. [33]).

To further investigate the manually identified set and its relationship to the employed mutation operators, Figure 9 illustrates the proportion of equivalent mutants per mutation operator (without including the ones that did not generate such mutants). Recall that the employed mutation operators are all the method-level operators of the muJava framework. The depicted data suggest that the Arithmetic Operator Insertion Short-cut (AOIS) and the Relational Operator Replacement (ROR) mutation operators generated most of the studied equivalent mutants, with AOIS generating more than 50 per cent of them.

In order to measure MEDIC's effectiveness, MEDIC was employed to the studied test subjects and the resulting set of automatically identified equivalent mutants was compared to the manually identified one. Figure 10 presents this process. In essence, the following steps were performed per test subject:

Step 1. WALA was applied to the classes that contained the manually identified equivalent mutants of the corresponding subject.

Step 2. The output of WALA was automatically transformed into a format compatible with MEDIC's data model and then stored in a graph

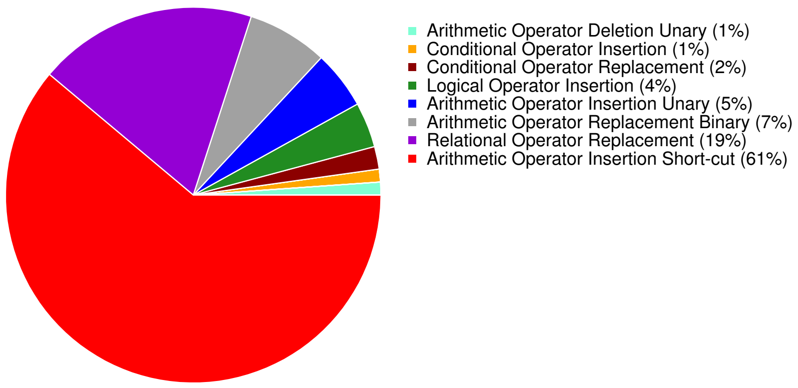

Figure 9: Contribution of each mutation operator to the manually identified set of equivalent mutants.

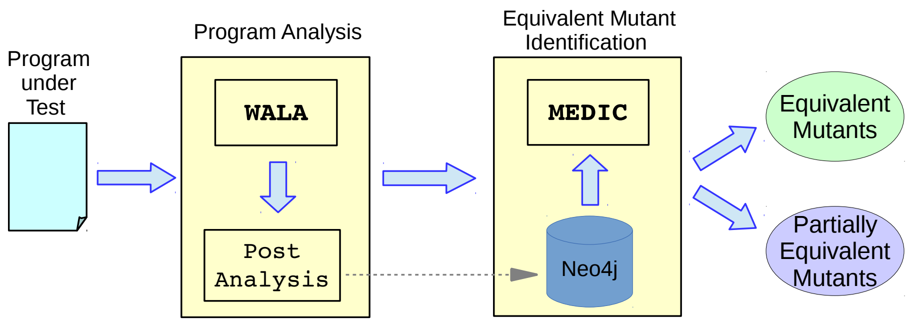

Figure 10: Application process of MEDIC: first, the program under test is analysed by WALA; second, WALA's output is transformed into a format compatible with MEDIC's data model and is stored in a Neo4j database; finally, MEDIC utilises this model to identify the equivalent and partially equivalent mutants of the examined program.

database.

Step 3. MEDIC utilised these data to detect the underlying equivalent and partially equivalent mutants.

The first step of the aforementioned process entailed the application of the WALA framework to randomly selected methods of the classes that were mutated during the manual analysis phase. Next, the output of WALA was processed in order to transform it into an appropriate format that was compatible with the data model of MEDIC. This transformation was performed automatically by a script. This adjusted output was stored in a Neo4j database. Finally, this database was given as input to the MEDIC system in order to perform its analysis. This process resulted in two sets of mutants: the set of automatically identified equivalent mutants and the set of the partially equivalent ones. The former of these sets is contrasted with the manually identified one in order to investigate MEDIC's effectiveness and the latter is utilised for the purposes of the third research question.

To study the efficiency of MEDIC, the run-time of the above-mentioned steps was measured. The experiment was conducted on a physical machine running GNU/Linux, equipped with an i7 processor (3,40 GHz, 4 cores) and 16GB of memory. Note that since the detection of partially equivalent mutants is integral to the detection of equivalent ones, the corresponding findings refer to the time that both these analyses required.

4.2.2. RQ2: Experimental Design

This research question examines whether MEDIC can detect equivalent and partially equivalent mutants in programs implemented in different programming languages (apart from Java). To answer this question, MEDIC was applied to a set of test subjects written in JavaScript. This particular language was chosen because WALA natively supports it and its application domain differs greatly from the one of Java. Table 2 presents details regarding the corresponding test subjects in a similar fashion to the Java test subjects (cf. Table 1). As can be seen, a total of four test subjects were considered, which vary in size and application domain; the latter of these being the primary reason for their selection. Since the studied research question is solely concerned with the feasibility of detecting equivalent and partially equivalent mutants in different programming languages, no exhaustive manual analysis of mutants was performed, but rather the mutants that were identified by MEDIC as equivalent or partially equivalent were manually inspected to ensure the correctness of the analysis.

The undertaken steps, for the purposes of this research question, are the same as the steps of the first one with two exceptions: at the first step, WALA was applied to randomly selected functions; and, at the second step, the transformation of WALA's output was performed in a semi-automated fashion. To elaborate on this, WALA's support for the JavaScript programming language did not cater for all the necessary information pertinent to MEDIC's data model. Specifically, it did not report the type of the examined variables. This fact is not due a limitation of the tool, but rather it is an inherent characteristic of the dynamic nature of JavaScript. To circumvent this issue, the corresponding

Program	**Short Description**	**LOC**
`dojox.calendar`	Calendar widget	8,524
`D3`	Visualisation library	11,594
`mathjs`	Mathematics library	13,112
`DateJS`	Date library	29,880
TOTAL		**63,110**

Table 2: Details of the JavaScript test subjects utilised for the purposes of RQ2.

information was added manually, based on either available comments in the corresponding source code or by inspecting the variables' usage. The rest of WALA's output was transformed via an automated script.

4.2.3. RQ3: Experimental Design

The final research question refers to the partially equivalent mutants and investigates their killability. Recall that the examined set of partially equivalent mutants is the one generated by MEDIC when applied to the Java test subjects. The first step in addressing this question was to examine whether this set consisted of killable or equivalent mutants based on the results of the manual analysis (performed for the purposes of the first research question). Next, the difficulty in killing these mutants was investigated. To quantify such an intangible property the concept of *stubborn* mutants [17] was utilised. For the purposes of this study, a stubborn mutant is considered to be a killable mutant that remains alive by a test suite that covers all the feasible branches of the control flow graph of the program under test. The same definition was also adopted in the study of Yao et al. [33]. Since a stubborn mutant cannot be killed by a branch adequate test suite, it is considered harder to kill than a non-stubborn one. Thus, if the partially equivalent mutant set consisted mainly of stubborn mutants then it would contain hard to kill mutants, or in the opposite situation, easy to kill ones.

The investigation of this research question is based on three metrics: (1) the proportion of the partially equivalent mutants that are equivalent; (2) the proportion of the partially equivalent mutants that are stubborn; and, (3) the proportion of the stubborn mutants that are partially equivalent. In order to calculate these metrics, the subsequent process was followed per test subject:

Step 1. The corresponding test subject was executed with the mutation adequate test suite to discover the remaining uncovered branches.

Step 2. The discovered branches were manually inspected to determine their (in)feasibility.

Step 3. The mutants of the test subject were executed against a set of five manually constructed branch adequate test suites in order to determine the stubborn ones.

Step 4. The stubborn partially equivalent mutants were discovered.

Step 5. The final results were averaged over the five repetitions.

The previously described process entails the identification of the branches of the considered test subjects that were not covered by the mutation adequate test suites and their manual inspection to determine their (in)feasibility. Note that the mutation adequate test suites were created for the purposes of the first research question. Next, five branch adequate test suites were constructed in order to discover the underlying stubborn mutants. These test suites were created in such a way that they did not overlap each other. The reason for utilising five branch adequate test suites instead of one is to cater for discrepancies caused by high quality test suites, i.e. branch adequate test suites that kill more mutants than other analogous ones. The next stage was to execute the mutants of each test subject with the previously constructed branch adequate test suites to determine the stubborn ones. The stubborn mutants were the ones that remained alive and were not equivalent. After the identification of the stubborn mutants, the stubborn partially equivalent ones were determined. Finally, the aforementioned metrics were calculated based on all five executions.

4.3. Empirical Findings

This section details the empirical findings of the conducted experimental study. The corresponding results are presented according to the relevant research question.

4.3.1. MEDIC's Effectiveness and Efficiency

The first research question (RQ1) investigates the effectiveness and efficiency of MEDIC in detecting equivalent mutants. Table 3 presents the corresponding findings with respect to the tool's effectiveness. The table is divided into three parts: the first part refers to the utilised test subjects; the middle part presents the automatically identified equivalent mutants per studied data flow pattern; and the last one depicts the automatically identified equivalent mutants per test subject. It should be mentioned that the results presented in the middle part of the table are grouped by the corresponding categories of the studied patterns. To exemplify, the *UD* column of the table refers to the Use-Def category of patterns and its *SL*, *SB* and *DB* columns to the SameLine-UD, SameBB-UD and DifferentBB-UD patterns, respectively. The results of the remaining patterns are presented analogously.

By examining the table, it becomes obvious that MEDIC manages to automatically identify 93 equivalent mutants for the Java test subjects, which

	Categories of Problematic Patterns									
	UD			*DD*	*UR*			*DR*		
Program	SL	SB	DB		SL	SB	DB	SB	DB	**TOTAL**
Bisect	-	4	2	-	2	-	-	-	-	8
Commons	-	-	12	-	-	-	4	-	-	16
Joda-time	-	-	-	-	-	-	4	-	-	4
Pamvotis	-	-	-	11	-	-	32	-	-	43
Triangle	4	-	-	-	-	-	14	-	-	18
XStream	-	-	4	-	-	-	-	-	-	4
TOTAL	4	4	18	11	2	-	54	-	-	**93**

Table 3: Automatically identified equivalent mutants per studied data flow pattern and (Java) test subject. The middle part of the table presents the corresponding findings grouped by the relevant categories of the considered patterns.

corresponds to a **56** per cent reduction in the number of the examined equivalent mutants that have to be manually analysed (cf. Table 1). Additionally, it can be seen that the UD and UR categories of patterns identified the most equivalent mutants, followed by the DD category of patterns. Interestingly, three problematic patterns did not identify any equivalent mutant for the Java test subjects. This fact does not limit their usefulness; the SameBB-UR problematic pattern detected equivalent mutants in the JavaScript test subjects and DifferentBB-DR identified partially equivalent ones in the examined Java programs. The only pattern that did not detect equivalent or partially equivalent mutants for all test subjects is the SameBB-DR pattern.

In order to better investigate the nature of the automatically identified equivalent mutants, Figure 11 visualises the proportion of these mutants (with respect to all studied equivalent ones) per mutation operator. It is evident that the majority of the equivalent mutants produced by the Arithmetic Operator Insertion Short-cut (AOIS) and Arithmetic Operator Replacement Binary (AORB) mutation operators were automatically identified. Recall that AOIS was the operator that produced the most equivalent mutants (see Figure 9). Additionally, equivalent mutants created by the Arithmetic Operator Insertion Unary (AOIU) and Logical Operator Insertion (LOI) were also detected, although in a smaller scale. It should be mentioned that all mutants detected by MEDIC as equivalent were indeed equivalent ones and thus, they can be weeded out safely.

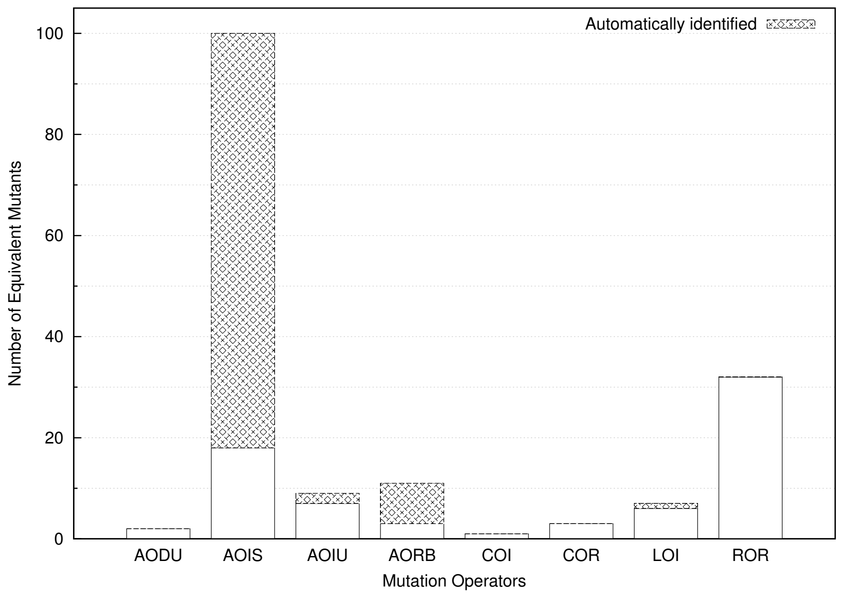

Figure 11: Proportion of the automatically identified equivalent mutants per mutation operator for the Java test subjects.

The previous results lend colour to MEDIC's effectiveness in detecting equivalent mutants. Even though these findings are encouraging, the practicality of any automated framework depends greatly on its efficiency. To provide insights regarding the performance of MEDIC, Table 4 depicts the run-time of the corresponding analysis. More precisely, it presents the run-time of the WALA framework when applied to the examined test subjects and the respective one of the MEDIC system. Note that all figures are in seconds. It can be seen that MEDIC required approximately 96 seconds to complete the equivalent mutant detection, utilising the program analysis data that had been collected in 29 seconds. Recall that the program analysis was performed only for the classes of the considered test subjects that contained the manually identified equivalent mutants. By examining the table, it becomes apparent that most of the studied classes were analysed in under 30 seconds, with `XStream`'s one being the exception. The increased run-time for this class is attributed to its internal complexity and the large number of the available combinations of uses and definitions of variables that MEDIC had to evaluate, which exceeded $150,000$!

To summarise, MEDIC managed to automatically detect **93** equivalent mutants in just **125** seconds. Assuming an equivalent mutant requires approximately 15 minutes of manual analysis [20], the analysis of the examined equivalent ones would necessitate $165 \times 15 = 41$ man-hours! The utilisation of MEDIC decreases this cost from 41 to 18 man-hours, a **56** per cent reduction in the involved manual effort, thus boosting the practical adoption of mutation.

4.3.2. Equivalent Mutant Detection in Different Programming Languages

The results of this subsection are relevant to the cross-language nature of the MEDIC framework (RQ2), i.e. its capability to detect equivalent and partially equivalent mutants in programs implemented in different programming languages. The corresponding findings are depicted in Tables 5 and 6. The former one presents the automatically identified equivalent mutants and the latter, the partially equivalent ones per studied data flow pattern and JavaScript test subject. Note that the SameBB-DR and DifferentBB-DR patterns are not included in the tables because they did not detect any such mutant. From Ta-

Program	**WALA Analysis** (secs)	**MEDIC Analysis** (secs)	**TOTAL** (secs)
Bisect	3	7	10
Commons	5	7	12
Joda-time	4	3	7
Pamvotis	10	18	28
Triangle	3	15	18
XStream	4	46	50
TOTAL (secs)	29	96	**125**

Table 4: Run-time of MEDIC's equivalent detection process for the Java test subjects (results are presented in seconds).

	Categories of Patterns							
	UD			*DD*	*UR*			
Program	SL	SB	DB	-	SL	SB	DB	**TOTAL**
nthRoot	4	-	-	-	-	4	4	12
BisectJS	-	-	8	-	4	-	-	12
StorageManager	-	-	-	-	-	2	2	4
DateJS	-	-	-	-	-	-	2	2
TOTAL	4	-	8	-	4	6	8	**30**

Table 5: Automatically identified equivalent mutants per studied data flow pattern and (JavaScript) test subject. The middle part of the table presents the corresponding findings grouped by the relevant categories of the considered patterns.

ble 5, which is structured in a similar fashion to Table 3, it can be seen that 30 equivalent mutants were automatically identified, supporting the statement that MEDIC can detected equivalent mutants in different programming languages. The findings of Table 6 lead to the same conclusion but in this case for partially equivalent mutant detection.

The results of this subsection regarding the JavaScript test subjects and the findings of the previous one regarding the examined Java programs provide clear evidence to support the cross-language nature of the MEDIC framework. MEDIC is the first tool, to the authors' knowledge, that has been empirically evaluated to achieve such a task.

4.3.3. Partially Equivalent Mutants' Killability

The final research question (RQ3) investigates the killability of the partially equivalent mutants. Table 7 presents the corresponding findings. The table is divided into two groups of columns: the first group presents the partially equivalent mutant set's proportion of equivalent and stubborn mutants; and the second one, the proportion of the stubborn mutants that are partially equivalent ones. It can be seen that 16 per cent of the partially equivalent mutant set is composed of equivalent mutants and 14 per cent of stubborn ones, on average. Further-

	Data Flow Patterns			
Program	*DifferentBB-UD*	*DD*	*DifferentBB-UR*	**TOTAL**
nthRoot	-	-	16	16
BisectJS	4	-	8	12
StorageManager	-	-	2	2
DateJS	-	-	-	0
TOTAL	4	-	26	**30**

Table 6: Automatically identified partially equivalent mutants per studied data flow pattern and (JavaScript) test subject. The middle part of the table presents the corresponding findings grouped by the relevant categories of the considered patterns.

	Part. Equivalent Mutants		Stubborn Mutants
Program	*% Equivalent*	*% Stubborn*	*% Part. Equivalent*
Bisect	5	4	11
Commons	43	0	0
Joda-time	0	65	17
Pamvotis	0	19	3
Triangle	27	17	20
XStream	0	4	5
Grand Total	16	14	6

Table 7: Proportion of equivalent and stubborn mutants w.r.t. the partially equivalent ones (column *Part. Equivalent Mutants*) and proportion of the stubborn partially equivalent mutants w.r.t. the total stubborn ones (column *Stubborn Mutants*).

more, these stubborn mutants account for 6 per cent of the total stubborn ones. Figure 12 presents a Venn diagram illustrating the detected partially equivalent mutant set (denoted as PE) and the corresponding killable (K), equivalent (E) and stubborn (S) ones, in order to better visualise the relation between them. From the depicted data, it becomes clear that the partially equivalent mutant set contains equivalent and stubborn mutants, but consists largely of non-stubborn ones, i.e. easy to kill ones.

Considering the above findings, one might decide to discard this set altogether. Such a decision would result in a 12 per cent additional reduction in the number of equivalent mutants that have to be manually analysed and a 9 per cent reduction in the number of killable ones, for the studied test subjects. Although, such an act is not without its perils, i.e. the removal of valuable mutants, the fact that only a small portion of the killable mutants needs to be targeted in order to kill the whole generated set seems to mitigate the underlying risk (see *disjoint mutants* in the work of Kintis et al. [34] and *minimal mutant sets* in the study of Ammann et al. [35]).

To recapitulate this section's findings, MEDIC managed to automatically detect equivalent and partially equivalent mutants in the Java test subjects. MEDIC identified **56** per cent of the corresponding equivalent mutants in just **125** seconds. A reduction that can be further increased by **12** per cent if the discovered partially equivalent mutant set is discarded. These results indicate that MEDIC is a very cost-effective tool, managing to detect a considerable number of equivalent mutants in a negligible amount of time. Additionally, MEDIC identified equivalent and partially equivalent mutants in the JavaScript test subjects. These findings support the cross-language nature of the tool and suggest that the aforementioned benefits are not bound to a specific programming language.

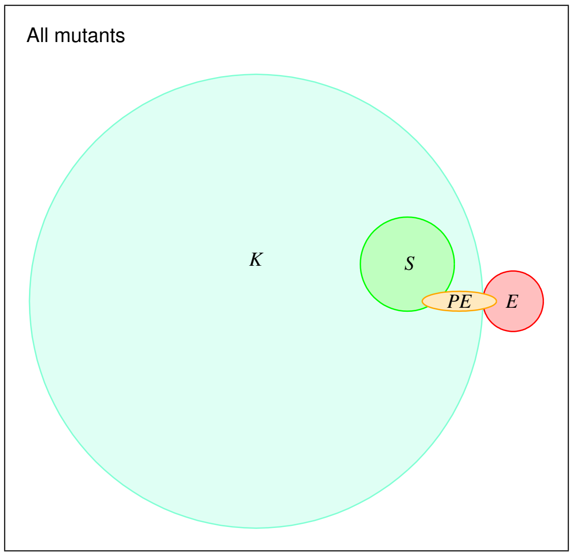

Figure 12: Venn diagram of the relation between the Partially Equivalent (PE) mutant set and the corresponding Killable (K), Equivalent (E) and Stubborn (S) ones. The size of the shapes is analogous to the cardinality of the illustrated sets.

5. Threats to Validity

All empirical studies inevitably face specific limitations on the interpretation of their results, this one is no exception. The present subsection discusses the corresponding threats and presents the actions taken to ameliorate their effects. First, threats to the construct validity are described and subsequently, the ones to the external and internal validity are discussed.

Threats to Construct Validity. This particular category is concerned with the appropriateness of the measures utilised in the conducted experiments. One threat that belongs to this category is relevant to the investigation of the difficulty in killing the partially equivalent mutants. For the purposes of this investigation, the proportion of the stubborn partially equivalent mutants was utilised. Owing to the fact that a stubborn mutant is harder to kill than a non-stubborn one, this metric is considered adequate for gaining insights regarding the nature of the partially equivalent mutants.

Threats to External Validity. This category of threats refers to the generalisability of the obtained results. No empirical study can claim that its results are generalisable in their entirety. Indeed, different studied programs or programming languages could yield different results for any empirical study. To mitigate these threats, the conducted empirical study was based on test subjects of various size and application domain and a considerable number of manually identified mutants. It should be mentioned that many of the studied programs have also been utilised in previous research studies, e.g. [19, 20, 36, 37].

Threats to Internal Validity. The threats to internal validity pertain to the correctness of the conclusions of an empirical study. One such threat is relevant to the manual analysis of the examined mutants and more precisely to the detection of equivalent ones. To control this threat, a mutant was evaluated thoroughly in order to be identified as equivalent. It should be mentioned that this threat is a direct consequence of the undecidability of the equivalent mutant problem and affects all the relevant mutation testing studies. Analogous considerations arise from the detection of stubborn mutants and, by extension, the detection of stubborn partially equivalent ones. To mitigate the effects of this threat, the corresponding process was based on five, non-overlapping branch adequate test suites. A final threat to the internal validity of the presented work is the correctness of the implementation of the WALA and MEDIC frameworks. In order to cater for this threat, the results of both tools were manually inspected to ascertain their correctness.

6. Related Work

Researchers have long battled the equivalent mutant problem, with both theoretical and empirical studies. A synopsis can be found in the works of Jia and Harman [3] and Madeyski et al. [38]. The former focuses on mutation testing in general, whereas the latter, on the equivalent mutant problem in particular. Yao et al. [33] conducted a large manual study of equivalent mutants and their relationship to stubborn ones. The obtained results suggest that although mutant equivalence is correlated with program size, mutant stubbornness is not.

At the core of the equivalent mutant problem is its undecidable nature [22], which prohibits the creation of fully automated solutions. As a consequence, the problem is solely susceptible of partial, semi-automated ones. Despite this fact, researchers have proposed several techniques to tackle this issue. These approaches can be divided into three general categories: Equivalent Mutant Detection techniques, Equivalent Mutant Reduction techniques and Equivalent Mutant Classification techniques[4].

6.1. Equivalent Mutant Detection Techniques

The techniques that belong to this category manage to correctly identify a portion of the total equivalent mutants of the program under test. One of the earliest works is due to Baldwin and Sayward [39] who suggested the idea of utilising compiler optimisation techniques to identify equivalent mutants. The key intuition of this approach is that mutants are, in a sense, optimised or de-optimised versions of the original program. In their work, Baldwin and Sayward

[4]Analogous, but not identical, terms have been utilised in the work of Madeyski et al., cf. [38].

proposed six types of compiler optimisation strategies that could identify equivalent mutants produced by certain mutation operators. These strategies were implemented and incorporated into Mothra [40], a mutation testing framework for Fortran 77 programs, by Offutt and Craft [41]. The implementation of these strategies resembles the one of MEDIC's data flow patterns: first, both implementations are based on the data flow analysis of the program under test, and second, they both operate on an intermediate representation of this program. The empirical evaluation of these strategies revealed that on average 10 per cent of the studied equivalent mutants could be automatically identified [41]. Unfortunately, most of these equivalent mutants were produced by mutation operators that are not included in the muJava mutation testing framework, utilised in the present study. Thus, a direct comparison between MEDIC and this particular method is not possible. Recently, the idea of utilising compiler optimisation techniques to identify equivalent mutants was revisited. Papadakis et al. [42] investigated whether the available compiler optimisation strategies of the GCC compiler could be leveraged to identify equivalent mutants. The obtained results suggest that 30 per cent of the studied equivalent mutants could be automatically detected. These findings imply that given a compilation framework with advanced code optimisation capabilities, like GCC, a considerable portion of the equivalent mutants of a program can be weeded out. Regrettably, such compilation frameworks are not available for all programming languages. For example, the default compiler of the Java programming language does not perform any optimisation at the compilation level; instead, all optimisations are deferred until the runtime of the program under test [43].

Offutt and Pan [16, 44] proposed another approach to tackle the equivalent mutant problem that is based on mathematical constraints. Specifically, they formally introduced a heuristic-based set of strategies in order to determine the infeasibility of constrain systems that modelled the conditions under which a mutant can be killed. If such an infeasible constraint system is detected, then the corresponding mutant can be safely identified as equivalent. The evaluation of this approach revealed that it could detect approximately 50 per cent of the examined equivalent mutants. A major difference between this approach and MEDIC is that the underlying analysis of the aforementioned strategies is based on the symbolic execution of the program under test. This implies that the performance and shortcomings of the utilised symbolic execution framework will greatly affect the effectiveness and efficiency of this method. Nica and Wotawa [45, 46] proposed a similar technique for equivalent mutant detection. Their approach creates a constraint representation of the program under test and its mutants based on the program's SSA form and attempts to generate test cases that kill them. If such test cases cannot be generated, then the examined mutant could be an equivalent one. The empirical evaluation of this technique suggests that it can be leveraged for equivalent mutant detection. It should be mentioned that, unlike MEDIC, this approach must be employed to a loop-free variant of the program under test.

Another direction in equivalent mutant identification is the utilisation of program slicing [17, 18, 47]: Voas and McGraw were the first to suggest that program slicing may ameliorate the adverse effects of the equivalent mutant problem [47]; Hierons et al. [17] suggested that the utilisation of program slicing could assist the manual analysis of equivalent mutants, a work that was later extended by Harman et al. [18] who proposed the use of dependence analysis to avoid the generation of equivalent mutants. In the study of Kintis and Malevris [21] several data flow patterns, whose presence in the source code of the program under test would lead to the generation of equivalent and partially equivalent mutants, were introduced. The present work further extends this study by implementing the aforementioned patterns and by empirically evaluating them. Finally, Offutt et al. [48] proposed several conditions that can be utilised in order to detect equivalent mutants that are generated by object-oriented, class level mutation operators.

It should be mentioned that most of the aforementioned techniques require the generation, compilation and analysis of the corresponding mutants of the program under test. In contrast, MEDIC is only applied to the original program. Thus, it can be argued that MEDIC has a clear advantage over these techniques, as far as efficiency is concerned.

6.2. Equivalent Mutant Reduction Techniques

The approaches that belong to this category attempt to reduce the number of the considered equivalent mutants. A prominent approach is based on the utilisation of higher order mutants [49], i.e. mutants that introduce more than one syntactic changes to the program under test. According to the higher order mutation's terminology, a mutant that induces one change is termed first order mutant, one that induces two is called second order mutant and so on. Polo et al. [50] proposed the creation of a set of second order mutants via the combination of the generated first order ones and the utilisation of this particular set for the purposes of mutation testing. The corresponding empirical evaluation revealed a substantial reduction in the number of the equivalent mutants that have to be manually analysed. Papadakis and Malevris [51] and Kintis et al. [34] further investigated this idea. Both studies corroborated the equivalent mutant reduction, but reported evidence of test effectiveness loss. Analogous findings were obtained by the empirical study of Madeyski et al. [38]. Thus, although these techniques succeed in reducing the number of the considered equivalent mutants and, by extension, the cost of mutation testing, they sacrifice a portion of their effectiveness. In contrast, MEDIC, like all equivalent mutant detection techniques, manage to reduce the cost of mutation testing while preserving its effectiveness.

6.3. Equivalent Mutant Classification Techniques

Recently, mutant classification techniques have been proposed as a new direction for equivalent mutant discovery [20, 37]. These approaches do not detect equivalent mutants, but rather they classify mutants as possibly killable or possibly equivalent ones based on specific characteristics of the program under test. It is postulated that mutants exhibiting these characteristics are more likely to be killable. Grün et al. [52] were the first to suggest that changes in the program behaviour between the original program and one of its mutants could indicate that the corresponding mutant is killable. Schuler et al. [53] and Schuler and Zeller [20, 54] further investigated this assumption, utilising different definitions of mutants' impact. In their first study, they examined whether the impact on dynamic invariants is a good indicator of mutants' killability and in the subsequent ones, they evaluated the classification power of the impact on coverage and on return values. The empirical results of these studies provide evidence that impact on coverage is a better indicator of mutants' killability. It should be mentioned that although this approach is implemented in the JAVALANCHE mutation testing framework [55], it is not directly comparable to MEDIC because it does not perform equivalent mutant detection. Nanavati et al. [56] took this idea a step further and proposed a killing condition that is based on the impact on coverage, i.e. they considered a mutant killed if its execution path differed from the one of the original program for at least one test execution.

Recently, Kintis et al. [19, 37] proposed the use of second order mutants to isolate first order equivalent ones. They posited that if a first order mutant is equivalent then it should not impact the output of the execution of another mutant when they are combined to form a second order one. The empirical evaluation of this approach revealed that it managed to correctly classify different killable mutants to the previous technique that utilised the impact on coverage. Thus, the combination of these two approaches resulted in a better mutant classification scheme. Finally, Kintis and Malevris [57] investigated whether mutants belonging to similar code fragments exhibit analogous behaviour with respect to their equivalence. The obtained results lend colour to this statement, suggesting that knowledge about the equivalence of one of these mutants could be utilised to classify others as possibly equivalent or possibly killable ones.

7. Concluding Remarks

It is generally known that the equivalent mutant problem impedes the adoption of mutation testing in practice. This situation is further exacerbated by its undecidable nature. Thus, the need to introduce automated solutions, albeit partial, to tackle this problem becomes apparent. In view of this, this paper introduces MEDIC, a static analysis framework for equivalent mutant detection. MEDIC implements a previously proposed set of data flow patterns whose presence in the source code of the program under test leads to the generation of equivalent and partially equivalent mutants.

MEDIC's Effectiveness and Efficiency. In order to investigate MEDIC's effectiveness and efficiency, an empirical study was conducted based on a manually analysed set of mutants from real-world programs written in the Java programming language. In particular, this set consisted of 1122 killable mutants and 165 equivalent ones. The obtained results indicate that MEDIC detected **56** per cent of the considered equivalent mutants in just **125 seconds**, a run-time cost that is far from comparable to the corresponding manual effort which approximates **23 man-hours** (93 mutants × 15 minutes).

MEDIC's Cross-language Nature. Apart from evaluating the detection power and performance of MEDIC, this study was also concerned with the cross-language capabilities of the tool, i.e. it was examined whether or not it could detect equivalent and partially equivalent mutants in programs written in different programming languages. For this reason, MEDIC was additionally applied to test subjects written in JavaScript. The respective findings suggest that MEDIC can indeed detect such mutants in the studied JavaScript programs, indicating that the tool's benefits are not confined to a specific programming language.

Killability of Partially Equivalent Mutants. The final research question that this study addresses pertains to the killability of the partially equivalent mutants, i.e. whether or not they tend to be killable, and the difficulty in performing such a task. To answer this question, (a) the proportion of the partially equivalent mutants that are equivalent, (b) the proportion of the partially equivalent mutants that are stubborn, and, (c) the proportion of the stubborn mutants that are partially equivalent were calculated. The obtained data indicate that 16 per cent of the partially equivalent mutant set is composed of equivalent mutants and 14 per cent of stubborn ones, which account for 6 per cent of the total stubborn mutants. Thus, this set consists largely of non-stubborn mutants, while containing equivalent ones. Based on these findings, one might decide to discard the partially equivalent mutant set completely, realising a total reduction of **68** per cent in the number of the equivalent mutants that have to be manually analysed. This translates into an additional 5 man-hour reduction in the involved manual effort, salvaging a total of **28 man-hours** that would have been wasted otherwise. Apart from the cross-language nature, the automated stubborn mutant detection constitutes another unique feature of MEDIC.

Future Work. An apparent future direction is the investigation of synergistic ways in which MEDIC can be combined with other equivalent mutant detection techniques. Due to MEDIC's efficiency, this combination will not be hampered by the utilisation of more than one tools. Another possible direction is the application of MEDIC to additional programming languages. Although this will entail the utilisation of a different program analysis framework, MEDIC relies on a framework-agnostic data model and thus this should not constitute a major hindrance.

Acknowledgements

The authors are grateful to the anonymous reviewers for their comments and valuable suggestions.

References

[1] R. Hamlet, Testing programs with the aid of a compiler, Software Engineering, IEEE Transactions on SE-3 (4) (1977) 279–290. doi:10.1109/TSE.1977.231145.

[2] R. DeMillo, R. Lipton, F. Sayward, Hints on test data selection: Help for the practicing programmer, Computer 11 (4) (1978) 34–41. doi:10.1109/C-M.1978.218136.

[3] Y. Jia, M. Harman, An analysis and survey of the development of mutation testing, Software Engineering, IEEE Transactions on 37 (5) (2011) 649–678. doi:10.1109/TSE.2010.62.

[4] A. Offutt, R. Untch, Mutation 2000: Uniting the orthogonal, in: W. Wong (Ed.), Mutation Testing for the New Century, Vol. 24 of The Springer International Series on Advances in Database Systems, Springer US, 2001, pp. 34–44. doi:10.1007/978-1-4757-5939-6_7.

[5] J. H. Andrews, L. C. Briand, Y. Labiche, Is mutation an appropriate tool for testing experiments?, in: Proceedings of the 27th International Conference on Software Engineering, ICSE '05, ACM, New York, NY, USA, 2005, pp. 402–411. doi:10.1145/1062455.1062530.

[6] J. H. Andrews, L. C. Briand, Y. Labiche, A. S. Namin, Using mutation analysis for assessing and comparing testing coverage criteria, IEEE Transactions on Software Engineering 32 (8) (2006) 608–624. doi:10.1109/TSE.2006.83.

[7] A. J. Offutt, J. Pan, K. Tewary, T. Zhang, Experiments with data flow and mutation testing., Tech. Rep. ISSE-TR-94-105, Department of Information and Software Systems Engineering, George Mason University (1994).

[8] A. P. Mathur, W. E. Wong, An empirical comparison of data flow and mutation-based test adequacy criteria, Software Testing, Verification and Reliability 4 (1) (1994) 9–31. doi:10.1002/stvr.4370040104.

[9] A. J. Offutt, J. Pan, K. Tewary, T. Zhang, An experimental evaluation of data flow and mutation testing, Software: Practice and Experience 26 (2) (1996) 165–176. doi:10.1002/(SICI)1097-024X(199602)26:2<165::AID-SPE5>3.0.CO;2-K.

[10] P. G. Frankl, S. N. Weiss, C. Hu, All-uses vs mutation testing: An experimental comparison of effectiveness, Journal of Systems and Software 38 (3) (1997) 235 – 253. doi:10.1016/S0164-1212(96)00154-9.

[11] N. Li, U. Praphamontripong, J. Offutt, An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage, in: Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on, 2009, pp. 220–229. doi:10.1109/ICSTW.2009.30.

[12] Y.-S. Ma, J. Offutt, Y. R. Kwon, Mujava: an automated class mutation system, Software Testing, Verification and Reliability 15 (2) (2005) 97–133. doi:10.1002/stvr.v15:2.

[13] R. A. DeMillo, A. J. Offutt, Constraint-based automatic test data generation, IEEE Trans. Softw. Eng. 17 (9) (1991) 900–910. doi:10.1109/32.92910.

[14] M. Papadakis, N. Malevris, Mutation based test case generation via a path selection strategy, Information and Software Technology 54 (9) (2012) 915 – 932. doi:10.1016/j.infsof.2012.02.004.

[15] G. Fraser, A. Zeller, Mutation-driven generation of unit tests and oracles, Software Engineering, IEEE Transactions on 38 (2) (2012) 278–292. doi:10.1109/TSE.2011.93.

[16] A. J. Offutt, J. Pan, Automatically detecting equivalent mutants and infeasible paths, Software Testing, Verification and Reliability 7 (3) (1997) 165–192. doi:10.1002/(SICI)1099-1689(199709)7:3<165::AID-STVR143>3.0.CO;2-U.

[17] R. Hierons, M. Harman, S. Danicic, Using program slicing to assist in the detection of equivalent mutants, Software Testing, Verification and Reliability 9 (4) (1999) 233–262. doi:10.1002/(SICI)1099-1689(199912)9:4<233::AID-STVR191>3.0.CO;2-3.

[18] M. Harman, R. Hierons, S. Danicic, The relationship between program dependence and mutation analysis, in: W. Wong (Ed.), Mutation Testing for the New Century, Vol. 24 of The Springer International Series on Advances in Database Systems, Springer US, 2001, pp. 5–13. doi:10.1007/978-1-4757-5939-6_4.

[19] M. Kintis, M. Papadakis, N. Malevris, Isolating first order equivalent mutants via second order mutation, in: Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on, 2012, pp. 701–710. doi:10.1109/ICST.2012.160.

[20] D. Schuler, A. Zeller, Covering and uncovering equivalent mutants, Software Testing, Verification and Reliability 23 (5) (2013) 353–374. doi:10.1002/stvr.1473.

[21] M. Kintis, N. Malevris, Using data flow patterns for equivalent mutant detection, in: Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on, 2014, pp. 196–205. doi:10.1109/ICSTW.2014.21.

[22] T. Budd, D. Angluin, Two notions of correctness and their relation to testing, Acta Informatica 18 (1) (1982) 31–45. doi:10.1007/BF00625279.

[23] B. Alpern, M. N. Wegman, F. K. Zadeck, Detecting equality of variables in programs, in: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '88, ACM, New York, NY, USA, 1988, pp. 1–11. doi:10.1145/73560.73561.

[24] B. K. Rosen, M. N. Wegman, F. K. Zadeck, Global value numbers and redundant computations, in: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '88, ACM, New York, NY, USA, 1988, pp. 12–27. doi:10.1145/73560.73562.

[25] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck, Efficiently computing static single assignment form and the control dependence graph, ACM Trans. Program. Lang. Syst. 13 (4) (1991) 451–490. doi:10.1145/115372.115320.

[26] T. j. watson libraries for analysis, last Accessed June 2015.
URL http://wala.sourceforge.net

[27] J. Maldonado, M. Delamaro, S. Fabbri, A. Silva Simão, T. Sugeta, A. Vincenzi, P. Masiero, Proteum: A family of tools to support specification and program testing based on mutation, in: W. Wong (Ed.), Mutation Testing for the New Century, Vol. 24 of The Springer International Series on Advances in Database Systems, Springer US, 2001, pp. 113–116. doi:10.1007/978-1-4757-5939-6_19.

[28] J.-D. Choi, R. Cytron, J. Ferrante, Automatic construction of sparse data flow evaluation graphs, in: Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '91, ACM, New York, NY, USA, 1991, pp. 55–66. doi:10.1145/99583.99594.

[29] M. Braun, S. Buchwald, S. Hack, R. Leia, C. Mallon, A. Zwinkau, Simple and efficient construction of static single assignment form, in: R. Jhala, K. De Bosschere (Eds.), Compiler Construction, Vol. 7791 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 102–122. doi:10.1007/978-3-642-37051-9_6.

[30] P. Briggs, K. D. Cooper, T. J. Harvey, L. T. Simpson, Practical improvements to the construction and destruction of static single assignment form, Software: Practice and Experience 28 (8) (1998) 859–881. doi:10.1002/(SICI)1097-024X(19980710)28:8<859::AID-SPE188>3.0.CO;2-8.

[31] Neo4j graph database, last Accessed June 2015.
URL http://neo4j.com

[32] J. Offutt, N. Li, The μjava home page (version 3), last Accessed June 2015.
URL http://cs.gmu.edu/~offutt/mujava/index-v3-nov2008.html

[33] X. Yao, M. Harman, Y. Jia, A study of equivalent and stubborn mutation operators using human analysis of equivalence, in: Proceedings of the 36th

International Conference on Software Engineering, ICSE 2014, ACM, New York, NY, USA, 2014, pp. 919–930. doi:10.1145/2568225.2568265.

[34] M. Kintis, M. Papadakis, N. Malevris, Evaluating mutation testing alternatives: A collateral experiment, in: Software Engineering Conference (APSEC), 2010 17th Asia Pacific, 2010, pp. 300–309. doi:10.1109/APSEC.2010.42.

[35] P. Ammann, M. E. Delamaro, J. Offutt, Establishing theoretical minimal sets of mutants, in: Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation, ICST '14, IEEE Computer Society, Washington, DC, USA, 2014, pp. 21–30. doi:10.1109/ICST.2014.13.

[36] L. Zhang, M. Gligoric, D. Marinov, S. Khurshid, Operator-based and random mutant selection: Better together, in: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, 2013, pp. 92–102. doi:10.1109/ASE.2013.6693070.

[37] M. Kintis, M. Papadakis, N. Malevris, Employing second-order mutation for isolating first-order equivalent mutants, Software Testing, Verification and Reliabilitydoi:10.1002/stvr.1529.

[38] L. Madeyski, W. Orzeszyna, R. Torkar, M. Jozala, Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation, Software Engineering, IEEE Transactions on 40 (1) (2014) 23–42. doi:10.1109/TSE.2013.44.

[39] D. Baldwin, F. Sayward, Heuristics for determining equivalence of program mutations., Tech. Rep. 276, Department of Computer Science, Yale University (1979).

[40] R. DeMillo, D. Guindi, W. McCracken, A. Offutt, K. King, An extended overview of the mothra software testing environment, in: Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on, 1988, pp. 142–151. doi:10.1109/WST.1988.5369.

[41] A. J. Offutt, W. M. Craft, Using compiler optimization techniques to detect equivalent mutants, Software Testing, Verification and Reliability 4 (3) (1994) 131–154. doi:10.1002/stvr.4370040303.

[42] M. Papadakis, Y. Jia, M. Harman, Y. LeTraon, Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique, in: Proceedings of the 2015 International Conference on Software Engineering, 2015, to appear.

[43] The Java Hotspot performance engine architecture, last Accessed June 2015.
URL http://www.oracle.com/technetwork/java/whitepaper-135217.html

[44] A. Offutt, J. Pan, Detecting equivalent mutants and the feasible path problem, in: Computer Assurance, 1996. COMPASS '96, Systems Integrity. Software Safety. Process Security. Proceedings of the Eleventh Annual Conference on, 1996, pp. 224–236. doi:10.1109/CMPASS.1996.507890.

[45] S. Nica, F. Wotawa, Using constraints for equivalent mutant detection, in: Formal Methods in the Development of Software, 2012. WS-FMDS 2012., Proceedings of the Second Workshop on, 2012, pp. 1–8. doi:10.4204/EPTCS.86.1.

[46] S. Nica, F. Wotawa, EqMutDetect – A tool for equivalent mutant detection in embedded systems, in: Intelligent Solutions in Embedded Systems, 2012. WISES 2012., Proceedings of the Tenth Workshop on, 2012, pp. 57–62.

[47] J. M. Voas, G. McGraw, Software Fault Injection: Inoculating Programs Against Errors, John Wiley & Sons, Inc., New York, NY, USA, 1997.

[48] J. Offutt, Y.-S. Ma, Y.-R. Kwon, The class-level mutants of mujava, in: Proceedings of the 2006 International Workshop on Automation of Software Test, AST '06, ACM, New York, NY, USA, 2006, pp. 78–84. doi:10.1145/1138929.1138945.

[49] Y. Jia, M. Harman, Higher order mutation testing, Information and Software Technology 51 (10) (2009) 1379 – 1393, Source Code Analysis and Manipulation, {SCAM} 2008. doi:10.1016/j.infsof.2009.04.016.

[50] M. Polo, M. Piattini, I. Garca-Rodrguez, Decreasing the cost of mutation testing with second-order mutants, Software Testing, Verification and Reliability 19 (2) (2009) 111–131. doi:10.1002/stvr.392.

[51] M. Papadakis, N. Malevris, An empirical evaluation of the first and second order mutation testing strategies, in: Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on, 2010, pp. 90–99. doi:10.1109/ICSTW.2010.50.

[52] B. Grün, D. Schuler, A. Zeller, The impact of equivalent mutants, in: Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on, 2009, pp. 192–199. doi:10.1109/ICSTW.2009.37.

[53] D. Schuler, V. Dallmeier, A. Zeller, Efficient mutation testing by checking invariant violations, in: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA '09, ACM, New York, NY, USA, 2009, pp. 69–80. doi:10.1145/1572272.1572282.

[54] D. Schuler, A. Zeller, (Un-)Covering equivalent mutants, in: Software Testing, Verification and Validation (ICST), 2010 Third International Conference on, 2010, pp. 45–54. doi:10.1109/ICST.2010.30.

[55] D. Schuler, A. Zeller, Javalanche: Efficient mutation testing for java, in: Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE '09, ACM, New York, NY, USA, 2009, pp. 297–298. `doi:10.1145/1595696.1595750`.

[56] J. Nanavati, F. Wu, M. Harman, Y. Jia, J. Krinke, Mutation testing of memory-related operators, in: Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on, 2015, pp. 1–10. `doi:10.1109/ICSTW.2015.7107449`.

[57] M. Kintis, N. Malevris, Identifying more equivalent mutants via code similarity, in: Software Engineering Conference (APSEC, 2013 20th Asia-Pacific, Vol. 1, 2013, pp. 180–188. `doi:10.1109/APSEC.2013.34`.