Using Data Flow Patterns for Equivalent Mutant Detection

Marinos Kintis
Department of Informatics
Athens University of Economics and Business
Athens, Greece
Email: kintism@aueb.gr

Nicos Malevris
Department of Informatics
Athens University of Economics and Business
Athens, Greece
Email: ngm@aueb.gr

***Abstract*—This paper introduces a set of data flow patterns that reveal code locations able to produce equivalent mutants. For each pattern, a formal definition is given and the necessary conditions implying its existence in the source code of the program under test are described. By identifying such problematic situations, the introduced patterns can provide advice on code locations that should not be mutated. Apart from dealing with equivalent mutants, the proposed patterns are able to identify specific paths for which a mutant is functionally equivalent to the original program. This knowledge can be leveraged by test case generation techniques in order not to target these paths when attempting to kill the corresponding mutants. An empirical study, conducted on a set of manually identified equivalent mutants, provides evidence regarding the detection power of the introduced patterns and unveils their existence in real world software.**

***Keywords*-data flow analysis; mutation testing; equivalent mutants**

I. Introduction

Mutation testing, originally proposed by Hamlet [1] and DeMillo et al. [2], is a very promising technique for testing software. It induces simple syntactic changes to the program under test and requires the detection of these changes by means of appropriate test data. Thus, the role of mutation in testing is twofold: it provides a test adequacy criterion, i.e., all the induced changes must be detected or proved undetectable, and during this process, it facilitates the discovery of faults.

The changes due to mutation, known as *mutants*, are based on a specific set of rules called *mutation operators*. Different mutation operators have been proposed for different programming languages and paradigms. A well-studied set is the one utilized in the Mothra mutation testing framework [3], [4], which is comprised of 22 mutation operators for the Fortran language [5]. Although studies do lend support to the fact that mutation's induced changes are representative of real faults, e.g., [6], mutation's testing efficacy is closely related to the adopted set of operators. In fact, a carefully chosen set can augment mutation's testing capabilities, whereas a poorly selected one can greatly impair its effectiveness [7], [8].

After generating the corresponding set of mutants, test cases must be created in order to distinguish the behavior of the program under test, known as the *original program*, from the one of its mutants. When such a test case exists, the mutant is said to be *killable*, i.e., the mutant can be *killed* by the corresponding test case. In such a situation, the mutant is no longer considered in the mutation testing process. When a test case fails to kill a mutant, i.e., the mutant remains *alive*, two explanations are plausible: the test is inadequate to detect the mutant or the mutant is functionally equivalent to the original program. In the latter case, the mutant is termed *equivalent*. This kind of mutants constitute a major issue in mutation testing due to the undecidable nature of the underlying problem [9], [10].

Despite being powerful, mutation lacks general acceptance in the industry. One reason is its cost: mutation requires excessive computational resources for mutant generation and execution and time consuming manual analyses for test case generation and equivalent mutant identification. Researches have attempted to reduce this cost, either by providing alternative techniques, e.g., *weak mutation* [11] and *higher order mutation* [12], or by attempting to (partially) automate the test case generation process, e.g., [13]–[15], and the equivalent mutant identification, e.g., [10], [16], [17].

This work focuses on reducing the adverse effects of the *Equivalent Mutant Problem* and is based on the work of Harman et al. [18], which proposed the use of dependence analysis [19] as a means of avoiding the generation of equivalent mutants. The present study augments the previous work by formalizing a series of data flow patterns that are able to detect problematic code locations of the original program. The given definitions utilize the *Static Single Assignment* (*SSA*) [20], [21] intermediate representation of a program which facilitates the underlying data flow analysis. The discovered code locations when mutated with specific mutation operators will result in *equivalent* or *partially equivalent* mutants. A partially equivalent mutant is considered to be a mutant that presents the same behavior as the original program on a specific subset of paths. That is, if one of these paths is to be executed with test data, the output of the mutant and the original program would be the same. It becomes apparent that discovering these problematic situations is beneficial to circumventing the equivalent mutant problem.

The contributions of this paper can be summarized in the following: **(1)** the introduction of nine problematic data flow patterns that are able to detect equivalent and partially equivalent mutants and **(2)** an empirical study, based on a set of manually identified mutants from several open source projects, which indicates that the introduced patterns exist in real world software and provides evidence regarding their detection power.

The rest of the paper is organized as follows: Section II introduces the necessary background information. Section III details the proposed data flow patterns along with appropriate examples. In Section IV, the empirical study and the obtained results are presented. Finally, Section V describes similar research studies and in Section VI concluding remarks are given.

II. Background

This section provides information about the relation between mutation testing and the employed mutation operators, discusses the equivalent mutant problem and finally, the *Static Single Assignment* (*SSA*) representation of a program.

A. Mutation Operators

The power of mutation depends largely on the employed set of mutation operators. As mentioned in the *Introduction*, a carefully chosen set can drastically increase the power of mutation. On the contrary, a poorly chosen one can render mutation useless. Thus, the design of appropriate mutation operators plays an important role in the effectiveness of mutation testing.

The notion of a mutation operator is general enough. Usually, a mutation operator resembles typical programmer mistakes, or forces the adoption of a specific testing heuristic [22]. Although this paper considers the mutation operators of the Java programming language as implemented in the `mujava` testing framework [23], [24], the introduced problematic patterns can be applied to other languages too. `Mujava` is based on the selective mutation approach [25], i.e., the tool utilizes a subset of mutation operators that are considered particularly effective. The implemented operators fall into six general categories based on the affected language operators: the *arithmetic*, *relational*, *conditional*, *shift*, *logical* and *assignment* operator categories.

The *Arithmetic Operator Insertion Short-cut* – `AOIS` is of particular interest: it inserts the *pre-* and *post*-increment and decrement arithmetic operators to valid variables of the original program, which in turn increase (or decrease) the corresponding variable *before* or *after* being used. That is, a variable affected by `AOIS` is changed *in-place*, either *before* or *after* it has been used. It should be mentioned that this mutation operator is not specific to Java: other programming languages, e.g., the C language, supports such an arithmetic operator and respective tools have implemented it as well, e.g., the `Proteum` mutation testing tool for C programs [26]. The *Use-Def* and *Use-Ret* problematic patterns, described in the next section, detect problematic situations, where the application of the `AOIS` mutation operator would result in equivalent mutants.

B. The Equivalent Mutant Problem

Equivalent mutants constitute a major cost of mutation testing. These mutants are semantically, but not syntactically, equivalent to the original program. Thus, no test case is able to discern the difference in their behavior. To worsen the situation, the equivalent mutant problem has been shown to be undecidable in its general form [9], [10]. This implies that a fully automated solution is unattainable. As a result, equivalent mutants must be removed manually from the generated set of mutants, i.e., each *live* mutant must be manually inspected and contrasted to the original program in order to establish some confidence about its equivalence.

Recent studies have shown that this mundane task is not trivial: approximately 15 minutes are required to identify a single mutant as equivalent [17], [27]. Considering this fact, along with the vast number of generated mutants, the need to create heuristics that ameliorate these adverse effects becomes apparent. To this end, the present paper introduces a series of data flow patterns that are able to detect equivalent and partially equivalent mutants, complementing previous attempts to overcome the equivalent mutant problem.

C. Static Single Assignment (SSA) Form

This paper suggests the utilization of an intermediate form of the program under test in order to detect pathogenic data flow cases. The corresponding intermediate representation is known as *Static Single Assignment* (*SSA*) form [20], [21]. Many modern compilers use the SSA form to represent internally the input program and perform various optimizations. Furthermore, several research studies have utilized the SSA form as a means of program analysis, such as [28]. The basic characteristic of the SSA representation is that each variable of the program under test is assigned exactly once. More precisely, for each assignment to a variable, that variable is given a unique name and all its uses reached by that assignment are appropriately renamed [29]. Figure 1 presents an example of a code fragment and its SSA form from the work of Cytron et al. [29], who presented the first algorithm to efficiently translate a program to its SSA form. Note that this translation restricts the definition points of a variable: after the translation, each variable is bound to have only one definition point, instead of multiple in the original program (cf. variables v_1, v_2 and v of Figure 1). This fact enables a more compact representation of the underlying data flow information and facilitates powerful code analysis and optimization techniques.

III. Equivalent Mutants and Data Flow analysis

This work focuses on detecting equivalent mutants by leveraging the data flow information of the program under

$v = 4$	$v_1 = 4$
$x = v + 5$	$x_1 = v_1 + 5$
$v = 6$	$v_2 = 6$
$y = v + 7$	$y_1 = v_2 + 7$
(a)	(b)

Figure 1. Example of a code fragment and its SSA Form (adapted from [29]).

test. To this end, the SSA representation of the program is utilized. This is believed to lead to a more powerful analysis and a clearer definition of the conditions that need to hold for a particular situation in order to be pathogenic.

Four groups of problematic situations are being addressed, the *Use-Def*, the *Use-Ret*, the *Def-Def* and the *Def-Ret* problematic patterns. The first two categories refer to mutation operators that perform in-place changes, and the remaining ones, to mutation operators that affect variable definitions. Note that the search for these patterns is performed to the source code of the original program. Therefore, these patterns can be used in various ways: they can be incorporated in mutation testing frameworks to avoid the generation of equivalent mutants that belong to the identified problematic patterns or, if the corresponding mutants have already been generated, they can be of assistance in detecting equivalent or partially equivalent mutants, a term described shortly. Before elaborating on these categories, the necessary definitions are given below.

A. Definitions

This subsection introduces a set of functions that will be later used in the description of the problematic data flow cases. First, *control flow* and *data flow* related definitions are described and subsequently, the corresponding *mutation* related ones are depicted.

1) Control Flow Related Definitions:

$reach(p, s, t)$ Denotes that path p connects nodes s and t, starting at node s and ending at node t. Note that s and t need not be the starting and ending nodes of the corresponding control flow graph.

$frstIns(i)$ A function that returns the first instruction of the node that contains instruction i.

$bb(i)$ A function that returns the node which contains instruction i.

$isExit(n)$ Denotes that node n contains a statement that forces the program to exit.

The aforementioned functions use information from the *Control Flow Graph – CFG* of the program under test. Note that the terms *node* and *basic block* are used interchangeably; the nodes of a CFG are the basic blocks of the respective program. For the purposes of this study: a *path* is a sequence of two or more nodes, where each pair of adjacent nodes represents an edge of the corresponding CFG and an *instruction* is considered to be a simple statement that belongs to a single line and one basic block.

2) Data Flow Related Definitions:

$def(v_x)$ A function that returns the instruction that defines the variable v_x.

$use(n, v_x)$ A function that returns the instruction that contains the nth use of variable v_x.

$defAftr(i, v_x)$ A function that returns the instruction of the first definition of variable v of the original program, *after* instruction i. The scope of this function is limited to the basic block that contains instruction i.

$useBfr(i, v_x)$ A function that returns the instruction of the first use of variable v of the original program, *before* instruction i. The scope of this function is limited to the basic block that contains instruction i.

$useAfr(i, v_x)$ A function that returns the instruction of the first use of variable v of the original program, *after* instruction i. The scope of this function is limited to the basic block that contains instruction i.

$defClr(p, v_x)$ Denotes that except from the starting and ending nodes of path p, no other node can include a definition of variable v of the original program, i.e., no *intermediate* node of p defines v.

$useClr(p, v_x)$ Denotes that except from the starting and ending nodes of path p, no other node can include a use of variable v of the original program, i.e., no *intermediate* node of p uses v.

As mentioned in the beginning of this subsection, the data flow analysis is based on the SSA representation of the program under test. Although the definitions of the last five functions refer to the original program, the desired behavior is achieved by utilizing information from the corresponding SSA form.

3) Mutation Related Definitions: This paper introduces the concept of a *partially equivalent* mutant, that is, a mutant that is equivalent to the original program for a specific subset of paths. Recall that this study refers to strong mutation testing, thus, the following definitions will be tailored to that particular approach. More formally (case of a **partially equivalent mutant**):

$\exists\ m \in Muts:\ \exists\ p \in Paths,$

- $\forall\ t \in T_p : output(orig, t) = output(m, t)$

where p refers to a path of the CFG of the program under

test that contains the basic block of the mutated statement, `Muts` to the mutated versions of the examined program, T_p to the set of test cases that cause the execution of p and `output(prog, t)` to a function that returns the output of program $prog$ with input t. Thus, this definition states that there is at least one path of the program under test, whose execution will not reveal the induced change in program m, i.e., the mutated version m is equivalent to the original program with respect to path p.

Given the definition of a partially equivalent mutant, the *equivalence* of a mutant can be defined as follows (case of an **equivalent mutant**):

$\exists\ m \in Muts:\ \forall\ p \in Paths,$

- $\forall\ t \in T_p : output(orig, t) = output(m, t)$

that is, the execution of each path p with all available test data will result in the same output for the original program and mutant m.

It should be mentioned that the partial equivalence of a mutant does **not** indicate that a mutant is equivalent. It indicates that the path for which the mutant is partially equivalent should not be used to kill that mutant.

B. Use-Def (UD) Problematic Pattern

This category detects equivalent mutants that are created by mutation operators that induce in-place changes. As mentioned in the previous section, these operators change the value of the variable they affect. An example of such an operator is the `AOIS` mutation operator, which inserts the *pre-* and *post-* increment and decrement arithmetic operators to valid targets. An instance of the application of this operator is depicted in Figure 2, which presents the source code of the `Bisect` program (along with its corresponding basic blocks). The application of `AOIS` operator at line 10 results in eight mutants that affect variables `x` and `M`. Examining the two subsequent lines of line 10, it becomes apparent that the depicted mutants are equivalent. This example reveals a data flow pattern that is able to detect equivalent mutants generated by the application of the `AOIS` mutation operator: a use of a variable on the right part of an assignment statement and the same variable as the target of the assignment. The *UD* category of patterns detects several problematic situations, along with this one, that are caused by the application of the `AOIS` mutation operator, or another operator that changes the affecting variable in-place.

The *UD* category is comprised of three problematic patterns, the *SameLine-UD*, the *SameBB-UD* and the *DifferentBB-UD* patterns. All three attempt to reveal equivalent or partially equivalent mutants by searching the source code of the original program for uses of variables that reach a definition and can be mutated by a mutation operator that supports in-place changes. As a special case to the *UD* problematic patterns, the *Use-Ret* (*UR*) pattern is also introduced; a pattern that searches the source code of the

```
1:  sqrt (double N) {
2:      double x = N;                                  // BB:1
3:      double M = N;
4:      double m = 1;
5:      double r = x;
6:      double diff = x * x - N;
7:      while (ABS(diff) > mEpsilon) {                 // BB:2
8:          if (diff < 0) {                            // BB:3
9:              m = x;                                 // BB:4
10:             x = (M + x) / 2;
            Δ x = (M + x++) / 2
            Δ x = (M + x--) / 2
11:         }
12:         else if (diff > 0) {                       // BB:5
13:             M = x;                                 // BB:6
14:             x = (m + x) / 2;
15:         }
16:         diff = x * x - N;                          // BB:7
17:     }
18:     r = x;                                         // BB:8
19:     mResult = r;
20:     return r;
21: }
```

Figure 2. The `Bisect` test subject.

examined program for uses that do not reach another use. Thus, the application of a mutation operator that supports in-place changes to the corresponding code locations would result in the creation of equivalent mutants. Analogously to the patterns of the *UD* category, the *UR* category consists of the *SameLine-UR*, the *SameBB-UR* and the *DifferentBB-UR* problematic patterns.

1) SameLine-UD Problematic Pattern: The *SameLine-UD* problematic pattern detects equivalent mutants that affect a variable that is being used and defined at the same statement. More formally, a problematic situation arises when the following conditions hold (case of the **SameLine-UD** problematic pattern):

$\exists\ v_k, v_j \in SVars,\ k \neq j,\ i > 0:$

- $use(i, v_k) = def(v_j)$
- $\nexists\ m > i : use(m, v_k) = use(i, v_k)$

where `SVars` is the set of variables of the SSA representation of the program under test that refers to the same variable of the original program (cf. variables v, v_1, v_2 of Figure 1). The first condition states that an instruction must exist that uses and defines the same variable with respect to the original program and the second one necessitates that the ith use of variable v_k is the last one in the corresponding instruction. It must be mentioned that variable v_k should be a valid target for a mutation operator that performs in-place changes, otherwise, this pattern does not constitute a

problem. An example of this problematic situation was given in the beginning of this section and involved the two mutants of line 10 of Figure 2, as depicted in the two subsequent lines.

A special case of this pattern is the *SameLine-UR* problematic situation, which involves uses of variables at `return` statements, or statements that cause the program to exit in general. The conditions that must be satisfied for the manifestation of this situation are the following (case of the **SameLine-UR** problematic pattern):

$\exists\ v_k \in SVars,\ i_\epsilon \in ExitIns,\ i > 0:$

- $use(i, v_k) = i_\epsilon$
- $\nexists\ m > i : use(m, v_k) = i_\epsilon$

where $ExitIns$ is the set of instructions that cause the program to exit. The first condition states that the instruction of the ith use of variable v_k must be an instruction that can exit the program and the next condition requires the ith use to be the ultimate one in the corresponding instruction.

An example of this problematic situation is also present in Figure 2 at line 20, where variable `r` is used in a `return` statement. The application of the `AOIS` operator will result in two equivalent mutants[1]. These mutants can be detected automatically by the present pattern.

2) SameBB-UD Problematic Pattern: This pattern resembles the previous one, but in this case the search focuses on the same basic block. In order for this pattern to be present in the source code of the original program the following must hold (case of the **SameBB-UD** problematic pattern):

$\exists\ v_k, v_j \in SVars,\ k \neq j,\ i > 0:$

- $bb(use(i, v_k)) = bb(def(v_j))$
- $\nexists\ m > i : bb(use(m, v_k)) = bb(use(i, v_k))$
- $defAftr(use(i, v_k), v_k) = def(v_j)$

The first condition states that the definition of v_j and the use of v_k must belong to the same basic block. Recall that v_j and v_k are variables of the SSA representation of the program under test and refer to variable v of the corresponding program. The second condition denotes that the ith use of v_k is the last one before the definition of v_j, in the respective basic block. The ultimate condition requires instruction $def(v_j)$ to be the first definition of variable v of the original program after instruction $use(i, v_k)$.

An example of this pattern is presented in part (a) of Figure 1: variable v is used in the second line of the code fragment and in the third is defined. Thus, mutating v with the `AOIS` operator will result in two equivalent mutants[2]. Examining the SSA representation of this code fragment (part (b)), it is evident that the aforementioned conditions do hold, thus, the corresponding equivalent mutants can be detected by the *SameBB-UD* problematic pattern.

As a special case of this pattern, consider line 18 of Figure 2: it can be seen that the application of the `AOIS` operator will result in two equivalent mutants[3]. This is due to the fact that variable `x` is never used between lines 18 and 20. This problematic situation leads to the *SameBB-UR* problematic pattern, which searches the source code of the original program for uses of variables that belong to a basic block that can exit the program. More formally (case of the **SameBB-UR** problematic pattern):

$\exists\ v_k \in SVars,\ i > 0:$

- $isExit(bb(use(i, v_k)))$
- $\nexists\ m > i : bb(use(m, v_k)) = bb(use(i, v_k))$
- $\nexists\ v_j \in SVars : defAftr(use(i, v_k), v_k) = def(v_j)$

where the first condition necessitates the existence of a use inside a basic block that can exit the program, the second condition is the same as in the case of the *SameBB-UD* pattern and the last one requires the absence of any definitions of the used variable after the using instruction. Although the last condition is not required, its exclusion would cause instances of the *SameBB-UD* pattern to be treated as instances of this particular one. For the same reasons, it is added in analogous cases of the *DD* problematic pattern.

3) DifferentBB-UD Problematic Pattern: The final problematic pattern of this category manages to detect partially equivalent mutants and in specific cases, equivalent ones. This category searches for uses of variables that can reach definitions by different paths, i.e., a variable is used in one basic block and defined in another and one or more paths connect these two basic blocks. In order for this pattern to constitute a problem, the intermediate nodes (cf. Section III-A) of at least one of the connecting paths must not include any uses or definitions of the respective variable. Such being the case, the change on the used variable (performed by a mutation operator that supports in-place changes) cannot be revealed on the corresponding path because the variable is redefined prior to being used. More formally (case of the **DifferentBB-UD** problematic pattern):

$\exists\ v_k, v_j \in SVars,\ k \neq j,\ i > 0,\ p \in Paths:$

- $reach(p, bb(use(i, v_k)), bb(def(v_j)))$
- $\nexists\ m > i : bb(use(m, v_k)) = bb(use(i, v_k))$
- $\nexists\ v_q \in SVars : def(v_q) = defAftr(use(i, v_k), v_k)$
- $defAftr(frstIns(def(v_j)), v_j) = def(v_j)$
- $\nexists\ v_q \in SVars,\ n > 0 : useBfr(def(v_j), v_j) = use(n, v_q)$
- $useClr(p, v_k)$

[1]Mutants: return r++ and return r--.

[2]Mutants: x = v++ + 5 and x = v-- + 5.

[3]Mutants: r = x++ and r = x--.

- $defClr(p, v_k)$

where `Paths` refers to the paths of the CFG of the program under test.

Elaborating on these conditions: the first one requires the existence of a path connecting the node that uses variable v_k and the one that defines variable v_j, i.e., the nodes that use and define variable v of the original program. The next two conditions refer to the basic block that contains the instruction of the ith use of variable v_k (as returned by $use(i, v_k)$), hereinafter referred to as the *using* basic block. The first one states that the ith use of v_k must be the last use of that variable in the *using* basic block, i.e., there are no other uses of v_k after instruction $use(i, v_k)$. The other one prohibits the existence of a definition of variable v of the original program after the ith use of v_j (belonging to the *using* basic block). The fourth and fifth conditions refer to the basic block that contains the instruction that defines variable v_j (as returned by $def(v_j)$), hereinafter referred to as the *defining* basic block. The first of them states that $def(v_j)$ is the first instruction of the *defining* basic block that defines variable v of the original program, i.e., there is no prior definition of variable v in the *defining* basic block. The next one necessitates that there are no uses of variable v of the original program before the $def(v_j)$ instruction in the *defining* basic block. Finally, the last two conditions require the intermediate nodes of path p not to include a use or definition of variable v of the original program. Recall that path p connects the *using* and the *defining* basic blocks.

As mentioned earlier, this pattern identifies partially equivalent or equivalent mutants. In the case of partially equivalent mutants, the aforementioned conditions hold for at least one such path, i.e., the mutant cannot be killed by targeting the corresponding path, but there are other paths for which the previously mentioned conditions do not hold. Thus, the mutant can be killable and by targeting those paths a killing test case may be produced. It must be mentioned that the present study does not consider the feasibility of these paths, i.e., if it is possible to execute them with test data, it is solely concerned with the existence of such paths, i.e., with the possibility of a mutant to be killable. In the case of equivalent mutants, the corresponding conditions hold for all paths connecting the respective basic blocks and thus, there is no path to consider for killing them.

An example of a problematic situation that falls into the described category is presented in Figure 2 and more precisely at lines 12 and 16, where variable `diff` is being used and defined, accordingly. The application of the `AOIS` mutation operator at line 12 would result in two equivalent mutants[4] due to the fact that `diff` is not used between lines 12 and 16, thus, the in-place change cannot be detected. This situation is managed by the *DifferentBB-UD* problematic pattern, which successfully identifies these mutants as equivalent.

A special case of this pattern, the *DifferentBB-UR* problematic pattern, investigates pathogenic situations that arise when a use of a variable does not reach another use, either because there is none or due to a `return` statement, or a statement that can cause the program to exit in general. In this specific situation, the following conditions need to hold in order to constitute a problematic situation (case of **DifferentBB-UR** problematic pattern):

$\exists\ v_k \in SVars,\ i_\epsilon \in ExitIns,\ i > 0,\ p \in Paths$:

- $reach(p, bb(use(i, v_k)), bb(i_\epsilon))$
- $\nexists\ m > i : bb(use(m, v_k)) = bb(use(i, v_k))$
- $\nexists\ v_q \in SVars : defAftr(use(i, v_k), v_k) = def(v_q)$
- $\nexists\ v_q \in SVars : bb(def(v_q)) = bb(i_\epsilon)$
- $\nexists\ v_q \in SVars,\ n > 0 : bb(use(n, v_q)) = bb(i_\epsilon)$
- $useClr(p, v_k)$
- $defClr(p, v_k)$

where $ExitIns$ is the set of instructions that cause the program to exit. These conditions are in accordance to the ones of the *DifferentBB-UD* problematic pattern. The first one requires the existence of a path connecting the basic blocks that contain the use of variable v_k and an instruction that forces the program to exit. The next two state that, at the basic block containing the $use(i, v_k)$ instruction, the ith use of v_k is the last one and that there is no definition of variable v of the original program after the corresponding use. The following two require that there are no definitions or uses of variable v of the original program at the basic block containing instruction i_ϵ, respectively. Finally, due to the last two conditions the intermediate nodes of path p must not contain definitions or uses of variable v.

C. Def-Def (DD) Problematic Pattern

This category identifies problematic code locations based on subsequent definitions of a variable. Expanding on this, if a definition can reach another definition of the same variable and there is no intermediate use of that variable, then any change affecting the first definition cannot be revealed due to the fact that the variable is redefined prior to being used. Thus, the corresponding mutants are bound to be partially equivalent or, in specific cases, equivalent. It should be clarified that this category is not restricted to a particular type of mutation operators, e.g., one that supports in-place changes; any mutation operator that can affect the right part of an assignment statement (or a variable definition in general) constitutes a possible target for the respective patterns.

An example of a problematic situation that is detected by the present pattern involves lines 3 and 13 of Figure 2, where the variable `M` is being defined. The only use of the variable is in line 10 of basic block 4. Thus, mutants affecting the

[4]Mutants: else if (diff++ $>$ 0) and else if (diff-- $>$ 0).

definition of line 3 are bound to be partially equivalent for all paths from node 1 that include node 6 and do not include node 4. Recall that the equivalent mutant problem, in this paper, is being addressed for the case of strong mutation, i.e., a mutant is considered equivalent if its output is the same as the one of the original program for every possible input. Furthermore, it should be mentioned that the partial equivalence of a mutant does not imply that the mutant will eventually be equivalent. Instead, this knowledge should be utilized in order to avoid targeting those paths that will not yield a killing test case for the examined mutant.

Unlike the *UD* patterns, this category is solely concerned with problematic situations involving code locations that belong to different basic blocks. Although, two definitions of the same variable can belong to the same basic block, if these definitions are not accompanied with intermediate uses, they constitute a problematic situation on their own. A not so obvious problem arises when one definition of a variable can reach another one (of the same variable) along a path that does not contain a node that uses that variable. Mutants affecting such definitions would be partially equivalent and the corresponding paths would never result in a test case that is able to kill the respective mutants. Furthermore, if all paths connecting the basic blocks of the two definitions do not include a node that uses the defined variable, then the above mentioned mutants can be safely classified as equivalent. Thus, the identification of such problematic cases will result in saving valuable resources, human and machine alike.

More formally, the necessary conditions that need to hold in order to detect the described problematic situations are (case of **DD** problematic pattern):

$\exists\ v_k, v_j \in SVars,\ k \neq j,\ p \in Paths$:

- $reach(p, bb(def(v_k)), bb(def(v_j)))$
- $\nexists\ v_q \in SVars : defAftr(def(v_k), v_k) = def(v_q)$
- $\nexists\ v_q \in SVars,\ n > 0 : useAfr(def(v_k), v_k) = use(n, v_q)$
- $defAftr(frstIns(def(v_j)), v_j) = def(v_j)$
- $\nexists\ v_q \in SVars,\ n > 0 : useBfr(def(v_j), v_j) = use(n, v_q)$
- $useClr(p, v_k)$
- $defClr(p, v_k)$

These conditions are analogous to the ones of the *DifferentBB-UD* problematic pattern, except that in this case two definitions of a variable are considered and not a single use and a definition. The first condition requires the existence of a path connecting the node that defines variable v_k and the one that defines variable v_j. The next two conditions state that the $def(v_k)$ instruction is the last definition of variable v of the original program in the corresponding basic block and that there are no uses of v between the defining instruction and the end of the basic block. The remaining conditions are the same as in the case of the *DifferentBB-UD* pattern and are restated here in order for this section to be self-contained: they state that the $def(v_j)$ instruction is the first definition of variable v of the original program in the corresponding basic block and that there are no uses of v between the first instruction of the basic block and the defining instruction. Finally, the last two conditions require path p not to contain any intermediate nodes that use or define variable v of the original program.

Another case that belongs to this particular pattern is the *Def-Ret* (*DR*) problematic situation. This pattern resembles the previously described one, but instead of searching for two definitions without an intermediate use, it searches for a definition that cannot reach a use, either because there is none or due to a statement that forces the program to exit. This situation can manifest in the same or different basic blocks of a program under test. In the former of these, the *SameBB-DR* problematic pattern, the defining statement belongs to a basic block that can exit the program. In order for this pattern to become problematic the following conditions must hold (case of **SameBB-DR** problematic pattern):

$\exists\ v_k \in SVars$:

- $isExit(bb(def(v_k)))$
- $\nexists\ v_q \in SVars,\ n > 0 : useAfr(def(v_k), v_k) = use(n, v_q)$
- $\nexists\ v_j \in SVars : defAftr(def(v_k), v_k) = def(v_j)$

where the first condition states that the instruction that defines variable v_k belongs to a basic block that can exit the program, the next one requires that there are no uses of variable v of the original program after the corresponding defining instruction and the last one prohibits the existence of another defining instruction of the considered variable after instruction $def(v_k)$.

Finally, in the case of the *DifferentBB-DR* pattern, this problematic situation manifests in different basic blocks, i.e., one basic block which defines a variable is connected by at least one path (that does not contain any intermediate nodes that use the corresponding variable) to a basic block that can exit the program. More formally (case of **DifferentBB-DR** problematic pattern):

$\exists\ v_k \in SVars,\ i_\epsilon \in ExitIns,\ p \in Paths$:

- $reach(p, bb(def(v_k)), bb(i_\epsilon))$
- $\nexists\ v_q \in SVars : defAftr(def(v_k), v_k) = def(v_q)$
- $\nexists\ v_q \in SVars,\ n > 0 : useAfr(def(v_k), v_k) = use(n, v_q)$
- $\nexists\ v_q \in SVars : bb(def(v_q)) = bb(i_\epsilon)$
- $\nexists\ v_q \in SVars,\ n > 0 : bb(use(n, v_q)) = bb(i_\epsilon)$
- $useClr(p, v_k)$
- $defClr(p, v_k)$

where $ExitIns$ is the set of instructions that cause the program to exit. These conditions are analogous to the ones of the *DD* problematic pattern. The first one requires the existence of a path that connects the basic block that contains the definition of variable v_k and one that can exit the program. The next two state that, at the defining basic block, the $def(v_k)$ instruction is the last one that defines variable v of the original program and that there is no use of the variable in the corresponding basic block after the definition. The following two require that there are no definitions or uses of variable v of the original program at the node containing instruction i_ϵ, respectively. Finally, the last two conditions require path p not to contain any intermediate nodes that use or define variable v of the original program.

IV. Empirical Study

This work tackles the equivalent mutant problem by identifying problematic situations in the data flow information of the program under test that can lead to the generation of equivalent or partially equivalent mutants. In order to investigate the detection power of the introduced patterns, as well as their presence in real world software, an empirical study was conducted on a set of manually classified mutants, hereinafter referred to as the *control mutant set*. In the next subsections, details about the experimental setup, the considered test subjects and the obtained results are given.

A. Experimental Setup

1) Control Mutant Set: The control mutant set was created for a previous study [30] which proposed the use of code similarity as a means of identifying more equivalent mutants. To this end, the concept of *mirrored mutants* was introduced, i.e., mutants that belong to similar code fragments. Figure 3 presents the corresponding selection process (taken from [30]): for each test subject, all similar code fragments were detected. Next, two similar code fragments (i.e., a *clone pair*) were randomly chosen and their mutants were generated by applying all method-level mutation operators of `mujava` [23], [24] (version 3 of the tool was utilized), if their number was greater than 40, the corresponding *mirrored mutants* were generated. The aforementioned process was repeated, until a suitable match was found. In essence, the control mutant set of the present study is comprised of the examined *mirrored mutants* of the previous one. Since their selection was not based on any specific data flow information, this set is considered adequate for the purposes of this experiment.

```
1: foreach test subject ts
2:     foreach clone pair of ts
3:         select a random clone pair cp
4:         generate the mutants of cp
5:         if (mutants > 40)
6:             create the mirrored mutants of cp
7:             return the mirrored mutants
8:         end if
9:     end foreach
10: end foreach
```

Figure 3. Selection process of the control mutant set (taken from [30])

The control mutant set is comprised of 747 manually classified mutants from several open source projects and includes 663 killable and 84 equivalent mutants. The left part of Table I presents information about the corresponding test subjects: in the first three columns, the name of the test subjects with a short description of their usage and the corresponding lines of code are given. In the fourth column, the number of the considered equivalent mutants is depicted. Since this study targets the equivalent mutant problem and the introduced patterns do not yield false positive results, only the equivalent mutants of the control mutant set are utilized. Studying the respective killable ones would be valuable in the case of partially equivalent mutants – this task, though, is left for future research.

2) Experimental Procedure: In order to investigate the detection power of the introduced problematic patterns, the mutated location of each equivalent mutant was manually investigated in order to determine whether or not this particular location can be identified as problematic by the proposed patterns, i.e., if the conditions of at least one of the introduced patterns are satisfied. Note that the manual investigation is performed solely for the purposes of this experiment. The proposed data flow patterns can be incorporated in a automated framework and, thus, can facilitate the automatic detection of equivalent mutants.

B. Experimental Results

The empirical results of the conducted study are presented in the right part of Table I. The last two columns of the table illustrate the number of equivalent mutants that can be automatically detected and the achieved reduction in the number of mutants that have to be manually analyzed. It can be seen that the introduced patterns are able to automatically detect 58 out of 84 equivalent mutants and can decrease the number of equivalent mutants that have to be manually analyzed by approximately 70%. In other words, only 30% of the examined equivalent mutants have to be manually analyzed. A clear outlier is the `XStream` test subject, whose equivalent mutants could not be detected by the presented patterns. Upon inspection, this particular program did not contain any of the introduced data flow patterns.

It should be mentioned that the `AOIS` mutation operator accounted for 58 equivalent mutants of the control mutant set and these mutants were the ones that were automatically detected. This fact justifies the introduction of the *UD* and *UR* categories of problematic patterns and provides evidence about the extent to which `AOIS` is responsible for the generation of equivalent mutants.

Table I
EMPIRICAL RESULTS W.R.T. DATA FLOW PATTERNS

Program	Description	LOC	Equivalent Mutants (*manually identified*)	Equivalent Mutants (*automatically detected*)	Cost Reduction
`Bisect`	Square root calculation	36	4	4	100%
`Commons`	Various utilities	19,583	28	16	57%
`Joda-time`	Date and time utilities	25,909	4	4	100%
`Pamvotis`	WLAN simulator	5,149	10	8	80%
`Triangle`	Returns triangle type	32	30	26	87%
`Xstream`	XML object serialization	16,791	8	0	0%
TOTAL		**67,500**	**84**	**58**	**69%**

Finally, it is noted that most of the undetected equivalent mutants are caused by the *Relational Operator Replacement*–`ROR` mutation operator[5], i.e., a mutation operator that cannot be handled solely by data flow information.

V. RELATED WORK

This study is not the first attempt to tackle the equivalent mutant problem. Many researchers have introduced techniques and heuristics to ameliorate its negative effects. In 1979, Baldwin and Sayward [31] proposed the utilization of compiler optimization techniques to detect equivalent mutants, an idea later subsumed by the use of mathematical constraints, suggested by Offutt and Pan [10], [32]. The obtained empirical results suggested that approximately 50% of the examined equivalent mutants could be automatically detected. It should be noted that the present work does not subsume the previously described ones; on the contrary, it complements them.

Apart from the above mentioned techniques, Hierons, Harman and Danicic [16] proposed the utilization of program slicing to assist the manual analysis of equivalent mutants. An extension of this work suggested the use of dependence analysis as a means of avoiding the creation of equivalent mutants [18]. As already discussed, the present study improves upon this particular work by providing concrete, formalized cases where dependence analysis is utilized to detect equivalent and partially equivalent mutants.

Another promising direction in tackling the equivalent mutant problem is the use of mutant classification techniques [17], [33]. Although, these approaches do not prove the equivalence of the examined mutants, the obtained experimental results provide evidence about their appropriateness.

Finally, another recent study suggested the use of code similarity for discovering more equivalent mutants [30]. This particular study introduced the concept of *mirrored mutants*, mutants that belong to similar code fragments, and examined whether or not they exhibit analogous behavior regarding their equivalence. The obtained empirical results provided encouraging evidence towards that direction.

[5]Replaces relational operators, e.g., $>$ with $<$.

VI. CONCLUDING REMARKS

In this paper, nine problematic patterns able to detect equivalent and partially equivalent mutants are formally introduced. A partially equivalent mutant is a mutant that is equivalent to the original program for a subset of paths. For each introduced pattern, specific conditions between variable *definitions* and *uses* are defined, which are necessary for the existence of the respective patterns in the source code of the program under test. The *Single Static Assignment* intermediate representation of the examined program is utilized for the purposes of the underlying data flow analysis. The salient feature of this representation is that each variable is assigned exactly once, thus, it facilitates powerful code analysis and a clearer definition of the aforementioned conditions. The described data flow patterns can be utilized in two ways: on the one hand, they can provide advice on specific code locations that should not be mutated because they will generate equivalent mutants and on the other, they can guide the test case generation process to paths that could yield a killing test case for the examined partially equivalent mutants. An empirical study, conducted on several open source projects, investigated the detection power of these patterns and their existence in real world software. The obtained results revealed that the introduced patterns were indeed present in the considered test subjects and were able to detect approximately 70% of the examined equivalent mutants.

REFERENCES

[1] R. Hamlet, "Testing programs with the aid of a compiler," *Software Engineering, IEEE Transactions on*, vol. SE-3, no. 4, pp. 279–290, 1977.

[2] R. DeMillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[3] R. DeMillo, D. Guindi, K. N. King, and W. M. McCracken, "An overview of the mothra software testing environment." Technical Report SERC-TR-3-P, Purdue University, Tech. Rep., 1987.

[4] R. DeMillo, D. Guindi, K. N. King, W. M. McCracken, and A. Offutt, "An extended overview of the mothra software testing environment," in *Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on*, Jul 1988, pp. 142–151.

[5] K. N. King and A. J. Offutt, "A fortran language system for mutation-based software testing," *Software: Practice and Experience*, vol. 21, no. 7, pp. 685–718, 1991.

[6] M. Daran and P. Thévenod-Fosse, "Software error analysis: A real case study involving real faults and mutations," in *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '96. New York, NY, USA: ACM, 1996, pp. 158–171.

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

[8] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 608–624, 2006.

[9] T. Budd and D. Angluin, "Two notions of correctness and their relation to testing," *Acta Informatica*, vol. 18, no. 1, pp. 31–45, 1982.

[10] A. J. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *Software Testing, Verification and Reliability*, vol. 7, no. 3, pp. 165–192, 1997.

[11] W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Trans. Softw. Eng.*, vol. 8, no. 4, pp. 371–379, Jul. 1982.

[12] Y. Jia and M. Harman, "Higher order mutation testing," *Information and Software Technology*, vol. 51, no. 10, pp. 1379 – 1393, 2009, ¡ce:title¿Source Code Analysis and Manipulation, {SCAM} 2008¡/ce:title¿.

[13] R. A. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Trans. Softw. Eng.*, vol. 17, no. 9, pp. 900–910, Sep. 1991.

[14] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *Software Engineering, IEEE Transactions on*, vol. 38, no. 2, pp. 278–292, 2012.

[15] M. Papadakis and N. Malevris, "Automatic mutation test case generation via dynamic symbolic execution," in *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, 2010, pp. 121–130.

[16] R. Hierons, M. Harman, and S. Danicic, "Using program slicing to assist in the detection of equivalent mutants," *Software Testing, Verification and Reliability*, vol. 9, no. 4, pp. 233–262, 1999.

[17] D. Schuler and A. Zeller, "Covering and uncovering equivalent mutants," *Software Testing, Verification and Reliability*, 2012.

[18] M. Harman, R. Hierons, and S. Danicic, "The relationship between program dependence and mutation analysis," in *Mutation Testing for the New Century*, ser. The Springer International Series on Advances in Database Systems, W. Wong, Ed. Springer US, 2001, vol. 24, pp. 5–13.

[19] D. Jackson and E. J. Rollins, "A new model of program dependences for reverse engineering," in *Proceedings of the 2Nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, ser. SIGSOFT '94. New York, NY, USA: ACM, 1994, pp. 2–10.

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

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

[22] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2008.

[23] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: an automated class mutation system," *Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 97–133, Jun. 2005.

[24] J. Offutt and N. Li. The μjava home page. Last Accessed January 2014. [Online]. Available: http://cs.gmu.edu/~offutt/mujava/

[25] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 2, pp. 99–118, Apr. 1996.

[26] J. Maldonado, M. Delamaro, S. Fabbri, A. Silva Simão, T. Sugeta, A. Vincenzi, and P. Masiero, "Proteum: A family of tools to support specification and program testing based on mutation," in *Mutation Testing for the New Century*, ser. The Springer International Series on Advances in Database Systems, W. Wong, Ed. Springer US, 2001, vol. 24, pp. 113–116.

[27] D. Schuler and A. Zeller, "(Un-)covering equivalent mutants," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ser. ICST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 45–54.

[28] J. Dolby, M. Vaziri, and F. Tip, "Finding bugs efficiently with a sat solver," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 195–204.

[29] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, Oct. 1991.

[30] M. Kintis and N. Malevris, "Identifying more equivalent mutants via code similarity," *Asia-Pacific Software Engineering Conference*, pp. 180–188, 2013.

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

[32] A. Offutt and 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.

[33] M. Kintis, M. Papadakis, and 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.