UNIVERSIDADE FEDERAL DE ALAGOAS
INSTITUTO DE COMPUTAÇÃO
CIÊNCIA DA COMPUTAÇÃO

LUIZ MATHEUS DE ALENCAR CARVALHO

MUTANTES EQUIVALENTES EM SISTEMAS CONFIGURÁVEIS: UM ESTUDO EMPÍRICO

MACEIÓ - AL
2018

Luiz Matheus de Alencar Carvalho

Mutantes Equivalentes em Sistemas Configuráveis: Um estudo empírico

Este trabalho foi aceito para publicação no evento *Workshop on Variability Modelling of Software-Intensive Systems* com o título *Equivalent Mutants in Configurable Systems: An Empirical Study*, ano 2017, pelos autores Luiz Carvalho, Marcio Augusto, Márcio Ribeiro, Leonardo Fernandes, Mustafa Al-Hajjaji, Rohit Gheyi, Thomas Thüm e foi aqui replicado para fins de substituição do trabalho de conclusão de curso por artigo científico nos termos da Norma Complementar CC-IC 04/15.

Orientador: Prof. Dr. Márcio de Medeiros Ribeiro

Coorientador: Prof. Me. Leonardo Fernandes Mendonça de Oliveira

Maceió - AL

2018

Luiz Matheus de Alencar Carvalho

Mutantes Equivalentes em Sistemas Configuráveis: Um estudo empírico

Este trabalho foi aceito para publicação no evento *Workshop on Variability Modelling of Software-Intensive Systems* com o título *Equivalent Mutants in Configurable Systems: An Empirical Study*, ano 2017, pelos autores Luiz Carvalho, Marcio Augusto, Márcio Ribeiro, Leonardo Fernandes, Mustafa Al-Hajjaji, Rohit Gheyi, Thomas Thüm e foi aqui replicado para fins de substituição do trabalho de conclusão de curso por artigo científico nos termos da Norma Complementar CC-IC 04/15.

Data de Aprovação: 23/01/2018

Banca Examinadora

Prof. Dr. Márcio de Medeiros Ribeiro
Universidade Federal de Alagoas
Instituto de Computação
Orientador

Prof. Me. Leonardo Fernandes Mendonça de Oliveira
Instituto Federal de Alagoas
Coordenadoria de Informática
Coorientador

Prof. Dr. Alessandro Fabricio Garcia
Pontifícia Universidade Católica do Rio de Janeiro
Departamento de Informática
Examinador

Prof. Dr. Flávio Mota Medeiros
Instituto Federal de Alagoas
Coordenadoria de Informática
Examinador

Prof. Dr. Rodrigo de Barros Paes
Universidade Federal de Alagoas
Instituto de Computação
Examinador

Science never solves a problem without creating ten more.

George Bernard Shaw

RESUMO

Teste de mutação é uma técnica de transformação de programas que tem por objetivo avaliar a qualidade dos casos de teste estimando sua capacidade de detectar falhas artificialmente injetadas. Os custos de utilizar teste de mutação são elevados, dificultado seu uso na indústria. Pesquisas prévias mostraram que aproximadamente um-terço dos mutantes gerados em sistemas simples são equivalentes. Em sistemas configuráveis, um conjunto de operadores que focam em diretivas de pré-processamento (e.g., `#ifdef`) foram propostas. Contudo, não existem estudos que investiguem se mutantes equivalentes ocorrem com esses operadores. Para realizar essa investigação, fornecemos uma ferramenta que implementa os operadores de mutação propostos em trabalhos anteriores e conduzimos um estudo empírico usando 30 arquivos C de 6 sistemas de escala industrial. Em particular, fornecemos exemplos de mutantes equivalentes e informações detalhadas, como quais operadores de mutação geram esses mutantes e com que frequência eles ocorrem. Nossos resultados preliminares mostram que aproximadamente 45% dos mutantes gerados são equivalentes. Consequentemente, os custos de testes podem ser drasticamente reduzidos se a comunidade evidenciar técnicas eficientes para evitar esses mutantes equivalentes.

Palavras-chave: Teste de Mutação. Sistemas Configuráveis. Pré-processador.

ABSTRACT

Mutation testing is a program-transformation technique that evaluates the quality of test cases by assessing their capability to detect injected artificial faults. The costs of using mutation testing are usually high, hindering its use in industry. Previous research has reported that roughly one-third of the mutants generated in single systems are equivalents. In configurable systems, a set of mutation operators that focus on preprocessor directives (e.g., `#ifdef`) has been proposed. However, there is a lack of existing studies that investigate whether equivalent mutants do occur with these operators. To perform this investigation, we provide a tool that implements the aforementioned mutation operators and we conduct an empirical study using 30 C files of six industrial-scale systems. In particular, we provide examples of equivalent mutants and detailed information, such as which mutation operators generate these mutants and how often they occur. Our preliminary results show that nearly 45% of the generated mutants are equivalent. Hence, testing costs can be drastically reduced if the community comes up with efficient techniques to avoid these equivalent mutants.

Keywords: Mutation Testing. Configurable Systems. Preprocessor.

LIST OF FIGURES

LIST OF TABLES

SUMMARY

1 INTRODUCTION

Mutation testing (DEMILLO et al., 1978) is a program-transformation technique that injects artificial faults to check whether the existing test cases can detect them. The quality of mutation testing largely depends on the used mutation operators. Ideally, mutation operators represent realistic faults, i.e., they mimic faults that programmers usually make. Recently, researchers have proposed to apply mutation testing for highly-configurable systems (AL-HAJJAJI et al., 2016a; ARCAINI et al., 2015; HENARD et al., 2014b; LACKNER; SCHMIDT, 2014; PAPADAKIS et al., 2014; REULING et al., 2015). Most of the existing approaches mutate feature models with aim of generating an effective set of products or to assessing the quality of the generated ones (ARCAINI et al., 2015; HENARD et al., 2014b; LACKNER; SCHMIDT, 2014; PAPADAKIS et al., 2014; REULING et al., 2015). Since a large percentage of the faults occur in the domain artifacts (i.e., source code) (ABAL et al., 2014), mutating the source code of configurable systems has a huge potential for fault detection. While conventional mutation operators can be used, special mutation operators are needed to mimic variability-related faults. Thus, mutation operators that focus on preprocessor-based directives (e.g., `#ifdef`) have been proposed (AL-HAJJAJI et al., 2016a). Evaluating these operators showed that they can simulate variability-related faults (AL-HAJJAJI et al., 2016a).

Mutation testing suffers from high costs (JIA; HARMAN, 2011). One kind of mutant increases such costs: the equivalent mutant (FERNANDES et al., 2017). An equivalent mutant is useless because it shows the same behavior as the original program (BUDD; ANGLUIN, 1982; JIA; HARMAN, 2011; MADEYSKI et al., 2014). Recent researches have found that the rate of equivalent mutants in single systems might lie between 4% and 39% (MADEYSKI et al., 2014). In this context, it is unknown how often equivalent mutants occur for industrial-scale configurable systems and which mutation operators produce most equivalent mutants.

To evaluate whether equivalent mutants are indeed a problem for configurable systems, we have implemented a tool, called #MUTAF, to mutate `#ifdefs` in C code. In particular, we have implemented 10 out of 13 mutation operators proposed in prior work (AL-HAJJAJI et al., 2016a). The implemented operators make changes in the variability models, in the domain artifact (i.e., source code) and in the mapping between models and domain artifact. Then, we use the tool to generate mutants in 30 files of six preprocessor-based industrial-scale systems: *OpenSSL*, *Vim*, *lighttpd*, *nginx*, *nano*, and *gnuplot*. To identify equivalent mutants, we take the original files and mutate them. Then, we preprocess both files with all macros (i.e., `#define` directives) combina-

tions to yield all configurations. Afterwards, to check whether the corresponding configurations are equivalent, we use compiler optimization and diff techniques (BALDWIN; SAYWARD, 1979; OFFUTT; CRAFT, 1994; PAPADAKIS et al., 2015; KINTIS et al., 2017). In case all original configurations are equivalent to the corresponding mutant configurations, we define this mutant as *totally equivalent*. In case only a strict, but non-empty subset of configurations is equivalent, we define the mutant as *partially equivalent*.

In particular, we present how often the total and partial equivalent mutants occur for configurable systems. As a result, we investigate whether equivalent mutants in configurable systems are a problem, i.e. whether researchers need to avoid them. In addition, we investigate which mutation operators lead to more equivalent mutants than other operators. This information can be helpful for testers, because they can avoid applying these operators or use them carefully.

Our results reveal that 7.9% and 38.0% of the generated mutants are partially and totally equivalent, respectively. Regarding the mutation operators, we find that the ones that add and remove `#ifdef` conditions generate more totally equivalent mutants.

Furthermore, we consider invalid mutants when a mutant does not compile in any configuration and valid mutants when a mutant compiles at least one configuration. In this context, our empirical study also shows how often invalid mutants occurred.

In summary, this paper provides the following contributions:

- A mutant generator tool that considers mutation operators for configurable systems; and
- An empirical study to present numbers of partially and totally equivalent mutants in C preprocessor-based configurable systems, i.e., *OpenSSL*, *Vim*, *lighttpd*, *nginx*, *nano*, and *gnuplot*. In addition, we provide numbers about the operators that lead to more equivalent mutants than others and numbers on invalid and valid mutants per operator.

2 Equivalent Mutants in Configurable Systems

In the context of configurable systems, several approaches exploited mutation testing to generate an effective set of products (ARCAINI et al., 2015; HENARD et al., 2014b) or to assess the quality of a set of generated products (REULING et al., 2015). For this purpose, several mutation operators have been proposed to mutate configurable systems. However, most of the existing ones focus mainly on feature models (ARCAINI et al., 2015; HENARD et al., 2014b; REULING et al., 2015; LACKNER; SCHMIDT, 2014). In general, the existing conventional mutation operators in single-system engineering can also be used to mutate configurable systems by applying these operators to the generated products or to the original source code. Nevertheless, these mutation operators may not able to mimic faults that are caused due to variability (ABAL et al., 2014). Thus, mutation operators that take the variability into account may cause faults in configurable systems that cannot be triggered using the conventional operators. Recently, mutation operators for preprocessor-based configurable systems have been proposed (AL-HAJJAJI et al., 2016a). In this work, we consider the aforementioned operators to generate mutants in configurable systems.

The costs of mutation testing are usually high due to the large amount of possible mutants (JIA; HARMAN, 2011). Kintis et al. (KINTIS et al., 2017) show that, for some cases, more than a third of these mutants for single systems are equivalents. The equivalent mutant problem has been a barrier that prevents mutation testing from being more widely used. To detect if a program and one of its mutants are equivalent is undecidable (BUDD; ANGLUIN, 1982). As a result, the detection of equivalent mutants alternatively may have to be carried out by humans, but manually checking mutant equivalence is error-prone (people judged equivalence correctly in about 80% of the cases (ACREE, 1980)) and time consuming (approximately 15 minutes per equivalent mutant (SCHULER; ZELLER, 2013)).

In mutation testing for configurable systems, mutating a configurable program S yields a configurable mutant S'. To analyze whether the mutant is equivalent, we need to compare each configuration yielded by S against each corresponding configuration produced by S' for each valid configuration.[1]

Given this context, we now explain the problem of equivalent mutants in configurable systems and introduce the concepts of *totally* and *partially* equivalent mutants.

We assume a semantics function S_c that takes a configurable system S and a configuration

[1] Valid configuration in this context means compilable or specified in artifacts such as Feature Models and Configuration Knowledge

c to generate the respective product. Two products S_c and S'_c are equivalent, denoted by $S'_c = S_c$, if S'_c preserves the observable behavior of S_c. Otherwise, they are not equivalent denoted by $S'_c \neq S_c$. Furthermore, we define function wf that takes a product S_c as input and indicates whether S_c is well-formed or not:

$$wf(S_c) = \begin{cases} true & \text{if } S_c \text{ has no compilation errors} \\ false & \text{if } S_c \text{ has compilation errors} \end{cases}$$

Given a configurable mutant M derived by applying a mutation operator o to a configurable system S (i.e., $M = o(S)$) and the set of configurations of S as C_S, we define that the configurable mutant M is *totally equivalent* if and only if

$$(\forall c \in C_S : \ wf(M_c) \ \rightarrow \ M_c = S_c) \ \wedge$$
$$(\exists c' \in C_S : \ wf(M_{c'})).$$

Similarly, a configurable mutant M is *partially equivalent* if and only if

$$(\exists c \in C_S : wf(M_c) \ \wedge \ M_c \neq S_c) \ \wedge$$
$$(\exists c' \in C_S : wf(M_{c'}) \ \wedge \ M_{c'} = S_{c'}).$$

Notice the following two examples that illustrate the definition presented. Figure 1 shows a code snippet from the *vim* configurable system.[2] At the top, we illustrate the original version of the code and a mutant yielded by using the AICC (Adding `#ifdef` Condition around Code) mutation operator (AL-HAJJAJI et al., 2016a) (see the added lines, i.e., 138 and 157). In this example, the mutant is *totally equivalent*, as all valid configurations generated by using this mutant are equivalent to all configurations generated by the original. Observing from the tester's perspective this mutant is useless, since no configuration produced a mutated program with a behavior different from the original program. Then all configurations can be discarded.

Figure 2 illustrates a mutant generated by using the RFIC (Removing Feature of `#ifdef` Condition) mutation operator (AL-HAJJAJI et al., 2016a). The mutant removed part of the `#ifdef` condition (see line 15 at the top-left corner of Figure 2). In this second example, we have a *partially equivalent* mutant, as only three out of four valid configurations are equivalent, i.e., `[], [FEAT_CRYPT], and [PROTO, FEAT_CRYPT]`. From the point of view of the tester, this mutant may be interesting to be tested, but only for the configuration that produces behavior different from the original program. In this case, only the equivalent configuration need to be discarded.

[2] <https://github.com/vim/vim/blob/edf3f97ae2af024708ebb4ac614227327033ca47/src/crypt_zip.c>

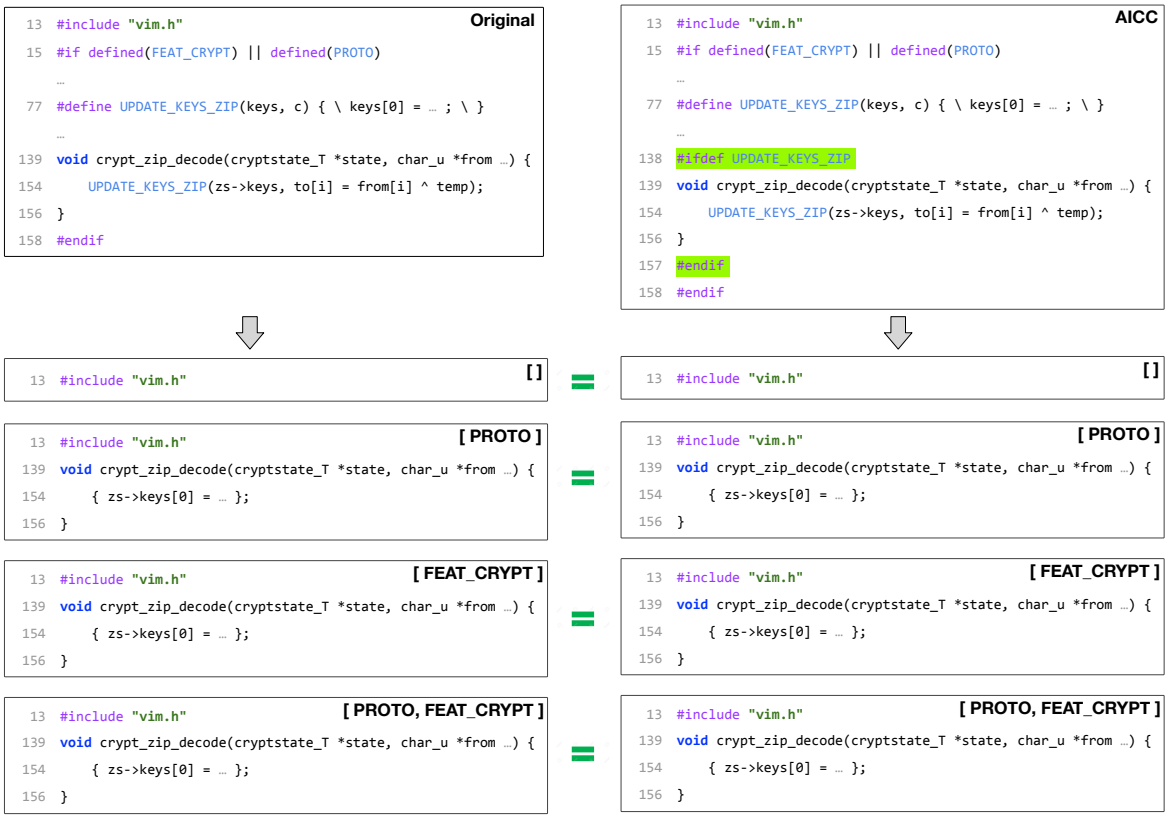

Figure 1 – Example of totally equivalent mutant. All configurations generated by the mutant (AICC) are the same when compared to all configurations generated by the original code.

Analogously to single systems, equivalent mutants increase costs in configurable systems. This means that executing the test suite against the configurations of the original configurable system will lead to the same results as executing the test suite against the corresponding configurations of the mutant. This way, the equivalent mutant problem is even more challenging for configurable systems, as we need to deal with it in many different configurations.

In this paper, we present an empirical study to raise the awareness of equivalent mutants in configurable systems. In particular, we focus on totally and partially equivalent mutants, as the ones we present in this chapter.

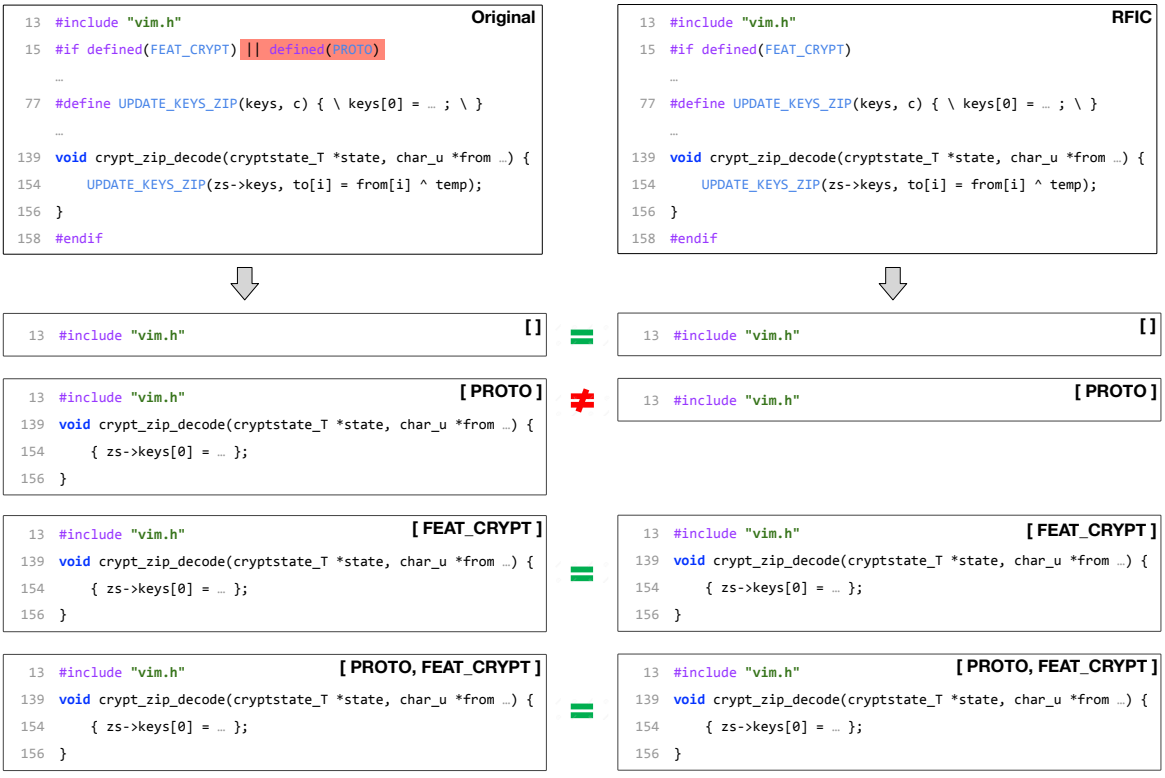

Figure 2 – Example of partially equivalent mutant. One corresponding configuration is different, i.e., `[PROTO]`.

3 Mutant generator

We have created #MUTAF to implement the mutation operators proposed by (AL-HAJJAJI et al., 2016a). In this chapter, we show the mutation operators that have been implemented. Moreover, we discuss how #MUTAF works and we exemplify the mutation operators with real cases from *OpenSSL*, *Vim*, *lighttpd*, *nginx*, *nano*, and *gnuplot*. The examples show the original code on the left side and the mutants on the right side.

3.1 #MUTAF

The #MUTAF is a mutant generator. The mutation operators are applied to C preprocessor (CPP) and they were implemented as described and exemplified in Section 3.2. #MUTAF uses the `srcML`[1] as a parser. It may parses a C file considering C preprocessor (CPP) directives. The `srcML` converts the original source code and returns an XML file with statements and CPP directives marked. After, #MUTAF parses the generated XML and applies mutations operators. For each generated mutant, #MUTAF calls the `srcML` to transform the mutant from XML to C code.

3.2 MUTATION OPERATORS

Remove Conditional Feature Definition (RCFD). The RCFD removes a `#define` inside the `#ifdef`, `#ifndef`, and `#if` block. The following mutant was generated from the file `popupmnu.c` contained in the `Vim` project.

```
#if defined(FEAT_INS_EXPAND)
    || defined(PROTO)
  #define PUM_DEF_HEIGHT 10
#endif
```

```
#if defined(FEAT_INS_EXPAND)
    || defined(PROTO)

#endif
```

Add Condition to Feature Definition (ACFD). The ACFD adds an `#ifdef` block or `#ifndef` block around the `#define`. The following mutant was generated from the file `history.c` contained in the `nano` project.

[1] www.srcml.org

```
#ifndef SEARCH_HISTORY

#define SEARCH_HISTORY
    "search_history"

#endif
```

```
#ifndef SEARCH_HISTORY
#ifdef POSITION_HISTORY
#define SEARCH_HISTORY
    "search_history"
#endif
#endif
```

Ifdef Condition around Code (AICC). The AICC adds a `#ifdef` block or `#ifndef` block around a code fragment. We decided to add `#ifdef` or `#ifndef` just around function to avoid exponential growth since this operator can be added to some code fragment. The following mutant was generated from the file `history.c` contained in the `nano` project.

```

void history_init(void)
{
    ...
}
```

```
#ifdef SEARCH_HISTORY
void history_init(void)
{
    ...
}
#endif
```

Adding Feature to ifdef Condition (AFIC). The AFIC changes the `#ifdef` conditions or `#ifndef` conditions adding a new dependency. We decided to implement this operator with the addition of the `&&` operator followed by a feature, already present in the file, in the `#if` expression. We chose this decision project to avoid exponential growth. The following mutant was generated from the file `json.c` contained in the `Vim` project.

```
#if defined(FEAT_EVAL)
  || defined(PROTO)

   ...

#endif
```

```
#if defined(FEAT_EVAL)
  || defined(PROTO)
 && defined(USING_FLOAT_STUFF)
   ...

#endif
```

Remove ifdef Condition (RIDC). The RIDC removes `#ifdef`, `#ifndef`, `#if`, `#else`, and `#endif`. However, it does not remove the block content. The following mutant was generated from the file `tables.c` contained in the `gnuplot` project.

```
#ifndef lint
   static char *RCSid() { ... }
#endif
```

```
static char *RCSid() { ... }
```

Removing Feature of ifdef Condition (RFIC). The RFIC removes a feature inside the `#if` condition. Alike AFIC, we have implemented to remove just one feature from `#if` to avoid exponential growth. The following mutant was generated from the file `json.c` contained in the `Vim` project.

```
#if defined(FEAT_EVAL)
    || defined(PROTO)
   ...
#endif
```

```
#if defined(FEAT_EVAL)

   ...
#endif
```

Replacing ifdef Directive with ifndef Directive (RIND). O RIND replaces the `#ifdef` by `#ifndef`. The following mutant was generated from the file `chunk.c` contained in the `lighttpd` project.

```
#ifdef __COVERITY__
    umask(0600);
#endif
```

```
#ifndef __COVERITY__
    umask(0600);
#endif
```

Replacing ifndef Directive with ifdef Directive (RNID). The RNID replaces the `#ifndef` by `#ifdef`, in other words, the opposite of RIND. The following mutant was generated from the file `s3_enc.c` contained in the `OpenSSL` project.

```
#ifndef OPENSSL_NO_COMP
    COMP_METHOD *comp;
#endif
```

```
#ifdef OPENSSL_NO_COMP
    COMP_METHOD *comp;
#endif
```

Removing Complete ifdef Block (RCIB). The RCIB removes `#ifdef`, `#ifndef` and `#if` blocks. Unlike RIDC, it removes the entire `#ifdef` block. The following mutant was generated from the file `ssl_sess.c` contained in the `OpenSSL` project.

```
dest->ext.hostname = NULL;
#ifndef OPENSSL_NO_EC
dest->ext.ecpointformats=NULL;
dest->ext.supportedgroups=NULL;
#endif
dest->ext.tick = NULL;
```

```
dest->ext.hostname = NULL;




dest->ext.tick = NULL;
```

Moving Code around ifdef Blocks (MCIB). The MCIB moves code fragments that are before a block of `#ifdef`, `#ifndef` or `#if` to after the block (`#endif`). Additionally, it moves the code fragment that is after the `#endif` to before `#ifdef`, `#ifndef`, or `#if`. The following mutant was generated from the file `prompt.c` contained in the `nano` project.

```
input = get_kbinput(...);
#ifndef NANO_TINY
if (input == KEY_WINCH)
        return KEY_WINCH;
#endif
```

```
#ifndef NANO_TINY
if (input == KEY_WINCH)
        return KEY_WINCH;
#endif
input = get_kbinput(...);
```

Remove Feature from Model (RFDM). The RFDM deletes a feature from a model. This operator has not been implemented because it is not applied in `#ifdef` condition. Furthermore, it is coupled to the variability model.

Modify Feature Dependency in Model (MFDM). The MFDM modifies a dependency of the feature in variability model. This operator has not been implemented for the same reason of the RFDM.

Conditionally Applying Conventional Operator (CACO). The CACO includes some traditional mutation operator. Therefore, all available through C-language mutation testing tools. This operator is very wide, then CACO needs to be implemented in tools more refined as proposed by (AL-HAJJAJI et al., 2017a). In this way, we have not implemented it.

The mutation operators implemented in the #MUTAF are summarized in Table 1.

Table 1 – Mutation operators implemented in #MUTAF.

Mutation Operator	**Acronyms**	**Description**
Remove Conditional Feature Definition	RCFD	Removes a feature definition.
Add Condition to Feature Definition	ACFD	Adds an #ifdef or #ifndef condition around an existing define statement.
Adding #ifdef Condition Around Code	AICC	Adds an #ifdef or #ifndef condition around a code fragment.
Adding Feature to #ifdef Condition	AFIC	Manipulates an #ifdef condition by inserting an additional logical dependency to the expression.
Remove #ifdef Condition	RIDC	Deletes an #ifdef or #ifndef condition.
Removing Feature of #ifdef Condition	RFIC	Removes the occurrence of a feature from an #ifdef expression.
Replacing #ifdef Directive with #ifndef Directive	RIND	Changes an existing #ifdef directive to an #ifndef directive.
Replacing #ifndef Directive with #ifdef Directive	RNID	Changes an #ifndef directive to an #ifdef directive.
Removing Complete #ifdef Block	RCIB	Deletes an entire #ifdef, #ifndef, and #if block.
Moving Code Around #ifdef Blocks	MCIB	Moves a certain code fragment around an #ifdef block.

4 Empirical Study

To better understand the problem of equivalent mutants in configurable systems, we perform an empirical study. In what follows, we present the settings of our study (Section 4.1), the results and discussion (Section 4.2), and threats to validity (Section 4.3).

4.1 SETTINGS

In our study, we intend to answer the following research questions:

- **RQ1:** How often do totally and partially equivalent mutants occur for configurable systems?
- **RQ2:** Which mutation operators lead to more equivalent mutants than other operators?
- **RQ3:** Which mutation operators lead to more invalid mutants?

Answering **RQ1** is important to understand the equivalent mutant problem for configurable systems. Answering **RQ2** and **RQ3** will help (i) testers to be aware of costly operators and thus avoid them; and (ii) researchers to know which mutation operators need to be improved. To answer these questions, we select 30 C files of six industrial-scale configurable systems that contain preprocessor directives. We list the systems we use in Table 2. To make our analysis feasible, we randomly select files with at most six different feature macros.

We execute #MUTAF on each file to generate the mutants. We then generate all configurations available in both the original files as well as the mutated ones, and perform a comparison as illustrated in Figures 1 and 2. In case a certain configuration does not compile in the original code, we do not analyze such configuration in any mutant. In addition, if the mutant does not compile for a certain configuration, we disregard this configuration from our numbers. For example, suppose a mutant with four configurations. If three are equivalents and one does not compile, we count this mutant as totally equivalent. This definition may seem strange to some researchers because by having a configuration different from the original, this mutant could not be called totally equivalent. However, we are observing from the tester's perspective. In this case, the mutant is completely useless and is not suitable for use in mutation testing. Not

Table 2 – Open-source configurable systems we use in our study.

System	Domain	LOC
OpenSSL	TSL and SSL protocol	269,621
Vim	Text editor	312,201
lighttpd	Web server	48,043
nginx	Http server	120,459
nano	Text editor	37,822
gnuplot	Command-line driven graphing utility	89,127

counting mutants that do not compile is in accordance to mutation testing previous research, as they produce invalid mutations (JUST, 2014).

To detect equivalent mutants, we rely on compiler optimizations for checking equivalence (BALDWIN; SAYWARD, 1979; OFFUTT; CRAFT, 1994; KINTIS et al., 2017). So, a mutant is equivalent to the original program if, after the transformations made by compile optimization, they end up with the same object code. Notice that this approach is sound in the sense that two equal binaries mean that the programs have the same behavior. However, we do not detect equivalent mutants with the same behavior, but with different object codes. This leads us to have no false positives. But, this approach is definitely not free to have false negatives.

We compile the configurations with the `gcc`[1] compiler. This compiler has many levels of optimization. In this study, we decided to setup the compiler to use the option *-O3*. This option has been used by previous studies to detect equivalent mutants (KINTIS et al., 2017; PAPADAKIS et al., 2015). Then, we check whether two binaries are equivalent by using the `diff`[2] utility with the flag "--binary." We execute our study on a Linux kernel 4.9.0 , `gcc` 6.3.0-11, and `diff` 3.5-3.

4.2 RESULTS AND DISCUSSION

In this section, we present the results and answer our research questions. All results of our study are available online: <https://lzmths.github.io/tcc/>. Figure 3 shows the distribution of percentages of the partially and the totally equivalent mutants for each analyzed system. We observe that large percentages of the mutants are totally equivalent, especially for configurable systems *gnuplot* and *nginx*, with median values 63.6% and 76.5%, respectively. Table 3 presents the results in more detail and answer **RQ1**. For each file, we illustrate the number of preprocessor macros, the number of valid mutants (the ones that compile), and the number of partially equivalent and totally equivalent mutants.

For example, when considering the `crypt_zip.c` file (*Vim*), #MUTAF generated 48 valid mutants. This file has 3 preprocessor macros which give us 8 possible configurations, without considering information from potential constrains (e.g., feature model). Notice that 18.8% and 62.5% of the mutants are partially and totally equivalent, respectively. The examples we present in Figures 1 and 2 are from the `crypt_zip.c` file.

In general, the number of totally equivalent mutants is higher than the number of partially equivalent mutants. In particular, 7.9% and 38.0% of the mutants are partially and totally equivalent, respectively. In this sense, these results answer our **RQ1** and evidence the importance of creating technical solutions to avoid equivalent mutants in configurable systems and thus reduce costs.

To answer **RQ2**, we refer to Table 4. The mutation operators that contribute most with totally equivalent mutants are AFIC, RFIC, RCFD, and RIDC. On the other hand, RCIB is the

[1] <https://gcc.gnu.org/>
[2] <https://www.gnu.org/software/diffutils/>

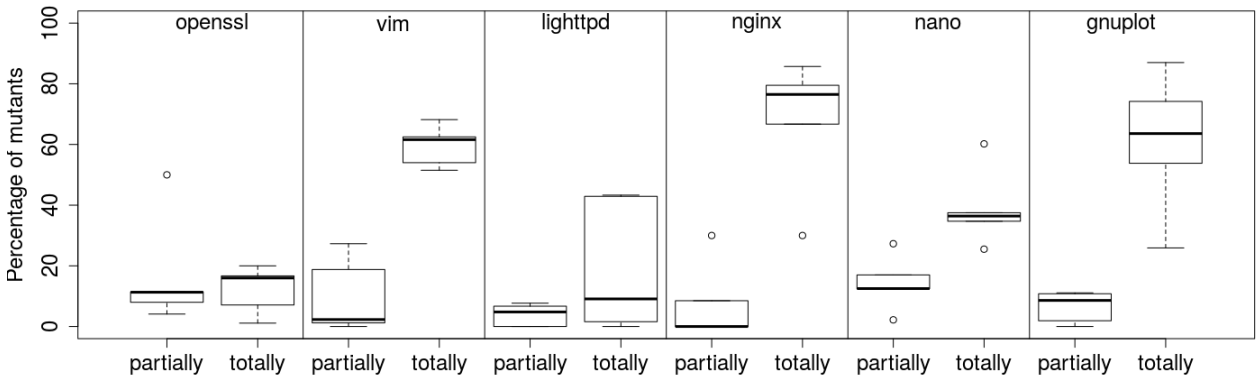

Figure 3 – Partially and totally equivalent mutants.

one that most contributes to partially equivalent mutants. To better understand the results, we refer to the code snippets presented in Figure 4. We now proceed by explaining and discussing the results from the operators that caused the highest and lowest numbers.

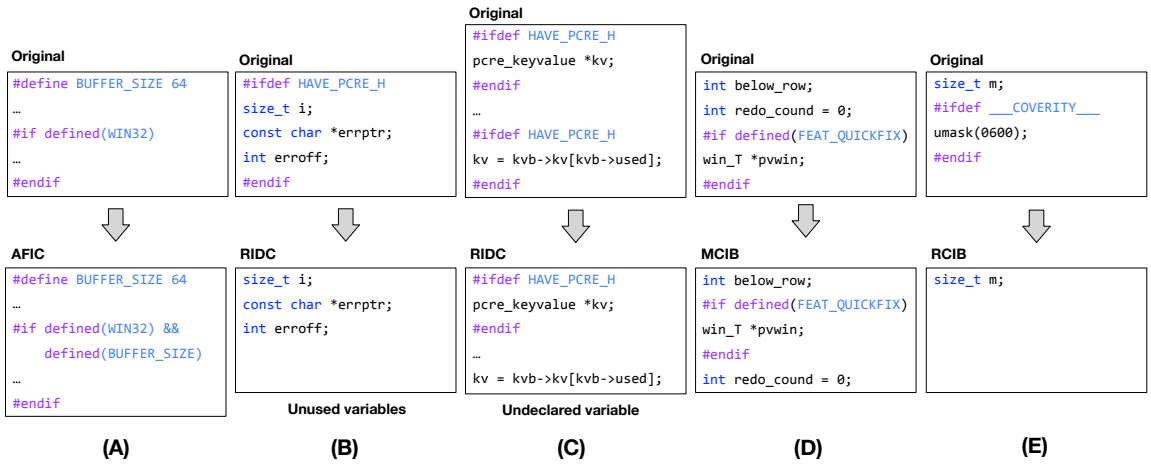

Figure 4 – Examples of applying some mutation operators.

AFIC adds a feature to `#ifdef` condition. RFIC, on the other hand, removes. In case the original code contains, for example, `#if defined(WIN32)`, AFIC may yield a mutant with the following: `#if defined(WIN32) && defined(BUFFER_SIZE)` (Figure 4(A)). Here, when comparing the original code with the mutant, only one configuration will be different, i.e., when we enable `WIN32` and disable `BUFFER_SIZE`. In our evaluation, this only one "different" configuration (such as enabling `A` and disabling `B`) did not compile in the majority of the cases. Because we do not count the configurations that do not compile, we ended up with many totally equivalent mutants. The same situation happens for the RFIC mutation operator. AFIC and RFIC generated 92.0% and 77.8% of totally equivalent mutants, respectively.

Table 3 – Number of partially and totally equivalents mutants for configurable systems.

Project	File	Macros	Valid Mutants	Partially Eq. Mutants		Totally Eq. Mutants	
openssl	crypto/rand/rand_lib.c	4	98	4	4.1%	11	11.2%
	ssl/s3_enc.c	3	35	4	11.4%	7	20.0%
	ssl/ssl_sess.c	4	168	19	11.3%	12	7.1%
	ssl/bio_ssl.c	1	6	3	50.0%	1	16.7%
	ssl/ssl_cert.c	3	25	2	8.0%	4	16.0%
vim	src/hashtab.c	1	86	1	1.2%	53	61.6%
	src/crypt_zip.c	3	48	9	18.8%	30	62.5%
	src/json.c	6	66	0	0.0%	34	51.5%
	src/nbdebug.c	4	22	6	27.3%	15	68.2%
	src/popupmnu.c	4	87	2	2.3%	47	54.0%
lighttpd	src/buffer.c	3	220	17	7.7%	20	9.1%
	src/chunk.c	1	125	6	4.8%	2	1.6%
	src/fdevent_libev.c	1	28	0	0.0%	0	0.0%
	src/fdevent_poll.c	2	30	2	6.7%	13	43.3%
	src/keyvalue.c	1	7	0	0.0%	3	42.9%
nginx	core/ngx_rwlock.c	3	24	2	8.3%	16	66.7%
	core/ngx_string.c	3	10	3	30.0%	3	30.0%
	mail/ngx_mail.c	3	17	0	0.0%	13	76.5%
	http/ngx_http_upstream_rr.c	4	56	0	0.0%	48	85.7%
	core/ngx_inet.c	5	73	0	0.0%	58	79.5%
nano	src/move.c	2	16	2	12.5%	6	37.5%
	src/chars.c	4	72	9	12.5%	25	34.7%
	src/prompt.c	5	77	21	27.3%	28	36.4%
	src/browser.c	6	47	8	17.0%	12	25.5%
	src/history.c	4	93	2	2.2%	56	60.2%
gnuplot	src/boundary.c	1	65	7	10.8%	35	53.8%
	src/breaders.c	4	93	8	8.6%	69	74.2%
	src/gpexecute.c	4	54	6	11.1%	14	25.9%
	src/tables.c	4	11	0	0.0%	7	63.6%
	src/alloc.c	6	54	1	1.9%	47	87.0%
Total		59	1812	144	7.9%	689	38.0%

RIDC removes the `#ifdef`. In case the `#ifdef` is encompassing declarations of variables and functions (see Figure 4(B)), they will be unused after applying RIDC and preprocessing the code with the macro disabled (e.g., `HAVE_PCRE_H`). Unused variables and functions are detected and properly removed during the compiler optimization, yielding equivalent mutants. However, when applying RIDC, we may also have compilation errors, i.e., undeclared variables and functions (Figure 4(C)). As configurations that do not compile are not present in our numbers, we increase the odds of having totally equivalent mutants (e.g., for four configurations, three are equivalent and one does not compile). RIDC generated 84.2% of totally equivalent mutants.

MCIB generated a moderate number of totally equivalent mutants (32.0%), especially in cases where the operator moved variable declarations around `#ifdef` blocks (see Figure 4(D)).

Table 4 – Number of partially equivalents, totally equivalents, generated mutants, and valid mutants per mutation operator.

File	Generated Mutants	Valid Mutants	Percentage of Valid Mutants	Partially Equivalent		Totally Equivalent	
AFIC	50	50	100.00%	4	8.0%	46	92.0%
RFIC	9	9	100.00%	2	22.2%	7	77.8%
RIDC	253	234	92.49%	20	8.5%	197	84.2%
RCFD	18	9	50.00%	0	0.0%	8	88.9%
ACFD	134	120	89.55%	10	8.3%	93	77.5%
MCIB	272	247	90.81%	21	8.5%	79	32.0%
AICC	918	875	95.31%	44	5.0%	223	25.5%
RCIB	154	121	78.57%	37	30.6%	22	18.2%
RNID	54	52	96.30%	0	0.0%	7	13.5%
RIND	111	96	86.49%	6	6.3%	7	7.3%
Total	1973	1813	91.89%	144	7.9%	689	38.0%

AICC generated a few number of totally equivalent mutants. In general, the totally equivalent cases happened when the operator adds the `#ifdef` with a macro that has already been defined by using the `#define` directive (see the example presented in Figure 1).

RIND and RNID generated a few number of totally equivalent mutants (7.3% and 13.5%, respectively). The main reason is that the configurations tend to be different when compared to the original code due to the different blocks that are added and removed after preprocessing the code.

RCIB generated a moderate number of partially equivalent mutants, i.e., 18.2%. For example, when removing the entire block encompassed by `#ifdef ____COVERITY____`, the configuration where `____COVERITY____` is disabled is equivalent (Figure 4(E)). On the other hand, in case such macro is enabled, we have a non-equivalent configuration. This balance between equivalent and non-equivalent configurations increases the odds of having partially equivalent mutants (30.6%).

To answer **RQ3**, we refer to Table 4. Invalid mutants do not compile in any configuration. The opposite, a valid mutant compiles at least one configuration and mutation operator that contribute most to invalid mutants is RCFD. The operators that most generated valid mutants were AFIC and RFIC. In general, the proportion of valid mutants is high (91.8%).

RCFD generated half of the invalid mutants. This occurs because it removes `#define` that are used to define constants in the code. AFIC and RFIC generated all valid mutants because they add or remove a feature in an `#if` condition. Whenever this feature is enabled, then the expression is equivalent to the original expression.

4.3 THREATS TO VALIDITY

The low number of configurable systems we used represents a threat to external validity. To alleviate this threat, we selected projects of different sizes and domains. We intend to conduct

further experiments in future using different sizes of configurable systems in order to generalize the outcomes of this paper.

Our implementation of #MUTAF is a threat to internal validity. Because there is no formal specification of the mutation operators, the implementation may vary according to the tool. For example, for some mutation operators we decided to avoid an exponential growth of the number of mutants: AICC (Adding `#ifdef` Condition Around Code) only applies `#ifdef` and `#ifndef` around functions. AFIC (Adding Feature to `#ifdef` Condition) only adds a logical dependency to the expression in case the added feature is defined in the same file. However, for other mutation operators, we have expanded the scope: RIDC and RCIB do not work only with `#ifdef`. They also consider preprocessors with `#ifndef`, `#if`, and `#else` condition. To check whether the tool is creating the mutants as expected, we sampled five mutants of each mutation operator and manually analyzed them. Moreover, the `gcc` compiler and the `diff` utility may also have faults. Nevertheless, we minimize this threat by relying on those systems that are heavily tested, deployed, and used in practice.

In this paper, we use the brute-force (all possible configurations) in files with at most six macros. It represents a threat because many configurable systems have a dedicated building tool that filters the possible configurations before performing compilation (eg.: `kconfig`). Our intention was to assess whether equivalent mutants represent a problem for configurable systems. Notice that this approach would not scale in case we decide to analyze files with many more macros.

The aforementioned manual analysis also represents a threat. We alleviate such a threat by double checking the questionable cases with a second researcher.

In our evaluation, we rely on compiler optimizations to check equivalence. Despite sound (no false positives), this technique may have false negatives. In this context, our results are conservative and represent a lower bound.

5 Related work

Recently, several mutation testing approaches have been proposed for highly configurable systems (AL-HAJJAJI et al., 2016a; ARCAINI et al., 2015; HENARD et al., 2014b; REULING et al., 2015; AL-HAJJAJI et al., 2017a; DEVROEY et al., 2016; DEVROEY et al., 2014). Al-Hajjaji et al. (AL-HAJJAJI et al., 2016a) propose mutation operators for preprocessor-based configurable systems and show that applying these operators can cause real faults. They aim with these operators to make changes that cause variability-related faults. In our work, we implemented the majority of those mutation operators and use them to generate mutants. We considered these generated mutants in our evaluation.

Arcaini et al. (ARCAINI et al., 2015) propose a fault-based approach that considers finding faults in feature models. For this purpose, they propose a set of mutation operators that mutate feature models. These mutated feature models are tested against configurations that are generated from the original ones. Reuling et al. (REULING et al., 2015) present a fault-based approach that generates an effective set of products with respect to the ability of finding faults. To achieve their goal, they propose a set of atomic and complex mutation operators that mutates feature diagrams. Similarly, Henard et al. (HENARD et al., 2014b) propose two mutation operators to alter the propositional formula of feature models. Using these operators, they generate a set of configurations that has the ability to detect the mutated feature models. Papadakis et al. (PAPADAKIS et al., 2014) report that considering fault-based approaches (i.e., mutation testing) to generate a set of products is more effective than generating products with combinatorial interaction testing with respect to the ability of detecting faults. Lackner et al. (LACKNER; SCHMIDT, 2014) assess the testing quality of configurable systems by exploiting mutation testing to measure the capability of detecting faults. With their approach, they consider model-based mutation operators. The aforementioned approaches (ARCAINI et al., 2015; HENARD et al., 2014b; LACKNER; SCHMIDT, 2014; PAPADAKIS et al., 2014; REULING et al., 2015) consider only mutation operators on the feature model level.

However, Abal et al. (ABAL et al., 2014) report that most of the reported faults are located in the domain artifacts (i.e., source code). In addition, they do not consider any techniques that may reduce the mutation testing costs. In our work, we mutate the source code of configurable systems. Furthermore, we investigate how often the equivalent mutants occur in configurable systems and how many equivalent mutants are generated by each mutation operator.

Al-Hajjaji et al. (AL-HAJJAJI et al., 2017a) propose to reduce the cost by decreasing the possibility of generating equivalent mutants. For this purpose, they propose to combine static analysis and T-wise testing to indicate in which code segments the mutation operators should be applied. However, further experiments are required to evaluate the effectiveness of their approach. In addition, Reuling et al. (REULING et al., 2015) propose two strategies to reduce the number of mutants, namely mutation selection and higher-order mutation. With mutation selection, they select a set of operators randomly. Furthermore, they also exploit the similarity notion in operators selection, where they consider dissimilar operators to be selected. However,

these strategies can be applied to the implemented mutation operators in future to reduce the mutation testing cost.

For highly configurable systems, several approaches have been proposed to select a set of products to be tested (JOHANSEN et al., 2012; AL-HAJJAJI et al., 2016b; PERROUIN et al., 2010; HENARD et al., 2014a). For example, Johansen et al. (JOHANSEN et al., 2012), Al-Hajjaji et al. (AL-HAJJAJI et al., 2016b), Perrioun et al. (PERROUIN et al., 2010) propose T-wise testing approaches to sample configurable systems. Henard et al. (HENARD et al., 2014a) suggest an alternative approach to T-wise testing by proposing a search-based approach that selects a set of products. Our approach can be used to assess the quality of the generated products with respect to the capability to find faults. Furthermore, numerous product prioritization approaches have been proposed to increase the fault detection rate by ordering products under test (AL-HAJJAJI et al., 2017b; SÁNCHEZ et al., 2014; LITY et al., 2017). These prioritization approaches can be applied to prioritize the generated mutants, which may increase the testing effectiveness.

In previous work, Braz et al. (BRAZ et al., 2016) proposes a change-centric approach that aims to compile only the configurations that affected by the changes. This previous approach can be used to avoid the equivalent mutants by considering only configurations that affected by changes as a result of applying the mutation operators.

In single systems, as surveyed by Jia and Harman (JIA; HARMAN, 2011), efforts have been made to reduce the mutation testing costs. For example, some approaches propose that selecting only small percentages of mutants are sufficient to achieve a high accuracy of mutation score (BUDD, 1980; ZHANG et al., 2013). However, applying these techniques of the single-system engineering may achieve promising results in the context of configurable systems. In a recent work (FERNANDES et al., 2017), we propose an approach to avoid generating equivalent mutants. In particular, we propose rules that are required to be applied during mutants generation. Therefore, the aforementioned approach may be also considered in future in the configurable systems to reduce the mutation testing cost.

6 Conclusions

We empirically assess whether the equivalent mutant in configurable systems is a problem. In particular, we distinguish between partially and totally equivalent mutants in configurable systems. In our evaluation, we have implemented a tool, called #MUTAF, to mutate `#ifdefs` in C code. Then we execute our tool in *OpenSSL*, *Vim*, *lighttpd*, *nginx*, *nano*, and *gnuplot*. To check whether the mutants are indeed equivalents, we rely on compiler optimizations and diff techniques. Our results revealed that 38.0% of the mutants are totally equivalent, i.e., no configuration generated by these mutants are useful for the mutation testing, and 7.9% of the mutants are partially equivalents, i.e., only a few configurations would be useful for testing. We also found that mutation operators that add or remove `#ifdef` conditions generate totally equivalent mutants more often. Our results bring evidence that equivalent mutants is a non-negligible problem for configurable systems. In addition, our evaluation is important to make testers aware of costly mutation operators. Last but not least, our findings are important to improve the mutation operators we explored in this work.

As future work we intend to investigate *duplicated mutants*. Duplicated mutants happen when two mutants are equivalent to each other. In this case, only one of them is useful. In addition, we intend to improve #MUTAF to: (i) automatically execute the test suite, (ii) compute the mutation score, and (iii) integrate techniques to avoid those kinds of equivalent and duplicated mutants.

REFERENCES

ABAL, I.; BRABRAND, C.; WASOWSKI, A. 42 variability bugs in the linux kernel: A qualitative analysis. In: ACM. **Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering**. New York, NY, USA, 2014. p. 421–432.

ACREE, J. A. T. **On Mutation**. Tese (Doutorado) — Georgia Institute of Technology, 1980.

AL-HAJJAJI, M.; BENDUHN, F.; THÜM, T.; LEICH, T.; SAAKE, G. Mutation operators for preprocessor-based variability. In: **Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems**. New York, NY, USA: ACM, 2016. p. 81–88.

AL-HAJJAJI, M.; KRIETER, S.; THüM, T.; LOCHAU, M.; SAAKE, G. Incling: Efficient product-line testing using incremental pairwise sampling. In: **Proceedings of the International Conference on Generative Programming: Concepts Experience**. New York, NY, USA: ACM, 2016. p. 144–155. ISBN 978-1-4503-4446-3.

AL-HAJJAJI, M.; KRüGER, J.; BENDUHN, F.; LEICH, T.; SAAKE, G. Efficient mutation testing in configurable systems. In: **Proceedings of the International Workshop on Variability and Complexity in Software Design**. Piscataway, NJ, USA: IEEE, 2017. (VACE '17), p. 2–8. ISBN 978-1-5386-2803-4.

AL-HAJJAJI, M.; THÜM, T.; LOCHAU, M.; MEINICKE, J.; SAAKE, G. Effective Product-Line Testing Using Similarity-Based Product Prioritization. **Software & Systems Modeling**, 2017. To appear.

ARCAINI, P.; GARGANTINI, A.; VAVASSORI, P. Generating tests for detecting faults in feature models. In: IEEE. **8th International Conference on Software Testing, Verification and Validation (ICST)**. Piscataway, NJ, USA, 2015. p. 1–10.

BALDWIN, D.; SAYWARD, F. **Heuristics for Determining Equivalence of Program Mutations.** [S.l.], 1979.

BRAZ, L.; GHEYI, R.; MONGIOVI, M.; RIBEIRO, M.; MEDEIROS, F.; TEIXEIRA, L. A change-centric approach to compile configurable systems with #ifdefs. In: **Proceedings of the International Conference on Generative Programming: Concepts and Experiences**. New York, NY, USA: ACM, 2016. p. 109–119. ISBN 978-1-4503-4446-3.

BUDD, T.; ANGLUIN, D. Two notions of correctness and their relation to testing. **Acta Informatica**, v. 18, n. 1, p. 31–45, 1982.

BUDD, T. A. **Mutation Analysis of Program Test Data**. Tese (Doutorado) — Yale University, New Haven, CT, USA, 1980.

DEMILLO, R.; LIPTON, R.; SAYWARD, F. Hints on test data selection: Help for the practicing programmer. **Computer**, v. 11, n. 4, p. 34–41, 1978.

DEVROEY, X.; PERROUIN, G.; CORDY, M.; PAPADAKIS, M.; LEGAY, A.; SCHOBBENS, P.-Y. A variability perspective of mutation analysis. In: **Proceedings of the International Symposium on Foundations of Software Engineering**. New York, NY, USA: ACM, 2014. p. 841–844. ISBN 978-1-4503-3056-5.

DEVROEY, X.; PERROUIN, G.; PAPADAKIS, M.; LEGAY, A.; SCHOBBENS, P.-Y.; HEYMANS, P. Featured model-based mutation analysis. In: **Proceedings of the International Conference on Software Engineering**. Piscataway, NJ, USA: IEEE, 2016. p. 655–666. ISBN 978-1-4503-3900-1.

FERNANDES, L.; RIBEIRO, M.; CARVALHO, L.; GHEYI, R.; MONGIOVI, M.; SANTOS, A.; CAVALCANTI, A.; FERRARI, F.; MALDONADO, J. C. Avoiding useless mutants. In: **Proceedings of the 16th International Conference on Generative Programming: Concepts Experience**. New York, NY, USA: ACM, 2017. p. 187–198.

HENARD, C.; PAPADAKIS, M.; PERROUIN, G.; KLEIN, J.; HEYMANS, P.; TRAON, Y. L. Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines. **IEEE Transaction Software Engineering**, v. 40, n. 7, p. 650–670, 2014. ISSN 0098-5589.

HENARD, C.; PAPADAKIS, M.; TRAON, Y. L. Mutation-based generation of software product line test configurations. In: **International Symposium on Search Based Software Engineering**. [S.l.]: Springer, 2014. p. 92–106.

JIA, Y.; HARMAN, M. An analysis and survey of the development of mutation testing. **IEEE Transactions on Software Engineering**, v. 37, n. 5, p. 649–678, 2011.

JOHANSEN, M. F.; HAUGEN, Ø.; FLEUREY, F. An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models. In: **Proceedings of the International Software Product Line Conference**. New York, NY, USA: ACM, 2012. p. 46–55. ISBN 978-1-4503-1094-9.

JUST, R. The major mutation framework: Efficient and scalable mutation analysis for java. In: ACM. **Proceedings of the International Symposium on Software Testing and Analysis**. New York, NY, USA, 2014. p. 433–436.

KINTIS, M.; PAPADAKIS, M.; JIA, Y.; MALEVRIS, N.; TRAON, Y. L.; HARMAN, M. Detecting trivial mutant equivalences via compiler optimisations. **IEEE Transactions on Software Engineering**, PP, n. 99, p. 1–1, 2017.

LACKNER, H.; SCHMIDT, M. Towards the assessment of software product line tests: a mutation system for variable systems. In: **Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools-Volume 2**. New York, NY, USA: ACM, 2014. p. 62–69.

LITY, S.; AL-HAJJAJI, M.; THüM, T.; SCHAEFER, I. Optimizing Product Orders Using Graph Algorithms for Improving Incremental Product-line Analysis. In: **Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems**. New York, NY, USA: ACM, 2017. p. 60–67. ISBN 978-1-4503-4811-9.

MADEYSKI, L.; ORZESZYNA, W.; TORKAR, R.; JOZALA, M. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. **IEEE Transactions on Software Engineering**, v. 40, n. 1, p. 23–42, 2014.

OFFUTT, A. J.; CRAFT, W. M. Using compiler optimization techniques to detect equivalent mutants. **Software Testing, Verification and Reliability**, v. 4, n. 3, p. 131–154, 1994.

PAPADAKIS, M.; HENARD, C.; TRAON, Y. L. Sampling program inputs with mutation analysis: Going beyond combinatorial interaction testing. In: **Software Testing, Verification and Validation**. New York, NY, USA: IEEE, 2014. p. 1–10.

PAPADAKIS, M.; JIA, Y.; HARMAN, M.; TRAON, Y. L. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In: **IEEE International Conference on Software Engineering**. Piscataway, NJ, USA: IEEE, 2015. p. 936–946.

PERROUIN, G.; SEN, S.; KLEIN, J.; BAUDRY, B.; TRAON, Y. L. Automated and scalable t-wise test case generation strategies for software product lines. In: **Proceedings of the International Conference on Software Testing, Verification and Validation**. Washington: IEEE, 2010. p. 459–468.

REULING, D.; BüRDEK, J.; ROTäRMEL, S.; LOCHAU, M.; KELTER, U. Fault-based product-line testing: Effective sample generation based on feature-diagram mutation. In: **Proceedings of the the International Conference on Software Product Line**. New York, NY, USA: ACM, 2015. p. 131–140. ISBN 978-1-4503-3613-0.

SÁNCHEZ, A. B.; SEGURA, S.; RUIZ-CORTÉS, A. A Comparison of Test Case Prioritization Criteria for Software Product Lines. In: **Proceedings of the International Conference on Software Testing, Verification and Validation**. Washington: IEEE, 2014. p. 41–50.

SCHULER, D.; ZELLER, A. Covering and uncovering equivalent mutants. **Software Testing, Verification and Reliability**, v. 23, n. 5, p. 353–374, 2013.

ZHANG, L.; GLIGORIC, M.; MARINOV, D.; KHURSHID, S. Operator-based and random mutant selection: Better together. In: **Proceedings of the International Conference on Automated Software Engineering**. Piscataway, NJ, USA: IEEE, 2013. p. 92–102.