How to Identify the Infeasible Test Requirements using Static Analyse? An Exploratory Study

João Choma Neto[1][a], Allan Mori[1], Ricardo Ferreira Vilela[1][b], Thelma E. Colanzi[2][c] and Simone R. S. de Souza[1]

[1]*Department of Computer Systems, University of São Paulo, São Carlos, São Paulo, Brazil*

[2]*Department of Informatic, State University of Maringá, Maringá, Paraná, Brazil*

Keywords: Software Testing, Structural Testing, Non-exacutability Problem.

Abstract: Context: Software testing is an essential activity to ensure the quality of the software. However, the selection and generation of test cases can be an expensive and hard task. A large number of infeasible test requirements (e.g. infeasible paths) collaborate to increase the effort on test data generation, and it is not a trivial task to identify them. Objective: To investigate and analyze an process of properties of infeasible test requirements identification in a static way without inputs data through an exploratory study. Methodology: We gathered a set of statistical properties to identify infeasible test requirements without the use of input data. We manually verified the identification process using a benchmark with 19 Java programs. Results and conclusions: The alternative process identified infeasible requirements without using input data and proved effective. This study highlights the tester's role in the process of identifying the infeasible elements and also the need to automate this process because level of complexity in decision making.

1 INTRODUCTION

In the context of enterprise information systems, it is essential that the system works well, preferably without faults. Testing activity should be employed in a systematic way to improve the software quality. Despite it is possible to automatize the software testing activity, some parts involve a high interaction of tester, for instance, to generate good test cases or to analyse the executability of some code paths. In this sense, solutions to facilitate the tester interaction is mandatory in order to reduce the testing activity time without compromise the quality of this activity.

Structural testing is a technique widely employed to validate software. This technique examine the internal structure of the program and thus ensure that functional requirements were tested. A limitation of this technique is the high number of test requirements to be covered and, consequently, the identification of infeasible test requirements (Vergilio et al., 2006; Yates and Malevris, 1989; Yates and Malevris, 1989; Ngo and Tan, 2008; Delahaye et al., 2015). An element is infeasible if there is no input data that leads to the execution of that element (Clarke, 1976; Frankl, 1987). The structural testing criteria use a coverage measure to assess the evolution of the testing activity, helping to decide if the software is being sufficiently tested. As the test activity is performed, new test cases are generated to execute required uncovered elements and thereby improve the testing coverage. In this process, a problem faced is the non-executable, which refers to required elements (e.g. paths) that, due to program semantics, are infeasible.

Most software has infeasible test requirements due to the program semantics. These infeasible test requirements interfere in the coverage evolution and progress of the testing activity when generating input data and the identification of them depends direct on tester skills (Barhoush and Alsmadi, 2013; Frankl, 1987; Bueno and Jino, 2000; Ngo and Tan, 2008; Yates and Malevris, 1989). This problem is recognized as a limitation of test activity as it depends on tester analysis and is an undecidable problem, so completely automating its determination is a difficult task (Clarke, 1976).

Some works have proposed approaches to face this issue due to the impact of infeasible test requirements on structural testing (Frankl, 1987; Bueno

[a] https://orcid.org/0000-0001-6504-7932
[b] https://orcid.org/0000-0001-5242-4938
[c] https://orcid.org/0000-0001-9761-1999

Neto, J., Mori, A., Vilela, R., Colanzi, T. and S. de Souza, S.
How to Identify the Infeasible Test Requirements using Static Analyse? An Exploratory Study.
DOI: 10.5220/0010497107820789
In *Proceedings of the 23rd International Conference on Enterprise Information Systems (ICEIS 2021) - Volume 1*, pages 782-789
ISBN: 978-989-758-509-8

and Jino, 2000; Barhoush and Alsmadi, 2013). The problem of non-executability presents itself in several program categories. Treating and mitigating the problem is not a trivial task and, for this reason, different approaches have been proposed. Symbolic execution, control-flow and data-flow analyses, branch correlation, polymorphic call are examples of used approached (Kundu et al., 2015; Papadakis and Malevris, 2010; Wang et al., 2014; Du and Dong, 2011; Pathade and Khedker, 2018; Ngo and Tan, 2008). The studies aforementioned above relied on input data to find infeasible elements, increasing the cost and testing effort. Despite the diversity of proposals, we found a relevant research gap related to features in the source code to find infeasible test requirements statically and with no input data.

Based on the literature and existing techniques, it was observed whether there are features present in the source code that enhances the generation of infeasible test requirements. These features can be organized into properties that can support the identification of infeasible test requirements. The properties can be separated into two groups. Group 1: properties where the infeasible test requirements are revealed through input data. Group 2: properties where the infeasible test requirements are revealed without input data. We did not find in the literature studies that analyze and/or measure the identification of infeasible test requirements through properties from Group 2.

This paper presents a compilation of a catalog of properties to support the identification of infeasible test requirements without input data. We built the catalog based on seven studies that discussed characteristics in the source code that revealed the infeasible behavior. The cataloging process considered properties that can logically and systematically applicable without dependence on input data. Therefore, the catalog is composed of five properties. [**P1**] Assignment of a constant value to a variable (Bodík et al., 1997; Vergilio et al., 1992). [**P2**] Opposite Predicates - Equal Predicates (Vergilio et al., 2006; Ngo and Tan, 2007). [**P3**] Correlation between conditional statements (Ngo and Tan, 2008; Bodík et al., 1997). [**P4**] Change of the definition of a variable during the analyzed path (Ngo and Tan, 2007; Hedley and Hennell, 1985) and [**P5**] Analyzes the existence of dead code (Barhoush and Alsmadi, 2013).

Our work presents an alternative process to identify infeasible test requirements. We developed an exploratory study to investigate the applicability of the catalog of properties to identify code snippets that lead to infeasible test requirements. The process uses the catalog to support the identification of code regions that are likely to generate infeasible required elements. Unlike existing process, our study statically identifies requirements before executing the code.

The exploratory study manually applied the properties cataloged in a set of 19 Java programs, available in (Ziviani, 2010). These programs present data structures normally employed in information systems. The suspicious code region were identified and marked with flags. We based the application of the flag on the guidelines formulated in the catalog. Later, based on the regions identified in the source code, we calculated the number of required elements that reaches the suspicious region and which ones are, in fact, infeasible required elements.

The results indicate that the proposed process is effective; however, we realize the importance of the tester in the process of mitigating the problem of infeasible requirements, since the tester's experience implies a speed of decision making. Finally, this study shows that an automated process that does not use input data is promising in addressing the problem of infeasible test requirements.

This paper is structured as follows: Section 2 presents how the properties were cataloged showing examples of their application. Section 3 presents the exploratory study, including its design, execution, and main results. Section 4 presents external and internal threats of our study. Section 5 summarizes the conclusions and make suggestions for future work.

2 CATALOG OF PROPERTIES FOR LOCALIZATION OF THE INFEASIBLE TEST REQUIREMENTS

This section presents a catalog with properties for identifying infeasible test requirements that do not use input data. The cataloged properties were taken from records in the literature and were organized based on their logic and systematic identification. We discuss the characteristics and functioning of the properties in detail and explained them through the complimentary examples.

P1: Assignment of a Constant Value to a Variable (Hedley and Hennell, 1985; Vergilio et al., 1992)**.** A constant assigned to a variable may imply a specific conditional direction causing an infeasible test requirement.

Property P1 is based on the dependency relationship that can exist between two instructions in the code, which are: a constant assignment statement and the conditional statement that uses the constant for verification. The code in Listing 1 provides an example of this property.

When an instruction denotes a constant in the code (e.g., in line *n1* (*final int x = 10;*)), its value will be store in memory at the compilation time, and it is not possible for any change of definition is allowed during code execution. Therefore, a statement of the conditional type, e.g. *IF* or *WHILE*, that uses the constant *x* (line *n1* of Listing 1) in its comparison structure may create an infeasible test requirement. Listing 1, constant *x* has the value 10 assigned, and it will be fixed to the constant at compilation time. Even if another variable uses constant *x*, the fixed value will still exist.

```
/* n1     */ final int x = 10;
/* n...   */ ...
/* n25    */ int age = x;
/* n...   */ ...
/* n175   */ if (name="Joao" && age >
    18){
/* n176   */         System.out.printl("
    True");
/* n177   */ } else {
/* n178   */         System.out.printl("
    False");
/* n179   */ }
/* n...   */ ...
/* n575   */ if (x > 21 && height <
    1.80){
/* n576   */         System.out.printl("
    True");
/* n577   */ } else {
/* n578   */         System.out.printl("
    False");
/* n579   */ }
/* n...   */ ...
/* n1000  */}
```

Listing 1: Example of Property P1.

We infer that after a few lines of code variable ***age*** (line *n25* of Listing 1) have being used in the conditional instruction of line *n175*, there will be no input data that covers any requirement that uses instruction set (*n175, n176*), a branch or path, for example. Likewise, if the constant *x* is used on line *n575*, there will be no input data that covers any requirement that uses instruction set (*n575, n576*), a branch or path, for example.

P2: Opposite Predicates - Equal Predicates (Vergilio et al., 2006; Ngo and Tan, 2007)**.** A dependency between opposite or equal predicates in one path may lead to infeasible paths because of the conditional assessment of one predicate interfering with the evaluation of the following predicate.

Property P2 is based on the dependency relationship that can exist between two instructions of the conditional type, exclusively the instruction of the *IF* type. The dependency can exist both for equal instructions (*if* $a < b$) and (*if* $a < b$) and for opposite instructions (*if* $a < b$) and (*if* $a > b$). Listing 2 provides an example of this property.

```
/* n1     */ ...
/* n...   */ ...
/* n175   */ if (a > b){
/* n176   */         System.out.println("A
    ");
/* n177   */ } else{
/* n178   */         System.out.println("!
    A");
/* n179   */ }
/* n...   */ ...
/* n275   */ if (a > b){
/* n276   */         System.out.println("B
    ");
/* n277   */ } else{
/* n278   */         System.out.println("!
    B");
/* n279   */ }
/* n...   */ ...
/* n375   */ if (c > d){
/* n376   */         System.out.println("C
    ");
/* n377   */ } else{
/* n378   */         System.out.println("!
    C");
/* n379   */ }
/* n...   */ ...
/* n475   */ if (c < d){
/* n476   */         System.out.println("D
    ");
/* n477   */ } else{
/* n478   */         System.out.println("!
    D");
/* n479   */ }
/* n...   */ ...
/* n1000  */}
```

Listing 2: Example of Property P2.

We can associate the dependency relation denoted by P2 with a truth table of the logical expression XOR and XNOR (Table 1). In this table, there are always two equal lines and two exclusive lines.

Listing 2 present a dependency between two equal conditional statements in lines *n175* and *n275*. On the basis of these two conditional, four different paths were derived to be covered. This denotes the four possibilities of the XOR truth table, where two of them are infeasible. The possible cases are those that the condition is false or true, for both predicates. The possible cases are those that the condition is false or true, for both predicates. *0 1* and *1 0*, requirements are infeasible. Any test requirement that is in the following paths: *n175, 176, ..., n277, n278* and *n178, n179, ..., n275, n276*, is infeasible. For the case of opposite instructions, *n375* and *n475*, the problem is the same, but, unlike the previous example, the truth table is XNOR. Therefore, any instruction that contemplates the exclusive cases will be feasible, since the cases of equal predicate will be infeasible. Afterward, any test requirement that traverses the following paths: *n375, n376, ..., n475, n476* and *n378, n379, ..., n478, n479*, is infeasible.

Table 1: Truth Table.

XOR	0	0	**1**	**XNOR**	0	0	**0**
XOR	0	1	**0**	**XNOR**	0	1	**1**
XOR	1	0	**0**	**XNOR**	1	0	**1**
XOR	1	1	**1**	**XNOR**	1	1	**0**

P3: Correlation between Conditional Statements (Ngo and Tan, 2008; Bodík et al., 1997)**.** Correlation between conditional statements along the way can cause infeasible test requirements. Property P3 is similar to P2, however, we deal with the existing dependency between a conditional of type *IF* and type *WHILE*.

```
/* n1     */ ...
/* n2     */ b = 100;
/* n...   */ ...
/* n175   */ while (a > b && short=="
    true"){
/* n176   */         System.out.println("
    while");
/* n...   */ ...
/* n276   */         if (a < 100){
/* n277   */                 System.out.
    println("A");
/* n...   */                 ...
/* n300   */         }
/* n...   */ ...
/* n1000    */ }
```

Listing 3: Example of Property P3.

Property P3 is based on the dependency relationship that can exist between two instructions in the code, which are: a conditional instruction of type *IF* and a loop instruction of type *WHILE*. The purpose of this property is to identify code snippets with great potential to present an infeasible test requirement. If the predicate in the *IF* statement is equal to or opposite to the predicate in the WHILE statement, it means that, as in the case of P2, the behavior of the paths will be according to the XOR or XNOR truth table. Listing 3 provide an example for our discussion.

Lines *n175* and *n276* of Listing 3 are dependent because they use the same variable ***a***. If the variable *a* does not change definition until the line *n276*, then there will be at least one infeasible test requirement, the one in which the conditions are opposite. In the cases of *IF* presenting an *ELSE* there will be infeasible test requirements when the conditions are the equals.

P4: Change of Definition of a Variable During the Analyzed Path (Ngo and Tan, 2007)**.** Changing the definition of a variable can generate an infeasible path in the presence of loops.

```
/* n1     */ ...
/* n2     */ int i = 11;
/* n175   */ while (a > b && short=="
    true"){
/* n176   */         System.out.println("
    while");
/* n...   */ ...
/* n276   */         if (buyTool){
/* n277   */                 System.out.
    println("A");
/* n...   */                 ...
/* n300   */         } else{
/* n301   */                 i = -1;
/* n302   */         }
/* n...   */         ...
/* n476   */         if (a < 100 && i > 10)
    {
/* n477   */                 System.out.
    println("A");
/* n...   */                 ...
/* n500   */         }
/* n501   */ }
/* n...   */ ...
/* n1000    */ }
```

Listing 4: Example of Property P4.

Property P4 was based on the dependency relationship that can exist between variables (use and definition), conditional structures, and loop structures. This property have a set of feature that must be fulfilled: (i) the variable must be defined before or during the repetition loop; (ii) the variable must be used inside the repetition loop by an instruction of the conditional type and; (iii) the variable must have a definition changed within the repetition loop. These features can help to reveal code snippets that are prone to infeasible test requirements. However, due to the number of features this property deals with, it is not possible to state that the identified instructions reveal an infeasible test requirement. In this context, the tester´s expertise is needs to be considered to analyze whether the occurrence of P4 is in fact an infeasible test requirement or not.

P4 can be illustrated by analyzing, in Listing 4, the lines **n2** (variable definition), **n175** (loop), **n301** (change of variable definition) and **n476** (conditional). In this case, due to the change of definition of variable *i* in line **n301** before the conditional instruction in line **n476**, it will not be possible to cover the positive condition of line **n477**, thus there being an infeasible test requirement.

P5: Analyzes the Existence of Dead Code (Barhoush and Alsmadi, 2013)**.** Logically inconsistent predicates related to dead codes cause an infeasible test requirement due to no possibility of execution of the code path. Property P5 was based on the identification of snippets of the code that are not used during the execution of the code and must be tested. All existing classes in the program must be tested, if there is a class not instantiated, then it will not be possible to cover any type of test criteria. In this way, this property searches for class-type instructions that are not called at any time in the entire program.

An example of a dead code was presented in Listing 5 and 6, ***Cfc*** class (Listing 5) was not used in the main program (Listing 6). ***Cfc*** class is a disconnected snippet of the main program, and, in reason of this scenario, do not allow test input to test the snippet of the code. Then, any requirement existing on lines *n1* to *n500* will be infeasible.

```
/* n1     */  ...
/* n2     */  public class Cfc {
/* n...   */  ...
/* n500   */  }
```

Listing 5: Example of Property P5. Class Cfc.

```
/* n1     */  ...
/* n2     */  public class Main {
/* n...   */  ...
/* n155     */      public static void
   main(final String[] args) {
/* n...   */            ...
/* n...   */            OtherClass other =
   new OtherClass();
/* n...   */            ...
/* n1000    */      }
```

Listing 6: Example of Property P5. Main code.

3 EXPLORATORY STUDY DESIGN

Several areas use exploratory studies to analyze and improve concepts, methods, and techniques (Baral and Offutt, 2020; Zhu et al., 2018; Rodrigues and Brancher, 2019). Our major objective in this study was to analyze the application process of the Properties for localization of the infeasible test requirements and measure their effectiveness. For this, we aim to answer a research question:

RQ1: *Are the cataloged properties useful for the identification of infeasible test requirements?*

For this, we conducted an exploratory study with its design inspired by empirical studies in several domains (e.g., mutation testing, testing robotic systems, and educational games) (Baral and Offutt, 2020; Koc et al., 2019; Rodrigues and Brancher, 2019). The study followed a strategy based on the concepts of empirical software, and we collected quantitative and qualitative data (Shull et al., 2001; Baral and Offutt, 2020; Koc et al., 2019; Rodrigues and Brancher, 2019). As a result, the experimental study was defined as follows.

3.1 Data-set

Java programs were taken from (Ziviani, 2010). The set brought together 19 programs that were developed for academic purposes and theoretically they do not have faults. These programs were selected because they are commonly used in the literature and their data structure are normally present in enterprise software systems. Moreover, these programs present characteristics found in real programs, such as definition and use of variables, changes in variable definition, repetition structures, decision structures, classes and methods. The source codes of the programs and a brief description of their features are present on GitHub [1].

In our study, we used *all-use* testing criterion (Rapps and Weyuker, 1985). This criterion requires that all associations between definition and use of a variable be exercised by at least one test case. Each association to be covered is formed by the definition of a variable and its consequent use. A definition occurs when a value is assigned to a variable; the use occurs when the variable value is employed in computation or an expression. The criterion was applied with the support of Baduíno testing tool [2].

3.2 Procedure

Data-set programs were analyzed looking for the satisfaction of properties, taking into account the properties catalog. When the properties appear in the program, their localization is marked with a flag that identifies which properties are presented into the program (e.g., P1, P2). The programs were executed in Baduino testing tool, and the list of required elements from the all-uses testing criterion was obtained. In the next step, the obtained required elements were analyzed to verify which one is affected by the flags inserted into programs. If the required element is affected, it is marked as probably infeasible.

[1] https://github.com/JoaoChoma/iceis2021
[2] https://github.com/saeg/baduino

3.3 Data Collection

Quantitatively, we calculate the number of lines of the codes and the required elements affected by the properties. Qualitatively, we analyzed the process of identifying the properties and how the properties can support the identification of infeasible test requirements.

3.4 Results

Table 2 includes some features of the results in question. The first column shows the analyzed program name, while the number of lines of the code of each program in the second column. The columns P1 to P5 indicate the number of code lines affected by each property. The column Affected LOCs shows the sum of the lines of the code affected by the properties. The column Test Requirements displays the number of test requirements generated by the all-use testing criterion. Finally, the last two columns indicate the number of test requirements affected by the properties and the number of identified infeasible requirements. The infeasible test requirements were identified by the authors, based on structure of each program.

The data-set and the results are organized in a repository in GitHub [3], promoting the replication of this study.

We applied the properties through the syntactic and semantic analysis of the programs to identify features that represent these properties. In the first step, we map the code snippets according to each investigated property. It is important to note that, in most cases, at least two lines of code were affected when a property was identified. The program analysis phase was the one that most demanded the tester's skills and time.

Through the syntactic and semantic analysis of code, we come across code snippets that present properties characteristics that cold cause an infeasible test requirement but up not result in an infeasible test requirement. This characteristic makes the testing activity more challenging. Although the localization of the property can be performed automatically, the determination of infeasible test requirements is based on the tester's skills.

Determining an infeasible test requirement is an error-prone and costly task, even for a tester with exceptional expertise. Also, an incorrect determination of an infeasible test requirement may not guarantee that a fault can be identified in the program under test. In view of this, we believe that the use of recommendation and optimization algorithms can be useful to decrease the tester's effort and prevent infeasible test requirements from being incorrectly classified.

[3]https://github.com/JoaoChoma/iceis2021

We noted that there are not tools for automation of infeasible test requirements identification without the use of test input in the literature. Therefore, we recognize the need to automate and systematize the process of identifying infeasible test requirements. An automated approach contributes to reduce the tester's work and decreasing the cost of generating input data. This reduction is because the infeasible test requirements will no longer be included in structural testing. Despite reducing the tester's work, the automation will not replace him/her as there are properties that need the tester's contribution to increasing their effectiveness. For this reason, we realized that including human perception in the automation process will be beneficial to the treatment of the infeasible test requirements.

Finally, we can answer the research question that motivated this experimental study, *Are the cataloged properties useful for the identification of infeasible test requirements?*. On the basis of the results of the exploratory study, we believe that the application process of the cataloged properties was effective. Once test requirements infeasible was found, even in programs constantly investigated in the literature.

4 THREATS TO VALIDITY

We recognize the following external and internal threats that could have affected the validity of our results.

A possible threat to external validity refers to the generalization of the results because our data-set samples were not selected under any rigor, they were selected due to their low level of complexity and because they are a popular reference. Therefore, we cannot generalize the results of this exploratory study, as it is not possible to guarantee that the set of 19 programs used are sufficiently representative and free from bias. Moreover, in this study, programs written in Java were used, so we cannot guarantee the generalization of results for other programming languages. Although the evidence shows that the cataloged properties were modeled in a generic way, to apply them in different contexts, new studies regarding other scenarios must be conducted to mitigate this threat. Another threat to external validity refers to the size of the programs used in the experiment. The programs are not as complex as real programs, however, as we are in an initial and exploratory study, we understand that the complexity of the data-set can be increased as the process matures.

Table 2: Data collected during the exploratory study.

Programs	LOC	P1	P2	P3	P4	P5	Affected LOCs	Test Requirements	Affected Requirements	Infeasible Requirements
MaxMin1	13	0	0	0	4	0	4	38	19	0
MaxMin2	14	0	0	0	4	0	4	38	19	1
MaxMin3	25	2	6	0	10	0	18	100	15	2
MergeSort	23	2	0	2	4	0	8	83	36	0
Sort	71	2	2	2	18	0	24	244	65	0
Fibonacci	11	2	0	0	4	0	6	16	12	0
StackArray	25	4	0	0	8	0	12	21	10	0
QueueArray	27	2	0	0	4	0	6	29	10	0
HeapSort Max	25	0	4	10	8	0	22	57	26	0
HeapSort Max 2	62	0	4	10	10	0	24	145	57	0
HeapSort MinInd	59	0	0	2	12	0	14	165	56	0
BinaryTree	88	0	16	0	14	0	30	159	72	24
Hashing	77	14	10	4	4	0	32	103	7	0
DepthFirstSearch	42	10	2	0	8	0	20	77	16	6
BreadthFirstSearch	57	12	2	0	4	0	18	105	31	3
Graph	85	2	4	2	40	0	48	179	79	9
PrimAlg	45	6	0	0	6	0	12	94	11	0
ExactMatch	54	4	0	6	10	0	20	210	68	0
AproximateMatch	28	2	0	0	8	0	10	85	8	0

The threats to internal validity include the construction of the property catalog, to mitigate the threat, the construction of the catalog was inspired on consolidated studies published on relevant bases. Another threat to internal validity is in the developed process used to carry out the exploratory study, to mitigate this threat, we inspired on empirical studies that explored concepts (Baral and Offutt, 2020; Koc et al., 2019; Zhu et al., 2018; Rodrigues and Brancher, 2019) and we base our exploratory process on the study (Shull et al., 2001) that defines guidelines for the development of empirical studies.

5 CONCLUSIONS

The problem of non-executability presents itself in several program categories and mitigating the problem is not a trivial task and, for this reason, different approaches have been proposed. The exploratory study showed that is possible to identify infeasible test requirements without using test data. The proposed process reduces the cost of generating test data allowing to statically mitigate the problem of non-executability. To develop this process, we designed a exploratory study to measure the applicability of a catalog of properties surveyed in the literature. As it was an preliminary study, the results are promising and will inspire new directions.

In general, properties presented in the literature can be classified into two major groups, the group of properties that uses input data to identify infeasible elements and those that do not need. As the properties that uses input data are handled by the automatic generation of test data through brute force, we identified that there was a need to improve the process to identify infeasible test requirements. Nevertheless, we realized the need to assess the effectiveness of properties that do not use input data to identify infeasible requirements.

The contributions of this study are the process and the catalog of properties for the identification of test requirements. The process is static and does not use input data. Besides, we reveal that the properties are effective in identifying snippets in the source code where there is a problem of infeasible requirements, and; we understand that the application process can be automated due to its systematization. Also, we suggest optimization and recommendation algorithms aid the tester in deciding whether a required element achieved by an infeasible property is an infeasible element.

Furthermore, this study highlighted other research questions that have not yet been analyzed: *Can the properties be applied in other program languages? Can the properties be applied to other programming paradigms?* (e.g. concurrent programs) *Is the automation process trivial? How much will the process of identifying infeasible requirements reduce the cost* *of generating test data? How much automation process will reduce the testator's workload?*

We propose the following future studies with target to mature research: Design an alternative approach for application of properties to identify infeasible test requirements without input data. Implementation of the automation proposal. Measure the number of eliminated infeasible test requirements during the automated process. Other proposals will certainly emerge along this path. Therefore, this exploratory study gave us the initial impetus to develop a new approach to address the problem of non-executability.

ACKNOWLEDGEMENTS

The authors acknowledge the São Paulo Research Funding, FAPESP, for the financial support under processes 2018/25744-6 and 2019/06937-0.

REFERENCES

Baral, K. and Offutt, J. (2020). An empirical analysis of blind tests. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 254–262.

Barhoush, B. and Alsmadi, I. (2013). Infeasible paths detection using static analysis. *Ijj.Acm.Org*, II(Iii).

Bodík, R., Gupta, R., and Soffa, M. L. (1997). Refining data flow information using infeasible paths. *ACM SIGSOFT Software Engineering Notes*, 22(6):361–377.

Bueno, P. M. S. and Jino, M. (2000). Identification of potentially infeasible program paths by monitoring the search for test data. *Proceedings ASE 2000: 15th IEEE International Conference on Automated Software Engineering*, pages 209–218.

Clarke, L. A. (1976). A system to generate test data and symbolically execute programs. *IEEE Transactions on software engineering*, (3):215–222.

Delahaye, M., Botella, B., and Gotlieb, A. (2015). Infeasible path generalization in dynamic symbolic execution. *Information and Software Technology*, 58:403–418.

Du, Q. and Dong, X. (2011). An improved algorithm for basis path testing. *BMEI 2011 - Proceedings 2011 International Conference on Business Management and Electronic Information*, 3:175–178.

Frankl, P. G. (1987). *The Use of Data Flow Information for the Selection and Evaluation of Software Test Data*. PhD thesis, New York, NY, USA. AAI8801533.

Hedley, D. and Hennell, M. A. (1985). The causes and effects of infeasible paths in computer programs. pages 259–266.

Koc, U., Wei, S., Foster, J. S., Carpuat, M., and Porter, A. A. (2019). An empirical assessment of machine learning approaches for triaging reports of a java static analysis tool. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 288–299.

Kundu, D., Sarma, M., and Samanta, D. (2015). A uml model-based approach to detect infeasible paths. *Journal of Systems and Software*, 107:71–92.

Ngo, M. N. and Tan, H. B. K. (2007). Detecting large number of infeasible paths through recognizing their patterns. *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering - ESEC-FSE '07*, page 215.

Ngo, M. N. and Tan, H. B. K. (2008). Heuristics-based infeasible path detection for dynamic test data generation. *Information and Software Technology*, 50(7-8):641–655.

Papadakis, M. and Malevris, N. (2010). A symbolic execution tool based on the elimination of infeasible paths. *Proceedings - 5th International Conference on Software Engineering Advances, ICSEA 2010*, pages 435–440.

Pathade, K. and Khedker, U. P. (2018). Computing partially path-sensitive mfp solutions in data flow analyses. *Proceedings of the 27th International Conference on Compiler Construction*, pages 37–47.

Rapps, S. and Weyuker, E. J. (1985). Selecting software test data using data flow information. *IEEE transactions on software engineering*, (4):367–375.

Rodrigues, L. A. L. and Brancher, J. D. (2019). Playing an educational game featuring procedural content generation: which attributes impact players' curiosity?

Shull, F., Carver, J., Bldg, A., and Travassos, G. (2001). An empirical methodology for introducing software processes. *ACM SIGSOFT Software Engineering Notes*, 26.

Vergilio, S. R., Maldonado, J. C., and Jino, M. (1992). Non-executable paths: Characterization, prediction and determination to support program testing - in portuguese.

Vergilio, S. R., Maldonado, J. C., and Jino, M. (2006). Infeasible paths in the context of data flow based testing criteria: Identification, classification and prediction. *Journal of the Brazilian Computer Society*, 12(1):73–88.

Wang, Y., Xing, Y., and Zhang, X. (2014). A method of path feasibility judgment based on symbolic execution and range analysis. *International Journal of Future Generation Communication and Networking*, 7(3):205–212.

Yates, D. and Malevris, N. (1989). Reducing the effects of infeasible paths in branch testing. *ACM SIGSOFT Software Engineering Notes*, 14(8):48–54.

Zhu, Q., Panichella, A., and Zaidman, A. (2018). An investigation of compression techniques to speed up mutation testing. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 274–284.

Ziviani, N. (2010). *Algorithms Project with Implementations in JAVA and C ++ (in Portuguese)*. Cengage Learning Edições Ltda.