Automatic Support for the Identification of Infeasible Testing Requirements

João Choma Neto
joaochoma@usp.br
Institute of Mathematics and Computer Sciences, University of São Paulo
São Carlos, São Paulo, Brazil

ABSTRACT

Software testing activity is imperative to improve software quality. However, finding a set of test cases satisfies a given test criterion, is not a trivial task because the overall input domain is very large, and different test sets can be derived, with different effectiveness. In the context of structural testing, the non-executability is a feature present in most programs, increasing cost and effort of testing activity. When concurrent programs are tested, new challenges arise, mainly related to the non-determinism. Non-determinism can result in different possible test outputs for the same test input, which makes the problem of non-executability more complex, requiring treatment. In this sense, our project intends to define an approach to support automatic identification of infeasible testing requirements. Hence, this proposal aims to identify properties which cause infeasible testing requirements and automate their application. Due to complexity of the problem, we will apply search-based algorithms in the automation of concurrent and sequential programs treatment.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Empirical software validation**; Search-based software engineering.

KEYWORDS

Software Testing, Structural Testing, Infeasible Path Problem, Search Based Software Testing

ACM Reference Format:
João Choma Neto. 2020. Automatic Support for the Identification of Infeasible Testing Requirements. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3395363.3402646

1 INTRODUCTION

Testing techniques and criteria have been proposed to systematize the testing activity. Among them, structural testing, also called white-box test, uses information about internal structure of system to derive test cases [20]. Each technique presents a set of testing criteria, which systematize the test activity, assisting in two aspects: 1) in the selection and/or generation of test data, and 2) in the decision when this activity can be completed.

Structural testing criteria can use a measure of coverage of testing requirements for evaluating the testing activity progress, helping to decide if tests are enough. This measure compares how many testing requirements were covered by test data set used. The process of applying test criteria is faced with the problem of non-executability also known as the problem of infeasible paths. This problem is recognized as a limitation of testing activity and it depends directly on the tester's skills [6]. The problem of non-executability extends to any testing requirements, so we will focus on infeasible testing requirements problem. Most software systems have infeasible testing requirements [4, 10] what negatively interfere in coverage analysis and evolution of the testing activity.

The infeasible testing requirements problem affects the effort of applying structural testing criteria [8, 22, 30, 32]. This raises the cost of generating test cases what may prevent the program from achieving satisfactory levels of quality during static analysis. One solution to this problem would be to remove infeasible paths from testing process, leading to improvements in code coverage results, although such action is not trivial [18].

In other perspective, concurrent programming is an essential paradigm to reduce computational time in many application domains (e.g. for instance, web servers). A concurrent program is composed of concurrent (or parallel) processes (or threads) which interact to solve a complex problem. This interaction may occur synchronously or not, in which these processes may or not compete for the same computational resources [23]. Concurrent programs are more complex compared to sequential programs due to the following features: [9, 23]: non-deterministic behavior, large number of synchronization sequences, new type of defects, and high complexity for automatic generation of test data. The problem of infeasible testing requirements is aggravated in concurrent programs due to non-determinism in the generation of synchronization sequences.

Although detection of all infeasible testing requirements is an intractable problem [21], many approaches have been proposed to support partial identification of infeasible testing requirements. Among the techniques developed are: symbolic evaluations, manual identification in control and data flow graphs, branch correlation and polymorphic call [15]. Other approaches have been developed based on heuristics [4, 17] and others have proposed code patterns for detecting of infeasible paths [21]. Research on infeasible testing requirements in concurrent programs focuses on eliminating its occurrence, for example, with the proposition of Reachability Testing [16], but with a high application cost. Other works explore

ISSTA '20, July 18–22, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8008-9/20/07…$15.00
https://doi.org/10.1145/3395363.3402646

static analysis and symbolic executions [5, 27] and data stream analysis and execution monitoring [1].

A new approach using search algorithms has emerged to address problems considered difficult to solve in Software Engineering. Since 1976, the number of publications involving software testing using search-based algorithms have increased, giving rise to the term Search Based Software Testing (SBST) [19]. The SBST is used to solve or minimize problems in the context of: functional testing, structural testing, integration testing, regression testing, stress testing, mutation testing, test prioritization, model-based testing and exception tests [19]. In this context, a gap regarding the use of SBST to address the problem of infeasible synchronization sequences in concurrent programs is observed. Although there are approaches which address the infeasible testing requirements problem, there are no studies that address the infeasible testing requirements problem in concurrent programs, using SBST techniques. The main benefit of this proposal is allows automatic identification of infeasible testing requirements, reducing cost of test data generation process.

Manual or automated detection of infeasible testing requirements prior to test data generation presents the following challenges: a) eliminating all infeasible testing requirements is a undecidable problem; b) the infeasible testing requirements problem is present in structural testing of sequential programs and is intensified in concurrent programs context, due to synchronization sequences; c) concurrent programs exhibit non-deterministic behavior, which produces a large set of possible synchronization sequences among processes; d) the computational cost of identifying infeasible paths at runtime is high; e) the cost of static identification of infeasible paths satisfactorily reduces the computational cost of generating test data; f) the search approach, often used for automatic test data generation, can be refined to identify infeasible paths; g) human perception hitherto used to detect infeasible paths can directly assist in the process of automatically detecting infeasible paths.

Previous works related to identification of infeasible testing requirements are input test dependencies, what favors using automatic test data generation as a means of identifying infeasible testing requirements. This approach demands a high computational cost and its effectiveness is not always achieved [4, 17]. In other approaches, infeasible testing requirements identification properties are applied manually and empirically. Despite using tester's experience, this approach loses its effectiveness due to cognitive exhaustion of the manual identification process [28].

We have identified a research gap focused on the need to define a set of properties for identifying automatically infeasible paths. Taking into account observations and the research gap, we consider that relevant properties should be automatically applied and without depending on the input data.

Studies indicate that infeasible paths are present in all software systems, impacting software testing activity, in particular, in analysis of test coverage and in generation of test data [8, 32]. The full automaton of the identification of infeasible testing requirements is undecidable problem [6]. The automatic determination of infeasible paths, although desired, is still a challenge, and use of search-based algorithms can be useful because it could mitigate the non-determinism of synchronization sequence formation in concurrent programs.

This paper presents a doctoral proposal which aims to address the problem of detecting infeasible testing requirements in structural testing of concurrent and sequential programs, with the support of search-based algorithms. For this, the paper is organized as follows: Section 2 presents motivations and objectives, Section 3 presents the proposal development methodology and progress and, finally, the Section 4 presents the expected results.

2 RESEARCH OBJECTIVES

The aspects which motivate the development of this doctoral project are: (i) the infeasible testing requirements problem is present in all programs and, in the context of concurrent programs, this problem is intensified due to existing communication and synchronization; (ii) there is no current approach that is capable of properly detecting the infeasible testing requirements, it is undecidable problem and its solution is not found in polynomial time; (iii) the detection of infeasible paths before the test data generation makes possible to drastically reduce the cost of this activity, allowing a more effective generation of test data; (iv) search-based algorithms, initially proposed to support the automatic generation of test data, can be adapted to identify infeasible paths; (v) human perception, previously used to manually detect infeasible testing requirements, can directly assist in the automatic detection process; (vi) concurrent programs have non-deterministic behavior which produces a high set of possible sequences of synchronization among processes, increasing the search space favoring the application of search-based algorithms.

Considering this scenario, the objective of this doctoral project is to investigate the application of search-based algorithms to support the identification of infeasible testing requirements. Initially, the proposal will be applied on sequential programs, which will serve as a basis to instantiate the proposal for concurrent programs. In this sense, this project intends to answer the following research question, ***Research Question:*** *How can search-based algorithms to be applied to support the identification of infeasible testing requirements?*. Based on the main research question, we identified other secondary questions that can be answered with our project.

Question 1: *Which properties for identification of infeasible testing requirements can be automatically applied?*, for the reason that literature does not present any process that automates the identification of infeasible testing requirements.

Question 2: *Which search-based algorithms resources are efficient to help the problem of infeasible testing requirements?*, for the reason that of non-determinism, computational, and cost to for automatic test data generation process, for concurrent programs.

Question 3: *Does the incorporation of human perceptions assist in the automatic identification of infeasible testing requirements?*, for the reason of non-determinism in synchronization sequences, and computational cost to automatic generation of test data, in concurrent programs.

Question 4: *Which benefits are achieved with the application of properties for identification of infeasible testing requirements in the structural testing activity?*, due to development of the doctoral project will directly benefit the software testing domain, specifically the structural test.

3 METHODOLOGY AND PROGRESS

To answer the research questions and build the proposed approach, we elaborated the following methodology:

- Step 1: Set properties to identify infeasible paths for concurrent and sequential programs.
- Step 2: Implement the algorithm for automatically applying properties.
- Step 3: Formalize the identification properties of infeasible testing requirements for concurrent programs.
- Step 4: Automate the properties developed in Step 3.
- Step 5: Use search-based algorithms to optimize the automation algorithm in the context of concurrent programs.
- Step 6: Incorporation of human perception in the process of identifying infeasible testing requirements.

In Step 1, we have developed a systematic mapping to identify studies which investigate the problem of non-executability in concurrent and sequential programs. We use Snowballing systematic methodology, which uses references and citations from seminal and highly cited studies to gather relevant studies to answer the research questions [24, 31]. The systematic mapping artifacts can be found at GitHub [1]. Among the results achieved in systematic mapping, we identify five properties that identify infeasible testing requirements, and that can be applied automatically:

- **P1 - Assignment of a constant value to a variable** [12, 29]: A constant assigned to a variable may imply a specific conditional direction causing an infeasible testing requirements.
- **P2 - Opposite Predicates - Equal Predicates** [21, 30]: a dependency between opposite or equal predicates in one path may lead to infeasible testing requirements due to the conditional assessment of one predicate interfering with the evaluation of following predicate.
- **P3 - Correlation between conditional statements** [3, 22]: correlation between some conditional statements along the can cause infeasible testing requirements.
- **P4 - Change of the definition of a variable during the analyzed path** [21]: Changing the definition of a variable can generate infeasible testing requirements in the presence of loops.
- **P5 - Dead codes** [2]: Logically inconsistent predicates related to dead codes cause infeasible testing requirements due to non-execution of the code element.

Some facts were obtained by mapping study: (i) source code properties that reveal potential infeasible testing requirements; (ii) which properties do not require input data to reveal infeasible requirements; (iii) the application of the properties can be automated, similar to automatic generation of input data [4, 7, 11, 13, 14, 22, 25, 26]; and, (iv) the application of property P4 needs help of tester to make decision about potential infeasible requirements.

Still in Step 1, we have executed an empirical study for evaluating the properties, manually applying in a set of sequential programs. We observed that properties reveal infeasible testing requirements without the need input data and can be applied automatically. Study artifacts can be found at GitHub [2].

An automated approach contributes to reduce tester's effort and decrease the cost of generating input data. This reduction is caused because infeasible requirements will no longer be considered in the structural testing. In spite of reducing tester's effort, the automation will not replace it once there are properties which need the tester's contribution to increase its effectiveness. Hence, we realized that including human perception in automation process, benefits to the treatment of the infeasible testing requirements problem can be achieved. The property identification algorithm will not be able to make decision in all cases, so on that occasion the tester will decide whether the testing requirements is feasible or infeasible. This procedure can later be replaced by a machine learning algorithm.

To start the development of Step 2, we outlined in, Figure 1, the main stages of the algorithm which automates the identification of infeasible testing requirements. Our goal is output generated by the approach will be a refined set of testing requirements that will reduce the cost of generating test data and consequently reduce the cost of testing activity.

Stage 1 - Instrumented code generation. This stage receives as input the code to be tested. This stage will be responsible for instrumenting the code, so that the control flow graphs are constructed.

Stage 2 - Selection and Application of Test Criteria. This stage will apply testing criteria to the instrumented code received as input. As output, it will generate a set of testing requirements. The testing criteria used will be those which can properties, e.g. data flow testing favoring Property P4.

Stage 3 - Application of properties. This stage will apply the identification properties to testing requirements generated in Stage 2. The application process will signal which testing requirements are infeasible or potentially infeasible. As a result of this stage, a set of feasible testing requirements and a set of infeasible testing requirements will be generated. The feasible set of testing requirements will be used in Application Test stage, while the infeasible set will be ignored.

Stage 4 - Test Application. This stage will cover the set of testing requirements received from Stage 3. The output will be the coverage rate that will be the criterion for stopping the process. If the coverage rate is satisfactory, then the process will be completed. On the other hand, else the testing requirements will be inserted again in Stage 3 and the identification process will be resumed.

In Step 2 We are building a set of sequential programs. Given the experience acquired, our proposal is use search-based to optimize $Stage\ 3 - Application\ of\ properties$ and, likewise, include human perception. The performance of identification algorithm will be measured based on the number of testing requirements identified and, consequently, an increase in the coverage rate of the test.

Since there are no properties for identifying infeasible testing requirements for concurrent programs, Step 3 will carry out an empirical study to identify properties for infeasible testing requirements in concurrent programs. Our research group has a set of concurrent programs that have been used in studies of automatic generation of test cases. Thus, we intend to empirically analyze the

[1] https://github.com/JoaoChoma/snowballing

[2] https://github.com/JoaoChoma/empiricalstudy

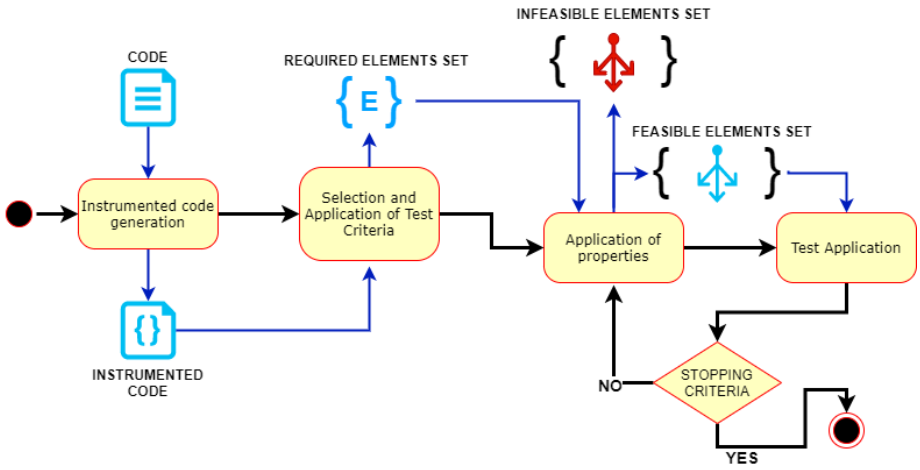

Figure 1: Envisioned Approach to Infeasible Testing Requirements Identification.

experimental results of the group and seek for properties that reveal infeasible synchronization sequences. In Step 4, we will extend the algorithm presented in Step 2 and automate the application of the properties identified in Step 3.

In Step 5, we will use search-based algorithms to optimize the algorithm of Step 4 to mitigate the infeasible synchronization sequences in concurrent programs. Initially we will use the genetic algorithm to search synchronization sequences that present properties of infeasible testing requirements. Later, it will be possible to compair the performance with other search algorithms.

Finally, in Step 6, will use tester's perception to ensure that all identified properties are classified, since only algorithm will not be able to make the decision in all cases. Step 6 will be applied in Steps 2 and 5.

Steps 4 and 5 will be evaluated with concurrent program source codes, already used in other studies in our research group, because these studies report difficulties related to existence of infeasible testing requirements. The performance of identification algorithm will be measured based on number of testing requirements identified and, consequently, an increase in coverage rate of the test.

4 EXPECTED CONTRIBUTIONS

We expect that our approach will be able to automatically identify infeasible test requirements and, thereby, minimize the incidence of the problem of non-executability, in sequential and concurrent programs. In addition, the use of search-based algorithms will optimize the process of identifying infeasible testing requirements in concurrent programs, which will minimize tester's effort to identify infeasible testing requirements manually. As a way of improvement the proposal, the use of human perception will adjust the decision-making process regarding the identification of infeasible testing requirements. As a future work, use of a machine learning algorithm to replace the tester in decision making with the aim of achieving 100% automation with high accuracy.

ACKNOWLEDGMENTS

The authors acknowledge the São Paulo Research Funding, FAPESP, for the financial support under process no. 2018/25744-6.

REFERENCES

[1] N Anastopoulos and N Koziris. 2008. Facilitating efficient synchronization of asymmetric threads on hyper-threaded processors. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on.* IEEE, 1–8.
[2] B Barhoush and I Alsmadi. 2013. Infeasible Paths Detection Using Static Analysis. *Ijj.Acm.Org* II, Iii (2013).
[3] R Bodík, R Gupta, and M L Soffa. 1997. Refining data flow information using infeasible paths. *ACM SIGSOFT Software Engineering Notes* 22, 6 (1997), 361–377.
[4] P M S Bueno and M Jino. 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* (2000), 209–218.
[5] IS Chung, Hyeon S Kim, H S Bae, Y R Kwon, and D G Lee. 1999. Testing of concurrent programs after specification changes. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on.* IEEE, 199–208.
[6] Lori A. Clarke. 1976. A system to generate test data and symbolically execute programs. *IEEE Transactions on software engineering* 3 (1976), 215–222.
[7] S S Dahiya, J K Chhabra, and S Kumar. 2011. *PSO Based Pseudo Dynamic Method for Automated Test Case Generation Using Interpreter.* 147 pages. arXiv:9780201398298
[8] M Delahaye, B Botella, and A Gotlieb. 2015. Infeasible path generalization in dynamic symbolic execution. *Information and Software Technology* 58 (2015), 403–418.
[9] Carl Dionne, Marc Feeley, and Jocelyn Desbien. 1996. A Taxonomy of Distributed Debuggers Based on Execution Replay.. In *PDPTA*. 203–214.
[10] P G Frankl. 1987. *The Use of Data Flow Information for the Selection and Evaluation of Software Test Data.* Ph.D. Dissertation. New York, NY, USA. AAI8801533.
[11] D Gong, T Tian, and X Yao. 2012. Grouping target paths for evolutionary generation of test data in parallel. *Journal of Systems and Software* 85, 11 (2012), 2531–2540.
[12] D. Hedley and M. A. Hennell. 1985. The causes and effects of infeasible paths in computer programs. (1985), 259–266.
[13] I. Hermadi and M. A. Ahmed. 2003. Genetic Algorithm based Test Data Generator. *The 2003 Congress on Evolutionary Computation, 2003. CEC '03.* May (2003), 184.
[14] I. Hermadi, C. Lokan, and R. Sarker. 2014. Dynamic stopping criteria for search-based test data generation for path testing. *Information and Software Technology* 56, 4 (2014), 395–407.
[15] D Kundu, M Sarma, and D Samanta. 2015. A UML model-based approach to detect infeasible paths. *Journal of Systems and Software* 107 (2015), 71–92.
[16] Y Lei and R Cz. 2005. A new algorithm for reachability testing of concurrent programs. In *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on.* IEEE, 10–pp.
[17] N. Malevris, D. F. Yates, and A. Veevers. 1990. Predictive metric for likely feasibility of program paths. *Information and Software Technology* 32, 2 (1990), 115–118.
[18] A W Marashdih and Z F Zaaba. 2018. Infeasible paths in static analysis: Problems and challenges. *AIP Conference Proceedings* 2016, September (2018).

[19] P. McMinn. 2011. Search-based software testing: Past, present and future. In *Software testing, verification and validation workshops (icstw), 2011 ieee fourth international conference on*. IEEE, 153–163.

[20] G J Myers, C Sandler, and T Badgett. 2011. *The art of software testing*. John Wiley & Sons.

[21] M N Ngo and H B K Tan. 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* (2007), 215.

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

[23] P Pacheco. 2011. *An introduction to parallel programming*. Elsevier.

[24] K Petersen, S Vakkalanka, and L Kuzniarz. 2015. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology* 64 (2015), 1–18.

[25] G H.L. Pinto and S R Vergilio. 2010. A multi-objective genetic algorithm to test data generation. *Proceedings - International Conference on Tools with Artificial Intelligence, ICTAI* 1 (2010), 129–134.

[26] SRS Souza, Silvia Regina Vergilio, PSL Souza, AS Simao, and Alexandre Ceolin Hausen. 2008. Structural testing criteria for message-passing parallel programs. *Concurrency and Computation: Practice and Experience* 20 (November 2008), 1893–1916. Issue 16.

[27] K Tai et al. 1999. Automated test sequence generation using sequencing constraints for concurrent programs. In *Software Engineering for Parallel and Distributed Systems, 1999. Proceedings. International Symposium on*. IEEE, 97–108.

[28] H. Takagi. 2015. Interactive evolutionary computation for analyzing human characteristics. In *Emergent Trends in Robotics and Intelligent Systems*. Springer, 189–195.

[29] S R Vergilio, J C Maldonado, and M Jino. 1992. Non-executable paths: Characterization, Prediction and Determination to Support Program Testing - In Portuguese.

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

[31] C Wohlin. 2014. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering* (London, England, United Kingdom) *(EASE '14)*. ACM, New York, NY, USA, Article 38, 10 pages.

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