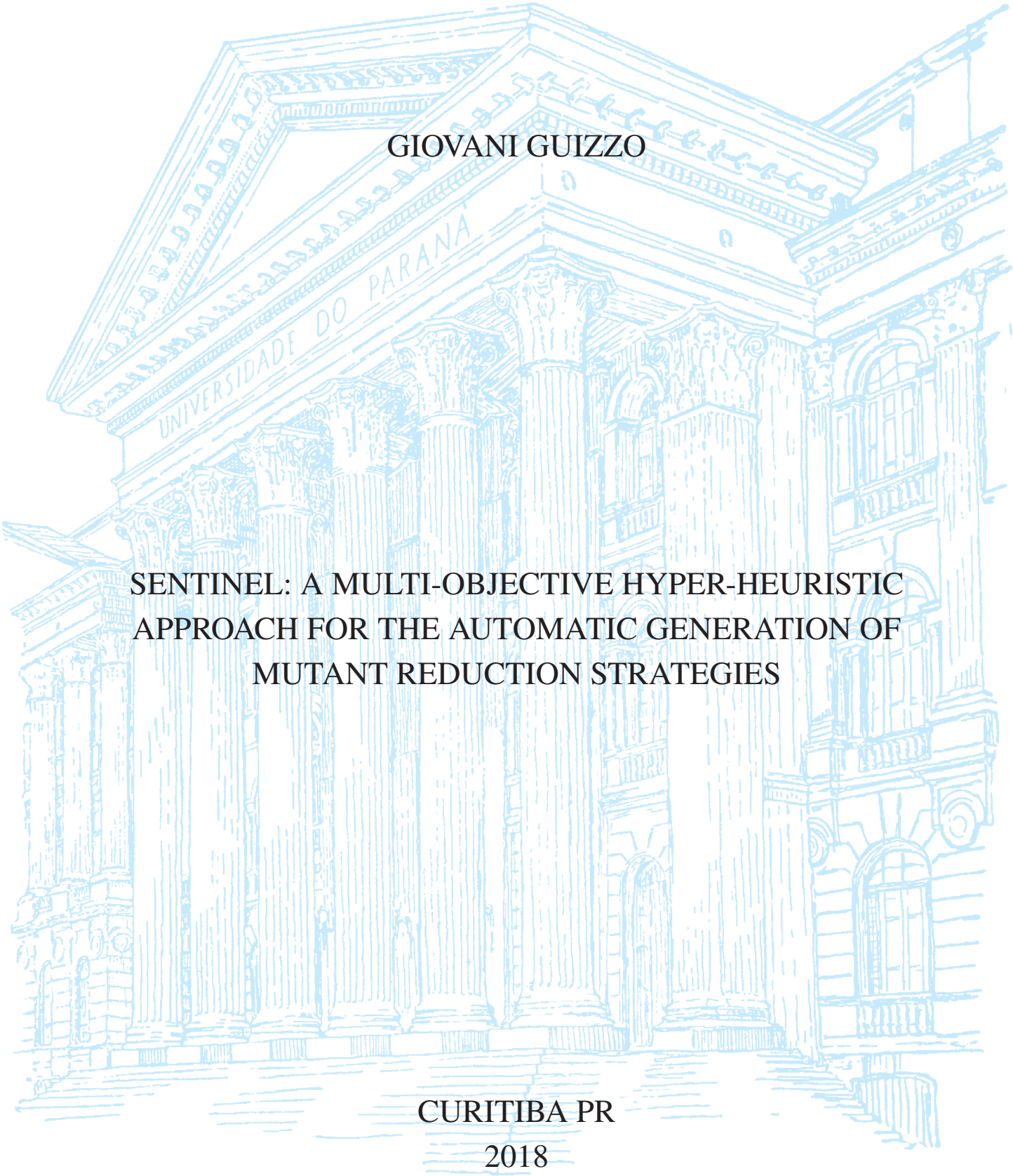
UNIVERSIDADE FEDERAL DO PARANÁ

GIOVANI GUIZZO



SENTINEL: A MULTI-OBJECTIVE HYPER-HEURISTIC APPROACH FOR THE AUTOMATIC GENERATION OF MUTANT REDUCTION STRATEGIES

CURITIBA PR
2018

GIOVANI GUIZZO

SENTINEL: A MULTI-OBJECTIVE HYPER-HEURISTIC APPROACH FOR THE AUTOMATIC GENERATION OF MUTANT REDUCTION STRATEGIES

Tese apresentada como requisito parcial à obtenção do grau de Doutor em Ciência da Computação no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de Concentração: *Ciência da Computação*.

Orientadora: Dr. Silvia Regina Vergilio.

Coorientadora: Dr. Federica Sarro.

CURITIBA PR
2018

Catalogação na Fonte: Sistema de Bibliotecas, UFPR
Biblioteca de Ciência e Tecnologia

G969s Guizzo, Giovani
Sentinel: a multi-objective hyper-heuristic approach for the automatic generation of mutant reduction strategies / Giovani Guizzo. – Curitiba, 2018.

Tese - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2018.

Orientadora: Silvia Regina Vergilio. Coorientadora: Frederica Sarro.

1. Software – Testes. 2. Engenharia de software. 3. Algorítmos computacionais. I. Universidade Federal do Paraná. II. Vergilio, Silvia Regina. III. Sarro, Frederica. IV. Título.

CDD: 005.10285

Bibliotecária: Vanusa Maciel CRB- 9/1928

MINISTÉRIO DA EDUCAÇÃO
SETOR SETOR DE CIENCIAS EXATAS
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO INFORMÁTICA

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da tese de Doutorado de **GIOVANI GUIZZO** intitulada: **Sentinel: A Multi-Objective Hyper-Heuristic Approach for the Automatic Generation of Mutant Reduction Strategies**, após terem inquirido o aluno e realizado a avaliação do trabalho, são de parecer pela sua aprovação no rito de defesa.

A outorga do título de doutor está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

Curitiba, 03 de Dezembro de 2018.

SILVIA REGINA VERGILIO
Presidente da Banca Examinadora

ANDREY RICARDO PIMENTEL
Avaliador Interno

FEDERICA SARRO
Coorientadora

SIMONE DO ROCIO SENGER DE SOUZA
Avaliador Externo

ARILO CLAUDIO DIAS NETO
Avaliador Externo

EDUARDO JAQUES SPINOSA
Avaliador Interno

To my parents.

Acknowledgements

First of all, I would like to thank my parents, Cristina Valiati and Joceli Guizzo, for all the support they have given me. Thanks for encouraging me to pursue my goals and to focus on my studies.

I would like to thank my advisor, Prof. Dr. Silvia Regina Vergilio. Without her supervision, this could not have been done. Thank you for standing by me all these years and thank you for teaching me so much about Computer Science and life. You are the best.

Special thanks to Dr. Jens Krinke and Dr. Federica Sarro from the CREST group at University College London, who spent their time to help me develop this work. Your knowledge and insights enhanced this thesis immensely.

Thanks to Prof. Dr. Aurora Pozo from the C-Bio group at UFPR, whose expertise in Artificial Intelligence supported my academic development during all these years.

Thanks to all my friends from the C-Bio GrES and CREST groups. I have learned a lot with you.

I would like to thank Microsoft for the Microsoft Azure Research Award granted to our work. This US$ 20,000.00 worth of cloud computing award was crucial for the experimental evaluation of Sentinel.

This work is also supported by the Brazilian funding agencies Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) and Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) under the grants: 307762/2015-7 and 473899/2013-2.

RESUMO

O Teste de Mutação utiliza programas mutantes gerados com pequenas modificações no programa em teste. Tais modificações são produzidas por operadores de mutação que descrevem possíveis defeitos. O objetivo é matar os mutantes com casos de teste que produzam diferentes saídas para os mutantes e para o programa original. Ao final, uma medida de avaliação do teste, o escore de mutação, é dada considerando o número de mutantes mortos em relação ao número de mutantes gerados. O teste de mutação é bastante eficaz em revelar defeitos, entretanto, possui um alto custo computacional relacionado à execução dos mutantes. Diferentes estratégias existem para reduzir o custo do teste de mutação. Por exemplo, um subconjunto de mutantes pode ser selecionado aleatoriamente, ou pode-se utilizar um conjunto menor de operadores, ou ainda algoritmos de busca para selecionar os melhores mutantes ou operadores sem reduzir o escore global. Experimentos reportados na literatura mostram que nenhuma estratégia pode ser considerada a melhor em todos os contextos. A determinação das melhores estratégias e suas combinações de parâmetros só é possível com a condução de experimentos comparando estratégias em cenários específicos, o que pode elevar o custo do teste. O uso de uma estratégia inadequada pode comprometer a eficácia do teste e implica em uma redução de custo insatisfatória. Uma abordagem automática para selecionar e configurar estratégias pode auxiliar o testador nesta tarefa. Hiper-heurísticas são opções viáveis para este propósito, uma vez que estas são utilizadas para selecionar ou gerar boas heurísticas (estratégias) ao invés de tentar resolver o problema diretamente. Hiper-heurísticas vêm sendo estudadas na literatura, e bons resultados foram obtidos na área de Engenharia de Software Baseada em Busca. Todavia, poucos trabalhos investigam a aplicação de hiper-heurísticas no teste de mutação, e nenhum deles auxilia na geração ou configuração de estratégias de teste de mutação. Diante disto, este trabalho propõe uma abordagem baseada em hiper-heurísticas chamada Sentinel. Essa abordagem tem como objetivo principal gerar automaticamente estratégias de redução de mutantes de modo a reduzir o custo do teste de mutação sem comprometer a eficácia do teste em termos de escore de mutação e a habilidade de revelar defeitos. Sentinel utiliza elementos provenientes de diversos tipos de estratégias de redução de mutantes e os combina de modo a gerar novas estratégias potencialmente mais efetivas. A ideia é que o testador execute Sentinel, reutilize as estratégias geradas automaticamente e consequentemente não necessite escolher e nem configurar as estratégias manualmente. Isso facilita a aplicação do teste de mutação e reduz assim o custo da atividade de teste como um todo. Para avaliar a viabilidade de Sentinel, múltiplos experimentos foram executados. Os experimentos foram conduzidos em 10 diferentes sistemas com 4 versões cada, totalizando 40 sistemas. As estratégias geradas por Sentinel se mostraram significativamente mais efetivas do que estratégias convencionais da literatura. Além disso, elas mantiveram os seus bons resultados em novas versões dos softwares, o que demonstrou a sua reusabilidade.

Palavras-chave: hiper-heurística, teste de software, geração de estratégias, redução de mutantes, evolução gramatical, engenharia de software baseada em busca, otimização multiobjetivo.

ABSTRACT

Mutation Testing uses mutant programs generated with small modifications in the program under test. Such modifications are produced by mutation operators that describe possible faults. The objective is to kill the mutants with test cases that produce different outputs for the mutants and the original program. In the end, a test assessment measure, called mutation score, is computed considering the number of dead mutants with respect to the number of generated ones. Mutation testing is very efficacious in revealing faults, however, it has a high computational cost related to the execution of mutants. There are several strategies to reduce the mutation testing cost. For instance, a subset of mutants can be randomly selected, or a smaller set of operators can be used, or even search based algorithms can be applied to select the best mutants or operators without reducing the global mutation score. Experiments reported in the literature show that no strategy has been proven to be the best one in all contexts. The determination of the best strategies and their parameter combination is only possible by conducting experiments comparing the strategies in specific scenarios, which can increase the testing cost. The use of an inadequate strategy might compromise the test efficacy and implies in unsatisfactory cost reduction. An automatic approach to select and configure strategies can assist the tester in this task. Hyper-heuristics are viable options for this end, since they are used for selecting or generating heuristics (strategies) instead of trying to solve the problem directly. Hyper-heuristics have been studied in the literature and good results were obtained in the Search Based Software Engineering field. However, few works have investigated the usage of hyper-heuristics in mutation testing, and none of them support the generation or configuration of mutation testing strategies. In light of this, this work proposes a hyper-heuristic based approach called Sentinel. This approach has as main objective the automatic generation of mutant reduction strategies for the cost reduction of mutation testing without compromising the test efficacy in terms of mutation score and ability in revealing faults. Sentinel uses features from several kinds of mutant reduction strategies and combines them to generate new strategies that are potentially more effective. The idea is that the tester executes Sentinel, reuses the automatically generated strategies and consequently does not need to select and configure strategies manually. This eases the mutation testing application and thus reduces the overall cost of the testing activity. For assessing the feasibility of Sentinel, multiple experiments were performed. The experiments were conducted using 10 different systems with 4 versions of each, in a total of 40 systems. The strategies generated by sentinel showed to be significantly more effective than conventional strategies of the literature. Furthermore, they kept their good results in newer versions of the software, which demonstrated their reusability.

Keywords: hyper-heuristic, software testing, strategy generation, mutant reduction, grammatical evolution, search based software engineering, multi-objective optimization.

List of Figures

List of Tables

List of Grammars

List of Acronyms

ACO	Ant Colony Optimization
AOP	Aspect Oriented Programming
CH	Choice Function
CPH	Competent Programmer Hypothesis
CPU	Central Processing Unit
DE	Differential Evolution
EA	Evolutionary Algorithm
EMT	Evolutionary Mutation Testing
FOM	First Order Mutant
GA	Genetic Algorithm
GE	Grammatical Evolution
GP	Genetic Programming
GPM	Genotype-Phenotype Mapping
GUI	Graphical User Interface
HC	Hill Climbing
HITO	Hyper-Heuristic for the Integration and Test Order Problem
HOM	Higher Order Mutant
HV	Hypervolume
IGD	Inverted Generational Distance
LGPL	GNU Lesser General Public License
LOC	Lines of Code
LLOC	Logical LOC
MAB	Multi-Armed Bandit
MHypEA	Multi-objective Hyper-heuristic Evolutionary Algorithm
MOEA	Multi-Objective Evolutionary Algorithm
MOEA/D	MOEA based on Decomposition
NSGA-II	Nondominated Sorting Genetic Algorithm II
OOP	Object Oriented Programming
PF_{known}	Known/Approximated Pareto Front
PF_{true}	True Pareto Front
RMS	Random Mutant Sampling
ROS	Random Operator Selection
RQ	Research Question
SA	Simulated Annealing
SBSE	Search Based Software Engineering
SBST	Search Based Software Testing
SM	Selective Mutation
SPEA2	Strength Pareto Evolutionary Algorithm 2

SPL	Software Product Line
TS	Tabu Search
UML	Unified Modeling Language

List of Published Work

The following works were published in the past 4 years within the subject of search based software engineering, software testing and hyper-heuristics:

Journal Papers

1. Guizzo, G., Colanzi, T. E., and Vergilio, S. R. (2017b). **Applying design patterns in the search-based optimization of software product line architectures**. Software & Systems Modeling, 1–26. [44];

2. Guizzo, G., Vergilio, S. R., Pozo, A. T. R., and Fritsche, G. M. (2017c). **A Multi-Objective and Evolutionary Hyper-Heuristic Applied to the Integration and Test Order Problem**. Applied Soft Computing, 56:331–344. [50];

3. Guizzo, G. and Vergilio, S. R. (2018). **A pattern-driven solution for designing multi-objective evolutionary algorithms**. Natural Computing, 1–14. [47].

Conference Papers

1. Guizzo, G., Fritsche, G. M., Vergilio, S. R., and Pozo, A. T. R. (2015a). **A Hyper-Heuristic for the Multi-Objective Integration and Test Order Problem**. In Proceedings of the 24th Genetic and Evolutionary Computation Conference (GECCO'15). [45];

2. Guizzo, G., Vergilio, S. R., and Pozo, A. T. R. (2015b). **Evaluating a Multi-Objective Hyper-Heuristic for the Integration and Test Order Problem**. In Proceedings of the 5th Brazilian Conference on Intelligent Systems (BRACIS'15). [48];

3. Guizzo, G., Vergilio, S. R., and Pozo, A. T. R. (2015c). **Uma Solução Baseada em HiperHeurística para Determinar Ordens de Teste na Presença de Restrições de Modularização**. In Proceedings of the VI Workshop de Engenharia de Software Baseada em Busca (WESB'15). [49];

4. Guizzo, G. and Vergilio, S. R. (2016). **Metaheuristic Design Pattern: Visitor for Genetic Operators**. In Proceedings of the 5th Brazilian Conference on Intelligent Systems (BRACIS'16). [46];

5. Lima, J. A. P., Guizzo, G., Vergilio, S. R., Silva, A. P. C., Filho, H. L. J., and Ehrenfried, H. V. (2016). **Evaluating Different Strategies for Reduction of Mutation Testing Costs**. In Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing (SAST'16). [71];

6. Mariani, T., Guizzo, G., Vergilio, S. R., and Pozo, A. T. R. (2016a). **A Grammatical Evolution Hyper-Heuristic for the Integration and Test Order Problem**. In Proceedings of the 25th Genetic and Evolutionary Computation Conference (GECCO'16). [76];

7. Mariani, T., Guizzo, G., Vergilio, S. R., and Pozo, A. T. R. (2016b). **Automatic Design of Algorithms Applied to the Multi-Objective TSP Problem**. In Proceedings of the XIII Encontro Nacional de Inteligência Artificial e Computaciona (ENIAC'16). [77];

8. Guizzo, G., Bazargani, M., Paixao, M., and Drake, J. H. (2017a). **A Hyper-Heuristic for Multi-Objective Integration and Test Ordering in Google Guava**. In Proceedings of the 9th International Symposium on Search Based Software Engineering (SSBSE'17). [43].

Contents

Chapter 1

Introduction

The main objective of software testing is to detect faults in the software under test and to increase the reliability that the software actually implements the required features [20, 87, 96]. A fault is a piece of the program that is not correct, it can be in a line of code, a model element, a model connector, and other artefacts. However, in order to reveal a fault, it is necessary that the tester executes test cases that try to make the program fail. A test case is composed by the input test data and the expected output. If the test case is executed and the obtained output is different from the expected output, then the program failed and probably has a fault.

Ideally, all possible inputs for the program should be executed. However, this is infeasible due to the size of the programs and the number of possible test data combinations [20]. Therefore, the tester must cautiously plan the design of the test cases to ensure that the software is efficaciously tested in a feasible time. Moreover, often the tester needs to act as an oracle by determining the expected output for each test data. These factors increase the testing cost. Actually, the cost of the testing activity is one of the greatest among all activities of the software development process [96]. Some authors estimate that the testing cost is equivalent to approximately half of the total software development cost [53, 87, 96].

In order to check if the software was sufficiently tested using only a subset of test cases from the set of possible ones, the tester can use some testing techniques. These techniques define criteria that must be satisfied by the test cases. The main techniques are: Functional Testing, Structural Testing and Fault-Based Testing [20]. The functional testing does not use information about the internal structure of the software, only information about its functionality. On the other hand, the structural testing has access to the software internal details, such as the code, components connections, control flow graphs and so on. The fault-based testing techniques emulate common faults introduced by developers in the software and try to reveal those faults.

Among the fault-based testing criteria, the most well-known is Mutant Analysis (or Mutation Testing) [1, 9], which is based on the concept of mutant programs. A mutant is a modified version of the program under test containing a small syntactic variation introduced by a mutation operator. The objective is to kill the mutant programs. A mutant is said to be killed when its output is different to the output of the original program with the same test case. If the output of the original program is correct, then the program does not contain the fault introduced in the dead mutant. Thus, the more mutants are killed, the more faults are discarded from the original program and the better is the test suite being executed. The adequacy to this criterion is measured by the mutation score. This score computes the ratio between the number of dead mutants and the number of available non-equivalent mutants. Equivalent mutants are mutants that are not killed by any test case, because they show the exact same behaviour when compared to the original program.

The ideal mutation score is 1.0 (100% of the mutants are killed), however, it is not always possible to kill all mutants with the available test cases. In this situation, the tester must create new test cases to try to kill alive mutants. Depending on the number of available mutants and test cases, this process can be lengthy and costly to the tester, because several program executions must be done. Usually, for each new test case, the original program and all the alive mutants are executed. This results in a great computational cost for the mutation testing activity [20, 62, 85].

One way to avoid this problem and to reduce the testing cost is by reducing the number of mutants. This reduction impacts directly on the execution cost of mutation testing, since fewer mutants means fewer program executions. The problem with mutant reduction is that the efficacy of the test must not be affected in terms of mutation score. In this work, the mutation test efficacy is measured in terms of mutation score, thus maintaining test efficacy herein relates to maintaining the mutation score and potentially the number of revealed faults by the mutants. In this sense, the set of test cases adequate to the subset of mutants (reduced set of mutants) must also be adequate to the set of all available mutants, i.e., the mutation score obtained by a set of test cases executed on the reduced set of mutants must be the same or very close to the mutation score obtained by the same set of test cases executed on all available mutants [62].

In order to reduce the number of mutants and maintain the test efficacy, the tester can use mutant reduction strategies [62]. These strategies select a subset of mutants from the set of all mutants, such that only those selected mutants are executed for the assessment of the mutation criterion. There are several ways of doing that, such as randomly selecting a subset of the generated mutants (Mutant Sampling) [81, 91, 102], applying only a subset of the available mutation operators (Selective Mutation) [21, 80, 84], determining a set of essential mutation operators to avoid generating redundant mutants and mutants that do not contribute to increase the mutation score [6, 82, 83, 100], applying clustering algorithms to group mutants and to select mutants according to those groups [57, 59], and so on. It is important to mention that there are other kinds of cost reduction strategies that do not focus on reducing the number of mutants. These strategies apply different techniques such as parallelism, symbolic execution, bytecode analysis and others to speed up the mutation testing activity. Even though this work focus only on mutants reduction strategies, these other kinds of strategies can be used in a complementary way.

Besides the conventional mutant reduction strategies, there are some strategies based on search algorithms [95]. The research field that proposes the use of search based algorithms to solve software engineering problems is called Search Based Software Engineering (SBSE) [55]. In the context of software testing, the sub-field is known as Search Based Software Testing (SBST) [53].

Some works [41, 42, 71, 105, 106] have done experimental comparisons of the main strategies in the literature. In summary, it is not clear which is the best one as different works find different results depending on the context of the test, program sizes, programming languages and others. Even the same authors have different conclusions [105, 106] as to which strategy to use.

Furthermore, these strategies are usually only concerned with reducing the number of mutants and maintaining the mutation score. However, when reducing the mutation cost, other objectives can be considered by the tester such as reducing the number of equivalent mutants, minimizing the execution time, increasing the number of revealed faults and so on. Another downside is that the selection and configuration of strategies can be manually intensive, but no approach in the literature has proposed the automation of this task. In this sense, we believe that a customizable and automated tool can be beneficial for assisting the tester in choosing which is the best strategy for their context.

1.1 Motivations

Given the great number of conventional and search based strategies available to reduce the number of mutants, a tester can be uncertain on which strategy to use and on how to configure them. Even conventional strategies must be configured/tuned to be effective. For instance, for the Mutant Sampling strategy, the tester must define a percentage of mutants to be sampled from the set of all mutants. If a low percentage is defined, the mutation score can decrease and consequently the test efficacy can be compromised, otherwise, if a high percentage is defined, the cost reduction might not be satisfactory. Another example is the number of mutation operators to be removed in the Selective Mutation. Moreover, the result of the mutation testing activity may depend on the type of the software under test, thus the test efficacy may be affected by the testing scenario. Therefore, because no mutant reduction strategy has been proven to be the best one for all contexts [41, 42, 71, 105, 106], good selection and configuration of strategies is essential for a good mutant reduction. The determination of the best strategy and its best configuration for a given scenario and context usually is only possible through an experimental evaluation. This becomes even more problematic and costly with search based algorithms that demand several parameters. In this sense, the selection and configuration of such parameters and strategies can be seen as an optimization problem by itself [31].

Take the following scenario as an example: the engineers test the software every time they need to commit the code to the repository. During the development of an open-source software for instance, it must be rigorously tested before each commit because there are potentially multiple contributors that depend on shared code, thus mutation testing is used for that end. In a scenario like this, using all mutants in each mutation testing may be infeasible. Of course this cost can be reduced with more powerful machines, parallelism and other techniques, but even then the time saved with mutant reduction strategies can be meaningful on programs with thousands of commits or hundreds of versions (such as Joda-Time[1] and JFreeChart[2]). Hence, an arbitrarily selected and configured strategy can result in unsatisfactory cost reduction and, in the long run, end up not saving as much time as a strategy tailored for the specific testing scenario. It can also be more impacting if the tester has to wait for the mutation testing process finish in order to continue the development of test cases.

The research field of Hyper-Heuristics [12] arises to solve problems like this. Instead of acting over the search space directly, hyper-heuristics are heuristics that are used to select or generate the best heuristics to solve a given problem. Hence, this kind of heuristic operates over the heuristic space. A hyper-heuristic can apply its selection or generation before (off-line) or during (on-line) the problem solving. Off-line hyper-heuristics train the heuristics before and then reuse those trained heuristics to solve unseen instances of the problem, whereas on-line hyper-heuristics are more dynamic by allowing the training “on the go”.

According to Harman et al. [51], in SBSE, many works still focus on the solving of specific problems instead of focusing on the optimization of the software development process. “Holistic” approaches are necessary for the integration and automation of the software development and deployment processes. The authors propose the usage of hyper-heuristics for the creation of approaches that are at the same time holistic and generic, in a way that the software engineer does not need to design an algorithm for each problem faced. Besides that, according to the authors, the engineers must focus only on the development process, and we cannot expect them to also be search algorithm experts. Analogously, in the mutation testing context, the tester must allocate their time on testing activities and not on the selection and configuration of search

[1] https://github.com/JodaOrg/joda-time
[2] https://github.com/jfree/jfreechart

algorithms. This manual task can be replaced by hyper-heuristics that can perform those activities automatically. In this scenario, hyper-heuristics can considerably reduce the development cost, ensuring that the software deployment and usage occur earlier.

However, Harman et al. [51] state that, despite capable of being more generic, holistic, less costly and easier to use, hyper-heuristics based approaches can sacrifice a bit of quality for those benefits. Specific algorithms for a given problem are usually better than more generic algorithms, but the customization process of those algorithms is seen as one of the limitations of the SBSE field [61]. Therefore, the main motivation for the usage of hyper-heuristics in SBSE is that in overall, the easiness of using hyper-heuristics overcome the little quality loss, specially for beginner engineers.

The use of hyper-heuristic in SBSE [8, 13, 34, 35, 43, 45, 48–50, 58, 60, 61, 68, 69, 76, 94, 98] has shown promising results, however, only few works apply them on mutation testing problems [34, 35, 58, 71, 98], which in turn do not tackle explicitly the mutant reduction problem. Moreover, none of those works use hyper-heuristics to automate the generation of mutant testing cost reduction strategies, let alone considering time as an objective to be optimized. We believe that the good results of hyper-heuristics in SBSE can also be extended to the cost reduction of mutation testing, more specifically to the generation of mutant reduction strategies.

1.2 Objectives

This work intends to investigate the advantages of hyper-heuristics in the context of cost reduction of the mutation testing activity. The hypothesis of this work is that an approach based on hyper-heuristics is capable of generating strategies that contribute to reduce the cost of the mutation testing activity without losing efficacy when compared to conventional strategies from the literature. Moreover, we expect that an approach such as this may be capable of automating the manually intensive and tedious activity of selecting and configuring mutant reduction strategies.

To test this hypothesis, we propose Sentinel[3]. Sentinel is an off-line and multi-objective hyper-heuristic approach that automatically generates mutant reduction strategies for the unit mutation testing of Java programs. Sentinel focuses on generating strategies that can minimize the execution time of mutation testing and maximize the global mutation score. As far as we are aware, there is no work in the literature that considers actual execution time as an objective to be optimized when using mutant reduction strategies. Furthermore, Sentinel can be customized to encompass other objectives during the generation, consequently obtaining strategies that are tailored specifically to the needs of the tester.

We designed Sentinel to generate reusable strategies, such that those strategies could be used in newer versions of the software under test without the need of constant training. Hence, the main goal of Sentinel is to automate the selection and configuration processes of strategies and provide reusable strategies. In this sense, a tester would use Sentinel once to generate strategies and then reuse those strategies when they must perform the mutation testing several times during the development of a program.

To evaluate Sentinel, we designed a set of experiments on real-world systems with multiple versions. The results are analysed to answer three Research Questions (RQs). RQ1 is concerned with the capability of the GE algorithm implemented with Sentinel in generating strategies in a better way than a random hyper-heuristic. RQ2 is concerned with the comparison

[3]Name inspired by the Sentinel character from Marvel's X-Men comics (`http://marvel.com/universe/X-Men`). In the Marvel universe, a Sentinel is a villain giant robot in charge of eliminating mutants.

of strategies generated by Sentinel with conventional strategies from the literature. Finally, RQ3 is concerned with the reusability of strategies in newer versions of the software.

1.3 Text Organization

This work is organized as follows:

Chapter 2 – Background: This chapter presents the background for the understanding of this work. In this chapter the main concepts of search based algorithms, hyper-heuristics and software testing are presented. Because the focus of this work is mutation testing, this kind of test is described more comprehensively. Moreover, mutant reduction strategies from the literature are described and discussed in this chapter;

Chapter 3 – Sentinel: This chapter presents Sentinel and contains the main details of the approach, such as the context in which it is applied, its structure and features, the objective functions to evaluate the generated strategies, the solution representation, the available operations and some implementation aspects.;

Chapter 4 – Experiments: This chapter presents the experimental evaluation conducted to asses the feasibility of Sentinel. In this chapter we describe the experimental set-up, report the obtained results, discuss what we observed and then answer the research questions;

Chapter 5 – Related Work: This chapter presents and describes related work. In this chapter we present papers that apply hyper-heuristics in SBSE, including the subfield of SBST;

Chapter 6 – Conclusion: This chapter presents the final remarks and future work.

Chapter 2

Background

This chapter describes some fundamental concepts for the understanding of this work. Evolutionary Algorithms are presented in Section 2.1 along with its subtypes that are relevant to this work: Genetic Programming, Grammatical Evolution and Multi-Objective Evolutionary Algorithms (MOEAs). Section 2.2 introduces hyper-heuristics, how they are classified and their main characteristics. Section 2.3 introduces the concept of Search Based Software Engineering (SBSE). Section 2.4 briefly discusses software testing, its objectives, techniques and criteria, focusing mainly on Mutation Testing. Finally, Section 2.5 highlights some subjects of interest in this thesis.

2.1 Evolutionary Algorithms

Some real world problems are way too hard to be solved, because for these problems it would take too much time to exactly find ideal solutions. Such solutions are too hard to be found given the size of the solution space. Heuristics can be applied to find a solution that is very similar to the optimal solution (ideal solution) with acceptable execution time [39]. Such heuristics use search procedures that are based on approximations to find solutions that are efficacious to the problem. Heuristics can be deterministic (do not use randomness; always yield the same result) or stochastic (use randomness) [15]. Search based algorithms (metaheuristics) can use either kind of heuristics to search for a solution in a solution space that contains all the possible solutions for a given problem.

Among the most used metaheuristics, Evolutionary Algorithms (EAs) stand out as the most used and promising ones [32, 55]. EAs can be described as search based algorithms inspired by the Theory of Evolution. In nature, individuals that are not well fitted to the environment do not survive for too long, getting extinct by natural selection as generations pass. The strongest individuals (the fittest) survive and reproduce several times, more often than the weakest. Hence, the fittest ones pass along their genes more often to new individuals, and those genes are propagated throughout several generations. This makes the subsequent generations to be more adapted to the environment due to their better genes. Besides that, as the individuals reproduce, some mutations occur in the genes of the offspring. These mutations can be either beneficial or disadvantageous for the individual. If a mutation is good, i.e., helps the individual in any way to be fitter than the others, then natural selection will probably propagate the mutated gene to the next generations and consequently will increase the fitness of the species as a whole. On the other hand, if the mutated gene is worthless or actually makes the individual less fit, then natural selection will probably discard such a gene during the next generations. As one can imagine,

these mutations not only enable new individuals to be fitter than their parents, but also insert diversity in the population.

In the EA terminology, a solution is an individual that has a genetic representation for its chromosome. Chromosomes are formed by a set of genes and each gene represents a genetic characteristic of the chromosome. These genes are mapped into phenotypic traits of the solution. This mapping requires a mechanism that works between the search space and the chromosome, converting genotypes (chromosome) into phenotypes (solution). The representation of an individual defines the structure of the chromosome and allows the algorithm to search for new solutions on the genotype level. The representation defines the type of the genes (e.g. integer, binary, real), how they are organized (e.g. permutation array, common array) and other details that are used by the genotype-phenotype mapping procedure to transform a set of genes into an actual solution to the problem. In fact, EAs usually only work on the representation of the problem and not on the problem in its raw form [15]. Moreover, EAs evolve a set of solutions called population, which in turn usually has a fixed size and is defined by a parameter.

Algorithm 2.1 presents the pseudocode of a common EA [32].

Algorithm 2.1: The general scheme of an EA (adapted from [32])

```
1  Input: n – Population size; p_crossover – Crossover probability; p_mutation – Mutation
       probability
2  begin
3      population ← Randomly initialize the population of size n;
4      Evaluate each solution in population;
5      while stop criterion is not met do
6          parents ← Select parents in population to recombine;
7          offspring ← Recombine parents with a probability of p_crossover;
8          Mutate offspring with a probability of p_mutation;
9          Evaluate each solution in offspring;
10         Select the n best individuals in population and offspring to survive to the
             next generation;
11     end
12     return Best solution in population;
13 end
```

The first step of an EA is to initialize the population (Line 3). Usually the algorithm randomly generates n (parameter) solutions to start the evolutionary process. Then these solutions are evaluated by the fitness function and are given each a fitness value based on how good they are for solving the problem in hand (Line 4). This is the first generation of solutions, and after this initial process, the EA starts to generate new solutions for the next generations.

EAs use mainly three operators to generate new solutions: Selection Operator, Crossover Operator and Mutation Operator (not to be confused with mutation operators used in mutation testing). For the crossover (recombination) procedure, existing individuals are recombined to generate new children chromosomes. Therefore, the EA first selects the parents using the Selection Operator (Line 6). The Selection Operator tends to select the best parents to recombine in hope of generating equally good or even better children in terms of fitness. Given a probability rate (parameter $p_{crossover}$), the selected parents are recombined by the Crossover Operator (Line 7) and n new solutions are generated. These new solutions are called children or offspring.

Solutions with the best fitness values commonly survive for the next generations, thus they propagate their genes to several other solutions more often than the others. As the evolution

goes on, the search procedure leans towards those fitter solutions in the population and, eventually, all the solutions converge to a single "shape", which makes the solutions to be very similar and dominated by a single good solution. Perhaps this single good solution is only good when compared to its neighbours in the search space and is not a very good solution when we consider the search space as a whole. In cases where there is a lack of diversity in the population, the EA might never find better solutions. To avoid that, the Mutation Operator is employed on the offspring (Line 8). With a certain probability (parameter $p_{mutation}$), this operator applies small random changes to the chromosome. If the mutation result is good for solving the problem, then the mutated solution will likely obtain a better fitness value, or a worse one otherwise. Typically, the mutation rate is low, because a high mutation rate makes the algorithm too random. Mutation is fundamental in EAs, because it reinserts diversity into the population and can prevent or even revert the situation of an EA falling into a local optimum (searching only a portion of the search space due to convergence to a single solution).

After generating the offspring, the EA then evaluates how good the generated solutions are, i.e., assess the fitness of those solutions (Line 9). Then, the EA must decide which solutions will compose the next generation of n solutions. This is called the Replacement procedure (Line 10), which ensures that the best children survive to the next generation and become parents to pass along their genes to new individuals. The most common replacement strategies replace the parents by their children, optionally using an elitism strategy. The elitism procedure forces a given number of best parents to survive to the next generation, even if those parents are worse than the offspring, implying in the discarding of the worst children. Another possibility is to rank all the individuals according to their fitness and select the n best solutions, regardless of whether they are parents or children.

The EA runs until a stop criterion is met (Line 5). The user can define this stopping condition as a time limit, number of generations, number of fitness evaluations, and so on. When this criterion is met, the EA stops and returns the best solution found so far as the approximate solution for the problem (Line 12).

There are several types of EA [32], the most well-know amongst them are Genetic Algorithms (GAs), Differential Evolution (DE) [97], Genetic Programming (GP) [89], Grammatical Evolution (GE) [66], and others. The most common EA in the literature is GA, because it implements the conventional EA schema. Other kinds of EAs have different types of fitness evaluation, crossover, mutation, solution representation and even different motivations. For instance, GP and GE are used to optimize programs, whereas DE are used to optimize numerical problems.

All of these algorithms can optimize more than one objective simultaneously, thus having several factors that impact on the fitness evaluation of an individual. Such algorithms are called Multi-Objective Evolutionary Algorithms (MOEAs) [15]. Some well-known MOEAs in the literature are: Nondominated Sorting Genetic Algorithm II (NSGA-II) [19], Strength Pareto Evolutionary Algorithm 2 (SPEA2) [108] and MOEA based on Decomposition (MOEA/D) [107].

Next we detail the main concepts of GP, GE and MOEAs, which are used in this work.

2.1.1 Genetic Programming and Grammatical Evolution

A Genetic Programming (GP) algorithm [66] is a type of EA used mainly to evolve other programs. Despite the different motivation, GP algorithms behave similarly to conventional EAs. The most notorious difference is in the chromosome representation. Conventional EAs use arrays of values, whereas for GP algorithms a solution is represented by a tree structure, like the one depicted in Figure 2.1.

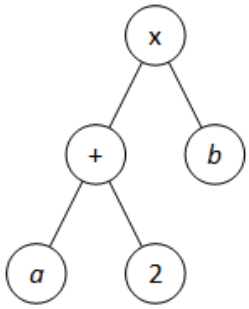


Figure 2.1: Example of a GP representation

In the example of the figure, the tree is used to represent an arithmetic expression. Each tree node is a gene containing a constant, a variable or an arithmetic operator. Each node can be categorized as a terminal node (leaf) or non-terminal node (non-leaf). In the example, the terminal nodes are the constants and variables, because they do not need other nodes to form an expression. On the other hand, the arithmetic operators are the non-terminal nodes, since they cannot form a valid expression by themselves. As seen in Figure 2.1, the non-terminal node + uses two values (a and 2) to actually perform the sum, whereas the $\times$ node use the + and b nodes. Using this tree as an in-order binary tree to print its node values in sequence, we can obtain the expression $a + 2 \times b$.

This kind of structure can be employed to generate several kinds of programs, basically changing the way the nodes are represented and how the tree is mapped into a program. For instance, if the objective is to evolve structured programs, each non-terminal node can be a function to be called and the terminal nodes can be the possible parameters for the function and procedure calls. Moreover, a tree-program mapping can be developed in a way that the program is built directly from the tree interpretation, thus making it easier to execute and evaluate the program.

During the evolutionary process, the GP algorithm recombines and applies mutations directly on the tree structure. The crossover of two trees can be done, for example, by dividing the parent trees into branches and then exchanging the tree parts to build new children. The mutation can change the value of a specific node by another similar gene of the same type (terminal/non-terminal). All of this implies that the engineer must carefully design their operators such that the algorithm can generate syntactically correct trees.

Just like a normal EA, the best individuals survive and reproduce more often. In the end, the best program is returned.

A Grammatical Evolution (GE) algorithm [89] can be considered a type of GP algorithm used to evolve programs. However, differently to a GP algorithm that uses a tree representation, a GE algorithm uses a binary or integer vector just like a conventional EA. The caveat here is that this vector is later mapped into a program using a context-free grammar file. This mapping is called Genotype-Phenotype Mapping (GPM), because it maps a vector (genotype) into an actual program solution (phenotype). Algorithm 2.2 presents the pseudocode of a conventional GE algorithm as proposed by Ryan et al. [89]. This algorithm is similar to Algorithm 2.1 presented in the previous section, but with the inclusion of the GPM procedure and the pruning and duplication operators. The common EA parameters were omitted for simplicity.

The GE evolution is applied to the chromosome (genotype), but only the program (phenotype) can be executed (Lines 5 and 14) and its results evaluated by the fitness function (Lines 6 and 15). Therefore, the use of the GPM procedure (Lines 4 and 13) is necessary to allow this execution and evaluation in each generation of the algorithm. The advantage of using GPM is

Algorithm 2.2: The general scheme of a GE (adapted from [89])

```
1  Input: GF – Grammar File; PO – Pruning Operator; DO – Duplication Operator
2  begin
3      population ← Randomly initialize the population of size n;
4      programs ← Perform the GPM on population using GF;
5      results ← Execute programs;
6      Evaluate results and assign a fitness for their respective solutions in
         population;
7      while stop criterion is not met do
8          parents ← Select parents in population to recombine;
9          offspring ← Recombine parents with a probability of p_crossover;
10         Apply PO on offspring;
11         Apply DO on offspring;
12         Mutate offspring with a probability of p_mutation;
13         programs ← Perform the GPM on offspring using GF;
14         results ← Execute programs;
15         Evaluate results and assign a fitness for their respective solutions in
             offspring;
16         Select the n best individuals in population and offspring to survive to the
             next generation;
17     end
18     return Best program in population;
19 end
```

that the search space can be freely explored without concerning about constraints, because those constraints are included into the grammar file such that all generated solutions are grammatically valid [7]. Hence, this increases the flexibility of crossover and mutation operators and allows different genotypes to generate the same program.

The most common representation used by GE algorithms is a binary array. This binary array is mapped into an integer array during the evolution, thus it is possible to avoid this step by using an integer array alone. Nonetheless, each integer in the array is a gene representing a program characteristic. In order to transform this array into a program, the GE algorithm reads a grammar file provided by the engineer, interprets the grammar rules and then uses the genes to decide which value is assigned to each rule. To illustrate this process, Grammar 2.1 presents an example of grammar used to evolve an arithmetic expression.

⟨*expr*⟩ ::= ⟨*var*⟩ | ⟨*expr*⟩ ⟨*op*⟩ ⟨*expr*⟩

⟨*var*⟩ ::= `x` | `y`

⟨*op*⟩ ::= `*` | `/` | `+` | `-`

Grammar 2.1: Grammar example used to evolve arithmetic expressions.

Items between ⟨ and ⟩ are rules/non-terminal nodes. A non-terminal node positioned on the left side of ::= can accept any option that is positioned on the right side of ::=, regardless of whether an option is composed by one or more nodes. Each option is delimited by the logical

operator "or" represented by |. Any value outside ⟨ and ⟩ is a terminal node, i.e., it is a concrete value and is not considered a rule because it is not assigned any other option. For instance, the rule $\langle op \rangle$ accepts any arithmetic operator listed as options in the form of terminal nodes ($*$, /, + or −). On the other hand, $\langle expr \rangle$ can only accept one out of the following two options: $\langle var \rangle$ or $\langle expr \rangle\langle op \rangle\langle expr \rangle$. When a rule is assigned an option that comprises non-terminal nodes (e.g. $\langle expr \rangle$), then the non-terminal nodes of that option must be evaluated and assigned values according to their respective options. This process continues until there are no more rules to be evaluated, having only terminal nodes in the solution. During these evaluations, the GPM consumes a gene from the array every time it needs to choose between the options and assign such an option to the rule being evaluated.

In order to exemplify this process, consider Grammar 2.1 and the following chromosome: $\{79, 4, 41, 18, 66, 6\}$. Firstly, the GPM selects the root grammar node/rule $\langle expr \rangle$. Because this rule is non-terminal and needs to be assigned one of its options, the GPM consumes the first gene 79. The choice is based on the remainder of dividing the consumed gene 79 by the number of available options for that rule (in this case 2): $79\%2 = 1$, where 1 corresponds to the index of the option that is assigned to the rule, in this case $\langle expr \rangle\langle op \rangle\langle expr \rangle$. The algorithm must continue consuming genes and assigning options, because the selected option is composed by non-terminal nodes. The first node of the chosen option is $\langle expr \rangle$ and the next gene to be consumed is 4. Given that $4\%2 = 0$, then the selected option is $\langle var \rangle$. To evaluate this non-terminal node, the GPM procedure consumes the gene 41, for which the chosen index is 1 (result of $41\%2$). y is the option of index 1 for $\langle var \rangle$, thus for this rule the terminal node y is assigned and the execution flow returns to the previous choice ($\langle expr \rangle\langle op \rangle\langle expr \rangle$), which is still not complete. Using what was evaluated up until now, we have the incomplete arithmetic expression is $y\langle op \rangle\langle expr \rangle$. The next node to be evaluated is $\langle op \rangle$, to which the GPM procedure uses the next gene 18. Because this is a non-terminal node and has 4 options, then the choice is done based on $18\%4 = 2$, which results in the terminal node +. Currently, the incomplete expression is $y + \langle expr \rangle$. Then, the execution flow returns to the previous rule and the next node is evaluated, resulting in x. In the end, the generated expression is $y + x$. Because there are no more rules to be evaluated, then $y + x$ becomes the phenotype of the genotype $\{79, 4, 41, 18, 66, 6\}$. Hence, for this specific grammar, each integer array results into an arithmetic expression.

When the GPM interpretation flow reaches the end of the chromosome but there are still rules to be evaluated, i.e., there are no more genes to be consumed but it needs more to complete the procedure, then a few strategies can be employed. The first and simpler option is to discard the solution. The second option is to perform a wrapping, which consists in starting to consume the genes from the beginning of the array all over again. However, the problem with wrapping is that the genes that are good to a given rule may be bad for another one. On top of that, if there is any recursion in the grammar (as in the example), then wrapping an array may lead to an infinite loop. Avoiding that is rather easy, e.g., setting a maximum number of wrappings, a maximum number of recursions or a maximum depth of recursion, but then other parameters and situations must be considered.

In this work, we use the variable length integer array representation as defined in [89]. This representation allows the inclusion of new genes at the end of the array, which implies that we can reduce the number of wrappings. However, this representation comes with the caveat of allowing the recombination and mutation of unused genes (when the chromosome is bigger than necessary). If that happens, then the crossover and mutation in that case will cause no effect. In order to avoid such a case, GE algorithms usually employ two new operators: Duplication Operator and Pruning Operator. The former selects a portion of the chromosome and then copy it

to the end of the array, whereas the latter simply prunes the chromosome at a given point. These operators are applied with similar probabilities to the mutation operator.

After building a program using GPM, the algorithm executes the built program and its results are evaluated by the fitness function. Just like in a conventional EA, it is expected that the best solutions survive to the next generations and propagate their genes more often. In the end of the GE execution, the best program is returned to the engineer.

2.1.2 Multi-Objective Evolutionary Algorithms

Many real world problems are related to several conflicting factors (objectives). Therefore, optimizing a single objective disregarding the others usually yields unacceptable solutions in respect to those other objectives [15]. In cases like this, there are several solutions to solve those problems: the non-dominated solutions from the Pareto dominance concept. If all objectives $z \in Z$ of a problem are of minimization, a solution x is said to dominate a solution y ($x \prec y$), if:

$$\begin{aligned} &\forall z \in Z : z(x) \leq z(y) \\ &\exists z \in Z : z(x) < z(y) \end{aligned} \tag{2.1}$$

Hence, if a solution x is better or equal to a solution y in all objectives and better in at least one objective, then x dominates y. If x does not dominate y and y does not dominate x, then those solutions are said to be non-dominated and can be seen as equally acceptable solutions for the problem in hand. In such a case, the engineer must choose between both solutions, thus the determination of the best solution is subjective to the engineer. If an objective z is preferable over the others, then the engineer can select the non-dominated solution with the best score on this objective.

For instance, when choosing a car to buy, the buyer must evaluate several factors such as the horsepower, price, comfort, security, and so on. If only the price is important to the buyer, then they can buy the cheapest car. However, if the cost of the car and the horsepower are equally important, then a car with greater horsepower but more expensive is equally good as a car that is cheaper but with lower horsepower. Conversely, a car that is more expensive and with lower horsepower than the others is probably an uninteresting car, because the cost-benefit of such a car is worse than the others. The idea is that the algorithm must be capable of finding the non-dominated solutions, as those represent the best possible solutions for the problem.

The set of all non-dominated solutions for a given problem instance is called Pareto optimal set. A Pareto optimal solution cannot have one of its objectives improved without worsening the others. These solutions compose the true Pareto front (PF_{true}) [15]. Even though the main objective of the multi-objective optimization is to find the Pareto optimal set, usually it is not feasible or even impossible due to the number of possible solutions. Depending on how hard a problem instance is to solve, there can be too many solutions to evaluate in feasible time, or even infinite solutions. In this sense, multi-objective optimization algorithms can only find reasonably good approximations of this set in reasonable time: the PF_{known} fronts [15].

An EA can be adapted to solve multi-objective problems. In order to do that, the EA must be able to generate new approximations of non-dominated solutions using crossover and mutation. Such algorithms are called Multi-Objective Evolutionary Algorithms (MOEAs). Moreover, some MOEAs do not need parameters like objective weights, objective prioritization and objective scaling, which made MOEAs popular among engineers for solving multi-objective problems [15].

The main difference between a mono-objective EA and a MOEA is the fitness function. For the latter, the fitness must consider all the conflicting objectives of the problem. Besides computing those values, the algorithm must also be able to interpret them and decide which solutions must survive to the next generations. However, comparing non-dominated solutions is a tricky task, since those are somewhat "incomparable". Usually the MOEAs compute all the objective values and assign a single fitness value to the solution considering all objectives. This fitness value can be obtained by decomposing the objective values [107] or by using the Pareto dominance concept [108] for example. On top of that, MOEAs should consider the convergence and diversity of solutions regarding several objective values. Given the complexity of those mechanisms and the sensibility of the results when one of them is changed, usually MOEAs differ from each other precisely on the fitness evaluation and replacement strategies [15].

Because MOEAs yield a set of solutions instead of just one like a mono-objective EA, the comparison between algorithms is not straightforward. The most common way of doing that is to compare the resulting approximated fronts with each other using quality indicators [109]. Such indicators are capable of computing a quality value for a front or a comparison value between two fronts. The hypervolume indicator (HV) [109] computes the area/volume of the objective space that is dominated by a given front in relation to a reference point (usually the worst possible point). The greater the hypervolume value, the greater the area of the objective space a front dominates, thus the better the front in terms of dominance. The Inverted Generational Distance (IGD) [109] indicator on the other hand, computes the distance between the true Pareto front and a given front. This is done by summing the distance from each point of the true Pareto front to the nearest point of the front being evaluated. Therefore, the lower the IGD value, the closer the front is to the true Pareto front, and consequently the better it is in terms of convergence.

Some of the most well-known MOEAs can be found in several frameworks of the literature, such as jMetal [30]. jMetal is a multi-objective optimization framework that works with metaheuristics and is developed in Java under the open source licence GNU Lesser General Public License 2.1 (LGPLv2.1). This framework implements several algorithms, problems, representations, operators, quality indicators and other features that are used in the development and execution of MEOAs. jMetal 5.1 is used in this work because it was already used in previous work [45, 48, 49, 76] and has proven to be reliable and flexible enough to develop the proposed approach.

2.2 Hyper-Heuristics

Hyper-Heuristics are commonly defined as "heuristics to choose or generate heuristics". Burke et al. [10–12] describe them as a set of approaches or as a high-level methodology with the main purpose (but not only) of automating the design and tuning of heuristic methods (low-level heuristics) to solve hard computational problems. A low-level heuristic can be seen as a genetic operator, a simple heuristic or even a metaheuristic. The hyper-heuristic field is divided into two main sub-fields: heuristic selection and heuristic generation. Thus, hyper-heuristics can be used to select or generate the most suitable heuristic using existing heuristics. Other authors, such as Chakhlevitch and Cowling [14] define hyper-heuristics as high-level heuristics that: i) manage a set of low-level heuristics; ii) search for a good method to solve a problem instead of searching for a solution directly; and iii) use only a limited portion of the problem information in the search for heuristics.

The main objective of hyper-heuristics is to find a method or sequence of heuristics to be used in a given situation instead of trying to solve the problem directly [10]. A peculiar characteristic of hyper-heuristics that differentiate them from other algorithms is that hyper-

heuristics work on the heuristic space instead of working on the search space [11]. Hence, one of the main ideas is to develop algorithms that are more generally applicable in multiple problems. By achieving this genericity, the algorithm can be reused with less effort and additionally obtain the best results. This idea is motivated by the difficulties regarding the application of conventional search algorithms (e.g. EAs, MOEAs, local search, random search), such as the great number of parameters and the lack of guidelines on how to select the correct low-level heuristics [12].

It is important to emphasize the "limited portion of problem information in the search for solutions" defined in [14]. The idea behind this limitation is to maintain a "domain barrier" that channels and filters specific domain information that are visible to hyper-heuristics. In other words, hyper-heuristics must be independent of the problem domain, having access only to some independent domain information provided by the domain barrier. Examples of independent information are the available low-level heuristics (but not how they work), objective values of the solutions and the direction of the optimization process (maximization or minimization) [12, 14].

In this work we use the classic definition of hyper-heuristics proposed by Burke et al. [12]: a hyper-heuristic is an automatic methodology for the selection or generation of heuristics capable of solving hard search problems. The authors proposed a summarized classification of hyper-heuristics presented in Figure 2.2.

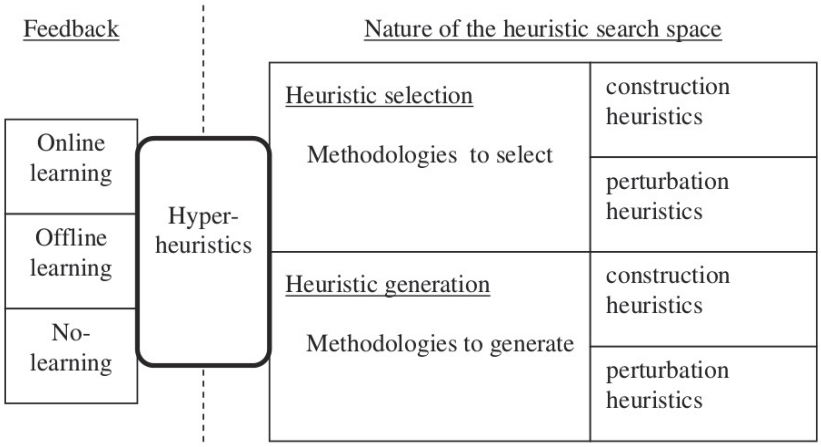


Figure 2.2: Hyper-heuristic classification as depicted in [12]

There are two main hyper-heuristic dimensions [12]: i) the nature of the heuristic search space; and ii) the source of feedback. The nature of the heuristic search space defines if a hyper-heuristic is used to select or generate low-level heuristics. There is a second level of that dimension regarding the type of the low-level heuristic being selected or generated. A construction low-level heuristic starts with an empty solution and gradually builds a solution until a complete solution is obtained. A perturbation low-level heuristic starts with a complete solution (regardless of whether it is randomly generated or built by a construction heuristic) and iteratively modifies it in order to improve it.

The second dimension regards to the source of feedback [12]. A hyper-heuristic may not use a learning mechanism to make a decision and, for instance, randomly select a low-level heuristic. However, a hyper-heuristic can use a mechanism to try to make decisions based on the performance of each low-level heuristic. The "learning" can happen in two ways: i) on-line (dynamic); or ii) off-line (*a priori*). In the off-line training, the hyper-heuristic collects training data from a set of training instances of a given problem, and then uses this knowledge to guide the selection/generation of low-level heuristics in unsolved instances of the same problem. On the other hand, in the on-line training a hyper-heuristic collects this data while the optimization is

happening, without the need of a training phase. This on-line training is potentially more flexible and generic than the off-line training. However, the off-line training is something that usually happens only once aiming at the generation of reusable heuristics in a given domain, and not every time the optimization process happens. Therefore, off-line hyper-heuristics can be less costly in the long run, considering that an on-line hyper-heuristic allocates part of the available resources to the training in each optimization.

2.3 Search Based Software Engineering

Search Based Software Engineering (SBSE) consists in the application of search based techniques for solving hard Software Engineering problems [55]. Such problems are usually optimization problems for which an exact and optimal solution is often impracticable to achieve. SBSE has been successfully applied to several fields such as testing, design, project management, requirements, and others. In fact, works in SBSE have been solving problems since the late 1970s, even though the term was firstly used in 2001 by Harman and Jones [54].

According to Harman et al. [55], there are two key elements that must be taken into consideration when applying SBSE: i) the problem representation; and ii) the fitness function. The problem representation can be adapted to represent the software artefact being optimized, which in turn can be any artefact used during the software development cycle. As for the fitness function, software engineering already has several metrics that can be used as initial candidates to compute properties of artefacts such as cost, size, quality and so on. Because both elements are often virtual, there is no need for emulation of a representation or even a simulation for computing the fitness. This makes software engineering problems well suited to the use of meta-heuristics such as EAs, MOEAs, Hill Climbing (HC), Simulated Annealing (SA) and others.

Several optimization algorithms have been successfully used with SBSE and the most popular ones are [55]: SA, HC, GA and GP. They are most commonly used for solving software testing problems. In fact, according to Harman et al. [55], software testing problems comprise approximately 54% of the problems being solved with SBSE. This might be due to the great cost associated to software testing [53, 87, 96] and the urge in optimizing the activity for reducing costs.

More recently, the field of Dynamic Adaptive SBSE [51] has emerged to investigate the usage of adaptive systems capable to self-optimising. In this context, hyper-heuristics have been successfully used to optimise not only the products produced by software engineers, but also the processes involved in the software development cycle.

The reader is referred to the survey presented by Harman et al. [55] for a complete review of the literature of SBSE. The referred paper contains a mapping of used techniques, problems faced and emerging areas up until 2012. Furthermore, other authors have done systematic reviews, mappings and surveys on the usage of search based for solving problems in specific areas of software engineering [26, 53, 78, 90].

2.4 Software Testing

Software Testing is a Software Engineering activity that aims at revealing faults in the software under test and determine if such a software “apparently” meets the requirements [20, 87, 96]. In this sense, the software testing is a Verification and Validation (V&V) activity [87, 96]. The main objective of the V&V activities is to stablish the reliability that the software is “adequate to

its purpose", i.e., the software must be good enough to be used [96]. Therefore, the software testing is essential during the whole software life cycle.

Software testing is usually performed in different phases with different purposes [96]. Initially, the unit testing is performed over the units of the software by the developers during the software development. This testing phase aims at testing the smallest part of the software, such as procedures, modules and classes. After that, the integration testing aims at detecting faults in the integration between units, i.e., tests if the units are communicating between each other as they should. The next step is the validation testing, which focuses on the validation process of the requirements to ensure that such requirements are actually implemented in the software under test. In the system testing phase, the tester not only evaluates the software, but also the environment in which the software is used to assert that all the elements are compatible. In this phase, the software is tested as a whole, considering other entities such as the hardware, users, databases, subsystems, used web services and so on.

According to Ammann and Offutt [3], a fault is a static defect in the software, such as a line of code containing a wrong instruction, a class designed without one of its methods or any other data incorrectly developed. Faults are inserted in the software due to mistakes committed by the developers and can cause errors. Error is an intermediate and incorrect software state that is the manifestation of a fault. Errors can cause the program to fail, which in turn is an external manifestation of an incorrect state with respect to requirements or other expected behaviours. In contrast to mistakes and faults, errors and failures can only happen if the software is executed. In order to purposely make the software fail and find the faults, the tester executes test cases. A test case is given by the test data and an expected output. Test data refers to possible inputs for the program that can be a sequence of instructions or even just some program parameters. The expected output comprises conditions that must be met by the program when the corresponding test data is informed. Examples of expected outputs are: a specific value printed on the screen, a memory state, the reaching of a control flow branch, and others. If the program generated an unexpected output, then this program probably has a fault. Good test cases can reveal faults that are hard to be discovered. The test case set for a given program is called test suite.

During the test case design, the test data is created and executed afterwards. Then, the obtained results are compared to the expected ones specified by the test case. If any result differs from the expected one, then the tester must debug the software to locate the fault. A program in this context can be any type of executable artefact that receives an input and yields an output [85]. Therefore, a test case can be executed with the source code or even with an executable software model, such as a Petri net, finite-state machine and the Unified Modeling Language (UML) models. Besides that, the program might have been developed using several paradigms, such as Object Oriented Programming (OOP), Aspect Oriented Programing (AOP), structured programming, and so on. All of these details must be considered by the tester during the testing activity, specially during the test cases design.

It is possible that a fault is present in the software, but that it will not be revealed, because no error or failure happens when the faulty program is executed using the available test cases. This demands a meticulously designed testing such that this situation does not occur frequently.

Ideally, all possible test cases should be designed and implemented, thus covering the whole program domain. In other words, the ideal scenario would be to develop a test suite that can thoroughly test the program and not leave out any possible test input. As one can imagine, this is infeasible and virtually impossible [20], given the size of the programs, their complexity and mainly the cost of the testing process [96]. Some authors of the community estimate that the testing cost is equal to approximately half of the total development cost [53, 87, 96]. Therefore,

it is essential to carefully design the test cases such that they are capable of revealing faults, but without making the process too costly. For that purpose, the tester can use some testing techniques.

There are mainly three types of testing techniques [20]: Functional Testing, Structural Testing and Fault-Based Testing. The functional testing works as a black-box, where the software structure is not revealed and the tests are designed based only on its functionality. Conversely, the structural testing uses information about the structure of the software to test it, working like a white-box. The fault-based testing technique uses mistakes often made by developers and common faults that can be present in the program under test.

Each of these techniques provides criteria that the tester must meet so that the tests can be considered adequate. In other words, the tester chooses a criterion that, when met, determines that the software was tested enough. For example, for the structural testing, a possible criterion is that the test data should reach all the execution branches of the program. In the functional testing, a common criterion is the selection or creation of test data in the boundaries of the program domain. Our work focuses on the fault-based testing criteria, more specifically on Mutation Testing. The next subsection describes Mutation Testing and the main strategies used to meet this criterion.

2.4.1 Mutation Testing

Mutation Testing is a fault-based criterion that uses the concept of mutants to reveal faults in the software [1, 9, 62]. It can be used to help the tester in creating good test cases or in evaluating the quality of existing ones.

Given a program P, a set of mutants M is derived. A mutant $m \in M$ is a version of P with a small syntactic variation inserted by a mutation operator. Mutation operators are transformation rules that slightly change the program under test and produce one or more mutants. Figure 2.3 depicts a mutation example in the source code of a program P.

```
1 if(a < b){
2     . . .
3 }
```

(a) Original

```
1 if(a > b){
2     . . .
3 }
```

(b) Mutant

Figure 2.3: Mutation example

In the code block depicted in Figure 2.3(a), the original program P has the conditional expression $a < b$. By applying a mutation operator that changes relational operators (e.g. $<$, $>$, $\leq$, $\geq$, $=$, $\neq$), a mutation was inserted in this instruction and a mutant program m was generated with the instruction $a > b$ replacing the original one as seen in Figure 2.3(b). The only difference between m and P is in this line of code and in this specific instruction. There are other mutation operators that perform different changes, such as removing a line of code, exchanging variables, exchanging constants and so on. There is also another kind of mutation testing called Higher Order Mutation [52]. A Higher Order Mutant (HOM) is a mutant with several faults, such as a combination of multiple conventional mutants (also known as First Order Mutants – FOMs). In this case, multiple mutation operators can be applied on a single mutant and more complex faults can be introduced in the software. In this work we only perform conventional mutation, which is concerned with conventional mutants/FOMs.

Mutation Testing usually applies small changes as shown in the example due to two fundamental hypotheses [62]: i) Competent Programmer Hypothesis (CPH); and ii) Coupling

Effect. CPH states that programmers are competent, which assumes that the developers create programs very close to the correct version and, if there is any fault in the software, such fault shall be very simple that can be fixed by small syntactic changes. The Coupling Effect hypothesis states that complex faults are composed by simpler faults in such a way that test cases that can reveal the simpler ones can also reveal the complex ones.

Before executing the mutants, the tester must provide a test set T to be executed in P, regardless of whether it is being created or if it already exists. If the test cases reveal a fault in this phase, then P must be fixed and new test cases must be created to reveal faults in this new version of the program. This cycle continues until a test set is unable to reveal a fault in P. Actually, in such case, the test set cannot prove that P is correct and is fault free, only that no failure was observed. P can have faults that do not generate error states with this test set, or error states that do not result in failures.

After successfully executing T in P, T is then executed against each mutant $m \in M$ until m dies or until there are no more tests left. If during the execution of m a test case t generates an output mismatching the output generated by itself when executed in P, then m is said to be killed/dead, or alive otherwise. After executing the mutants in M, the tester can compute the mutation score given by Equation 2.2.

$$MS(T, M) = \frac{DM(T, M)}{|M| - EM(M)} \tag{2.2}$$

let $MS(T, M)$ be the mutation score obtained when executing T against M, within the range $[0, 1]$; $DM(T, M)$ the number of mutants in M killed by T; $|M|$ the number of mutants in M; and $EM(M)$ the number of equivalent mutants in M. Briefly, the mutation score computes the percentage of dead non-equivalent mutants.

An equivalent mutant is a mutant that contains a change that always yields the same state yielded by P, regardless of the executed test case, i.e., there is no test case that can kill the equivalent mutant. There is a great chance that a mutation operator will generate an equivalent mutant that the tester is unaware of. So, after generating the mutants, the tester must evaluate them to decide which ones belong to EM set so that the mutation score can be accurately computed. The problem in this scenario is that automatically identifying equivalent mutants is undecidable [20, 62, 85]. Hence, the tester usually determines the equivalence manually and the cost of the software testing activity is increased as a consequence.

The ideal scenario is to always kill all mutants in M with T, obtaining the perfect mutation score of 1.0. A test set T that achieves the 1.0 score is said to have satisfied the mutation adequacy criterion and is called an adequate test set. However, finding such a set is a tricky and costly task. If the mutation score is below 1.0, then new test cases must be created, the original program must be executed with those test cases and the whole cycle described in this section must be redone. Depending on the number of available mutants and test cases, this process can be long and very costly to the tester. In fact, this is one of the main disadvantages of mutation testing and probably the single most important reason of its unpopularity in industry when compared to other software testing techniques [20, 62, 85]. Some cost reduction techniques in the literature offer solutions to this problem [62].

The next subsections present, respectively, some conventional strategies to reduce the cost of mutation testing and tools to aid such a task.

2.4.2 Mutant Reduction Strategies

According to Offutt and Untch [85], the mutation testing cost reduction strategies can be divided into three groups: “do fewer”, “do faster” and “do smarter”.

The “do faster” strategies consist in the faster execution of mutants with different approaches. Some examples of this kind of strategy are the execution of compiled programs instead of interpreted programs, mutant schema, parallelism, and others.

The “do smarter” strategies try to avoid the full execution of mutants. For example, instead of executing a mutant until the program reaches the end like in the conventional mutation process (also called Strong Mutation), Weak Mutation strategies [56] evaluate the state of the mutant right after executing the faulty instruction. Weak mutation is potentially faster to be executed, however, incorrect states not always produce failures, which can lead to misinterpretation of results. On the other hand, strong mutation demands the full execution of a mutant, which implies greater computational cost. There are also hybrid strategies called Firm Mutation [103].

The focus of this work is on “do fewer” strategies, which suggest the reduction in the number of mutants and test cases to be executed, without losing test efficacy. Such strategies are herein called Mutant Reduction Strategies. Furthermore, this work only employs strong mutation, but other kinds of “do faster” and “do smarter” strategies can be used in combination with mutant reduction strategies. For instance, the reduced set of mutants can be executed in parallel (“do faster” strategy) and with weak mutation (“do smarter” strategy). In fact, some tools already perform this combination (as described in Section 2.4.3).

The mutant reduction problem [62] can be then defined as the search for a subset of mutants M' derived from the set of all mutants M, such that $MS(T', M') \approx MS(T', M)$. Hence, when T' is adequate to M' ($MS(T', M') = 1.0$), it should also be close to be adequate to M. In other words, T' has a good mutation score considering all mutants available in M, but only $|M'|$ mutants need to be executed. This reduces the cost of the mutation procedure without compromising the test efficacy in terms of mutation score.

According to Jia and Harman [62], the main three types of mutant reduction strategies are: i) Mutant Sampling [1, 9]; ii) Mutant Clustering [57]; and iii) Selective Mutation [84]. These strategies were already investigated in the literature [62] and the results are promising. Furthermore, there are also strategies based on Search Algorithms to generate and select mutants [33, 95]. The next subsections discuss these strategies in more details.

Mutant Sampling

Mutant Sampling strategies consist in selecting a subset of mutants from a pool of mutants, usually using random selection. For that, the tester defines a maximum number or a percentage of mutants to be selected.

Such kind of strategy has been shown to be effective for the unit test of C programs in the work of Mathur and Wong [81]. The authors were able to reduce the number of mutants by 90% while losing only 0.16 points in the mutation score. In [102], the authors used a similar strategy in the same testing scenario, but instead of selecting a percentage of mutants from the set of all mutants, a percentage of mutants is selected from each available mutation operator. This strategy was equally effective.

Sahinoglu and Spafford [91] proposed a strategy based on the Bayesian Inference for sampling mutants. The strategy selects a random mutant and then decides if more mutants must be selected based on the variable space and on the mutants already selected. The authors evaluated the strategy in the unit testing of C programs, and they noticed a greater mutant reduction when compared to a conventional strategy.

The main disadvantage of Mutant Sampling strategies is that they perform the selection totally at random. In this sense, the mutation testing results can vary widely depending on the reduced set of mutants. Other kinds of strategies can use information about the domain and past tests to guide the mutant reduction and consequently provide more consistent results.

Mutant Clustering

Initially proposed in [57], the mutant clustering strategies use clustering algorithms to select mutants from different clusters. The k-means and agglomerative clustering algorithms were applied to cluster mutants based on the test cases that are able to kill them. The mutant similarity is given by a binary vector containing in each index the value 1 if the mutant is killed by the test case represented by that index or 0 otherwise. After computing the similarity, a mutant distance matrix is created and then k mutants are selected as centroids. Only then the clustering algorithms are executed. The idea is to select a small subset of mutants in each cluster found by the algorithms, because the test cases that kill a mutant likely kill the mutants in the same cluster. The problem with this strategy is that all the test cases must be executed against all mutants before applying the clustering algorithm every time the mutant reduction must be applied, which is a very costly task. Another problem is how to define the k value, since the clustering results are very sensible to this value. This strategy was evaluated in the unit testing of 5 C programs. The results showed a great reduction in the number of mutants and test cases, while maintaining the mutation score close to 1.0.

Ji et al. [59] propose a clustering strategy that uses information about the program domain to perform the mutant clustering. This information is extracted from variables, types of arithmetic operators and other details in the source-code to define the similarity between mutants. The advantage of this strategy is the generation and selection of mutants before executing the test cases. The authors evaluated this strategy in the unit test of Java programs. The results show that the strategy is capable of maintaining the mutation score as high as 0.9 while reducing in 50% the number of mutants.

Selective Mutation

Another way to reduce the number of mutants is to select fewer mutation operators to generate mutants [84]. The motivation behind this strategy is to avoid generating redundant mutants or mutants that are too similar that can be killed by similar test cases [5]. This excessive generation of mutants happens because mutation operators are applied several times in several parts of the software. For example, an operator that increments a variable can be applied with two variations: $++i$ and $i++$. Furthermore, this incrementation operator can be applied on all variables of the software, generating at least two mutants per line of code with variables.

The first efforts were applied on the exclusion of the operators that generate the greatest number of mutants. Mathur [80] omitted the two most costly operators out of the 22 mutation operators available in the Mothra tool [64] used to mutate Fortran 77 programs. These two operators generate between 30% and 40% of all mutants of the evaluated programs. By removing them, it is possible to obtain a mutation score of 0.9999 and a mutant reduction of 24% according to [84]. Offutt et al. [84] extended the work of Mathur [80] by omitting the 4 and 6 most costly operators. For the 4 operators exclusion, the mutant reduction was approximately 40% with a 0.9984 score, whereas for the 6 operator exclusion, the obtained score was 0.8871 with a 60% reduction.

Other approaches consist in finding a set of essential operators which can be considered the best for a given scenario. Barbosa et al. [6] defined some guidelines for the selection of essential operators:

1. Select the operators with the best mutation scores considering the test case adequacy for the mutants generated by them;
2. Consider one operator per operator category;
3. Remove operators included by other operators;
4. Stablish an incremental strategy for the operator evaluation. Operators that are more suitable to meet the test criteria (e.g. all nodes or all branches of functional testing) should be prioritized. Then, if necessary, more operators should be added to the operator set;
5. Consider the mutation operators capable of improving the mutation score obtained by the included operators so far;
6. Consider the operators with greater strength (difference to the mutation scores obtained by the other operators).

With these guidelines, the authors obtained a set of 10 essential operators among the 77 available in the Proteum tool [22] used for the unit test of C programs. This set was able to reduce by 65% the number of mutants while maintaining a mutation score of 0.996.

Vincenzi et al. [100] also used guidelines like these, but for the mutation in the integration testing phase (interface mutation). The results showed that it was possible to reduce by 73% the number of generated mutants while keeping the mutation score as high as 0.998.

Namin et al. [82] proposed a statistical analysis procedure with a linear regression model to find the set of essential operators for Proteum. The authors found a set of 28 operators which generated only 8% of all mutants and obtained a mutation score good enough for the mutation testing according to the R^2 measure.

Delamaro et al. [21] performed an empirical study using only one mutation operator for C programs. The hypothesis is that using only one powerful operator should be enough to perform a good mutation testing. The authors concluded that the mutation operator responsible for removing lines of code is the most effective operator. This operator generated approximately 3.26% of all mutants and obtains a score of approximately 0.92.

The main drawback of Selective Mutation is that the tester must know which operators generate the greatest number of mutants, while also being aware that the best operators found in the literature may not be the best ones for their software. Using guidelines such as the ones proposed by Barbosa et al. [6] would be ideal, but then again, this comes with a manual cost and computational resources spent on executing operators and mutants.

Search Based Mutant Reduction

Search Based Mutant Reduction strategies use some kind of optimization algorithm to try to minimize the number of mutants. These strategies are applied to generate and/or select mutants and operators.

Adamopoulos et al. [2] use a GA based on co-evolution to simultaneously select test cases and mutants. The co-evolution idea is based on living beings that compete between themselves for resources, or that help each other to obtain a better environment adaptation. In this case the test cases are predators that hunt mutants (prey). The test case fitness is computed by the

ability of killing mutants, whereas the mutants fitness is computed by the ability of surviving test cases, both populations evolving in parallel. In the end, the tester obtains a set of test cases good enough to kill mutants and a set of mutants good enough to reveal new faults. Some important details of this approach are: i) the fitness function considers alive mutants as bad mutants, so that the number of equivalent mutants is reduced; and ii) the fitness function considers the size of the mutants set, such that smaller sets are preferable during the evolution. Given these two factors, the testing cost can be reduced, mainly by reducing the number of mutants. The experiments emulated sets of test cases and mutants, and the results showed an overall increase in the mutation score of the selected test cases while also showing a reduction in the total number of test cases and mutants for the mutation testing activity.

The problem with the approach proposed in [2] is that all test cases and mutants must be executed at each evolutionary generation. This can lead to a great cost for the evolution process. In order to avoid such a problem and to propose a more effective approach, Oliveira et al. [18] proposed a similar approach, but only the best sets of mutants and test cases are executed in each generation. This reduces the number of executions during the co-evolutionary process and results in similar mutation scores. Moreover, the authors proposed evolutionary mutation and crossover operators specific to the problem. The work compares the proposed approach to other 5 mutant selection methods and obtains satisfactory results regarding cost, number of executed mutants and number of selected mutants.

Banzi et al. [5] formulated the mutation operator selection problem as a multi-objective problem. Two objectives were used in this work: number of mutants (minimization) and mutation score (maximization). The authors evaluated 3 search based algorithms: an Ant Colony Optimization (ACO) algorithm, a GA and a Tabu Search (TS) algorithm. In the experiments, the authors observed that it is possible to obtain a better test coverage with fewer mutants by using the proposed approach as opposed to using conventional strategies. Another advantage of using such an approach is that, by searching for operators for a specific software, the found operators can yield better results than using a set of mutation operators found by training on multiple programs. However, this can increase the cost of mutation testing, since the multi-objective algorithm must be executed for each new software. In order to reduce this cost, the authors selected a set of 12 essential operators that are most frequent in the Pareto front solutions. This set of operators obtained the best results when compared to conventional strategies.

Domínguez-Jiménez et al. [27–29] proposed an approach called Evolutionary Mutation Testing (EMT) based on evolutionary algorithms to select a subset of mutants that are hard to kill. The authors proposed a fitness function that computes the strength of a mutant, where a strong mutant is killed by very specific test cases, i.e., by test cases that exclusively kill that single mutant. If a set of mutants is killed by the same test case or by similar test cases, then these are weak mutants, because probably they are easily killed by several test cases. In the experiments, EMT was applied and the results showed a reduction of 15% when compared to a random mutant sampling strategy. These results are due, mainly, to the exclusion of redundant mutants and mutants that are killed in the first execution (around 80% [20]). Furthermore, the EMT approach selects all of the strong mutants faster than the random selection approach.

Quyen et al. [88] applied EMT to the mutation testing of Simulink models. Simulink is a software package for modelling, emulating and analysing dynamic system models (not only software models). The experiments were conducted over a single Simulink model. The authors observed that the mutation score could be maintained while using only 90% of all generated mutants.

Delgado-Pérez et al. [23, 24] extended EMT to test object-oriented C++ programs. The GiGan tool was presented for connecting the MuCpp mutation tool to the GAmera evolutionary

framework in order to allow the execution of EMT on C++ programs. GiGan was applied on four open-source programs during the experimentation phase. The results were compared to a Random Mutant Sampling strategy. According to the results backed by statistical tests, GiGan is better in reducing the number of mutants with large statistical difference.

Summary

The results with mutant reduction strategies are indeed promising, but it is still unclear which one is the best as several works in the literature diverge on their results [41, 42, 71, 105, 106].

Gopinath et al. [41] compared mutant sampling strategies that sample a constant number of mutants with strategies that sample a percentage of mutants. Strategies that use a constant number of mutants (as low as 1,000) can obtain an accurate mutation score (approximately 0.93) when compared to the whole set of mutants. Zhang et al. [106] compared strategies that select mutation operators with mutant sampling strategies. The results showed that selecting operators is not superior to sampling random mutants. Gopinath et al. [42] observed similar results with a theoretical and empirical analysis. In this study, the authors also concluded that the reduction limit over random mutant sampling strategies is 13%. Zhang et al. [105] discovered that using 5% random mutant sampling in combination with operator-based selection strategies can greatly reduce the number of mutants while maintaining a great mutation accuracy, being the best choice in most cases. Lima et al. [71] compared Selective Mutation, Random Mutant Sampling, HOM generation strategies and an Evolutionary Algorithm for selecting mutants and test cases in order to reduce the mutation cost of C programs. The authors discovered that Selective Mutation performed better than the other strategies and that the HOM strategies generated more mutants while also reducing the test efficacy (mutation score).

The results of strategies can vary according to the size of the program being tested, programming language, mutation tool and others. Perhaps this is why the aforementioned works achieve such different results. Furthermore, none of them propose comprehensive guidelines on how to select and configure strategies. Even the same authors in different works present different perspectives on which strategy to use (e.g. [106] and [105]).

Regardless of which strategy is the best, we believe that there is still room for improvement. Usually mutant reduction strategies employ simple heuristics to select mutants or operators. We advocate that it is possible to generate better and unforeseen strategies by combining the features from several kinds of strategies, such as observed by Zhang et al. [105] in their experiments. Depending on the program being tested, the budget for developing the software and the test rigorousness required for the system, such strategies can be very valuable.

As far as we are aware, no work comparing mutant reduction strategies actually measures the execution time as an objective for the reduction process. Usually, for measuring the cost reduction, these works compute the number of mutants in the reduced set and assume that the lower the number of mutants, the faster the mutation process. However, some mutants can take more time to execute than others, such as mutants with infinite loops, mutants with memory allocation faults, mutants that die due to time out, and so on. Hence, a single mutant can take longer to execute than a set of mutants. Indeed the objective of mutant reduction strategies is to reduce the number of mutants, but the ultimate goal of such strategies is to provide a faster execution of the mutant set. Therefore, measuring the execution time might be a more accurate approach. In this work, we actually compute the time and use it as comparison basis for the analysis and for the strategy generation.

2.4.3 Mutation Testing Tools

As seen in this section, the mutation testing demands a lot of steps to be followed that might be costly given the number of available mutants and test cases. Hence, the usage of automated tools is essential to aid the tester. This section presents 4 of the most used tools in the literature [62], which are also capable of reducing the cost of mutation testing.

μJava [73] is a mutation tool for Java programs. This is one of the most well-known mutation tools for Java programs in the literature. It uses mutant schemata generation and bytecode translation to perform the mutations, which potentially can speed up the mutation process. It also employs class and method operators for mutating polymorphism, extension, and so on. Moreover, μJava supports the selection of operators during the set up, thus it allows the tester to perform Selective Mutation.

Proteum [22] is a mutation tool for C programs. This tool works through command line or Graphical User Interface (GUI). Using the command line, the tester can run scripts and easily import and export mutants using the terminal, potentially increasing the level of automation. With the ability of using GUI, the tester can generate comprehensive reports on the test results, input test cases, visualize mutants source code, determine mutant equivalence and so on. Another distinguishable feature is the interface mutation testing support, where faults are inserted in the communication between system units. Proteum also offers the option to select operators, enabling selective mutation. Furthermore, the tool supports mutant sampling by providing a GUI to inform a percentage of mutants to be generated by each operator.

The Bacterio [79] tool is particularly interesting because, not only it performs the conventional mutation activity, but also enables the reduction of mutants using conventional and search based reduction strategies. The available strategies are: mutant sampling and selective mutation. This tool also supports parallel execution, bytecode mutation and mutant schemata, which can reduce the mutation cost during the mutant generation and execution. Lastly, the tester can select a "do smarter" technique, like weak mutation, strong mutation, flexible weak mutation (balanced mutation between strong and weak) and functional qualification (more rigorous version of strong mutation).

In this work we use PIT [16], a mutation tool for Java programs. Since Java is an OOP language, the PIT operators insert faults not only in the source code of the classes (e.g. method body and static code), but also in the structure and signature of classes and methods. This implies in mutations that modify polymorphism, extension, method overload and other OOP features in which programmers usually commit mistakes. Moreover, PIT generates mutants using the bytecode of the program rather than the source code. The benefit of working directly with bytecode is that there is no need in recompiling the mutant programs, which speeds up the mutation process. PIT also supports the execution of JUnit[1] tests, a widely used unit testing framework for Java programs.

PIT can be used mainly in two ways: by adding the dependency and configuration files to the project, and via command line. In both ways the tester can select the operators to be used, activate or deactivate test case prioritization, configure parallelism, and customize other attributes of the tool. By default, PIT uses 7 operators which are described next:

1. **Conditionals Boundary Mutator –** Replaces relational operators with another one;
2. **Negate Conditionals Mutator –** Negates conditional and relational operators;
3. **Math Mutator –** Replaces binary arithmetic operations with another one;

[1] `http://junit.org/`

4. **Increments Mutator –** Mutates increments, decrements and assignments of local variables with such operations;
5. **Invert Negatives Mutator –** Inverts the negative of integer or floating point numbers;
6. **Return Values Mutator –** Mutates the return values of method calls, depending on the type of the return;
7. **Void Method Call Mutator –** Replaces method calls with void method calls.

PIT already employs several cost reduction techniques, such as bytecode manipulation, multi-threading for mutants execution, essential operators, test case prioritization, code coverage analysis, and many more. According to recent work [65, 70], PIT is the fastest mutation tool for Java programs. However, we believe that this is not enough, as big systems can take several minutes to have their mutation testing activity executed. By using mutant reduction strategies in combination with PIT, we believe that we can reduce even more the cost of this expensive task by complementing the existing PIT strategies with mutant reduction ones.

2.5 Final Remarks

This chapter presented the main concepts of the three related fields for the development of this work: Evolutionary Algorithms, Hyper-Heuristics, SBSE and Software Testing.

As seen in this chapter, the software testing activity is very important for the software development cycle, but it is at the same time the single most expensive activity in this cycle. Some techniques and criteria are employed to improve the test quality, mutation testing is one of them. However, usually mutation testing demands a great amount of computational and human resources. For trying to reduce the mutation cost, some mutant reduction strategies can be used.

Since the mutant reduction is a hard problem to be solved, mutant reduction strategies can be employed in this context. However, no strategy has been proven to be the best one in all scenarios [41, 42, 71, 105, 106]. Selecting and configuring the best strategy for a specific scenario can be of utter importance, given the great impact that a strategy can have on the testing results. This can be a costly task, since the comparison of strategies demands their execution several times (given their stochastic nature) and manual analysis of their performance. To the best of our knowledge, there is no work in the literature that presents an approach capable of automating the selection and configuration of strategies, let alone allowing the customization of this task towards certain objectives.

As seen in previous works in SBSE [53, 55], search algorithms such as EAs, MOEAs and hyper-heuristics can be viable options for solving software engineering problems. In the context of mutant reduction strategies, their robustness can allow the automatic selection or generation of conventional strategies and new strategies unforeseen by the tester. Furthermore, by automating this process, hyper-heuristics can remove the boring and manually intense activity of selecting and configuring strategies [32], while also allowing the selection of the most effective one.

Moreover, hyper-heuristics can be flexible enough to allow the tester to customize the generation process towards what they want in a strategy. For example, if the tester wants a strategy that can achieve high mutation score and low number of equivalent mutants, then they can explicitly set those as objectives, and the generation will be tailored to such objectives. This is not always applicable to conventional strategies that are usually only concerned to the number of mutants and mutation score.

With that in mind, this work has as main objective the investigation of a multi-objective hyper-heuristic in the mutant reduction problem. For this end, the proposed approach merges several mutant reduction strategies features by generating new strategies with mixed functionalities. GE is used as hyper-heuristics in this context to aid the strategy generation, considering that such strategies can be seen as low-level heuristics. The next chapter presents the proposed approach in detail.

Chapter 3

Sentinel

This chapter presents Sentinel, a hyper-heuristic for the automatic generation of strategies for the reduction of mutants during the mutation testing activity of Java programs. The idea is that the generated strategies may be able to reduce the cost of the activity while maintaining the test efficacy, and also may be reused afterwards in newer versions of the software.

The main motivation for proposing Sentinel is the automation of the strategy selection and configuration processes, which by themselves are already very costly. The term "strategy configuration" can be understood in this work as the strategy parameter tuning, such as the percentage of mutants in mutant sampling or the number of operators in selective mutation strategies. Because strategy results are sensible to those parameters, a good strategy configuration is essential. According to Eiben et al. [31], the automatic heuristic selection and configuration (correspondingly mutant reduction strategy selection and configuration) is an optimization problem itself. Sentinel uses hyper-heuristics to automatically generate strategies that are at the same time effective and reusable, removing from the hands of the tester such an error prone, tedious and costly activity. Furthermore, according to Harman et al. [51], hyper-heuristic based approaches can also contribute to achieve a more holistic and generic SBSE, such that the multiple software development and deploy phases can be easily integrated.

The strategies generated by Sentinel are used before the mutants execution in the two scenarios in which the mutation analysis is usually employed: i) evaluation of existing test cases; or ii) creation of new test cases. Sentinel can be used on both scenarios for generating a strategy that can efficaciously maintain the mutation score while also reducing the cost of the execution of mutants.

For the evaluation of test cases in a test set T, a program under test is submitted to mutation. A mutant reduction strategy e among the set of non-dominated strategies E generated by Sentinel is then applied and a reduced set of mutants M' is obtained from the whole set of mutants M. Hence, the test set T is executed only in M' until the mutants in M' are killed or there are no more tests to be executed, potentially resulting in fewer mutants executions than using the complete mutant set M. It is expected that $MS(T, M')$ is equal or very close to $MS(T, M)$. Similarly, to guide the creation of test cases, only M' is used. If the tester is able to create a test set T that is M'-adequate, then it is expected that T is also M-adequate or very close to achieve this adequacy.

If a strategy e can find the reduced set M' such that $MS(T', M) \approx MS(T, M)$, where T' is the subset of test cases executed to kill M', then the strategy e is capable of reducing the number of mutants executions, while also being able to accurately evaluate an existing test set or to guide the generation of a test set as if the whole set of mutants was being used. Therefore,

when reducing the number of mutants, a good strategy e is crucial for the effectiveness of the mutation testing activity.

Sentinel uses an off-line hyper-heuristic based on GE, which means that it needs a training phase. In this phase, the GE algorithm uses an instance of the problem to evolve a strategy population and then provides a set containing the best strategies found during the evolutionary process. In this case, before evaluating or creating a new test set to kill a set of mutants obtained by a strategy, the tester must execute the Sentinel training to obtain such a strategy.

In order to evaluate the efficacy of the strategies during the training, first all the mutants M must be executed against all test cases T. The resulting killing matrix containing information about which test cases kill which mutants is then used during the training for assessing if an M'-adequate test set is also M-adequate. This implies that the tester must allocate computational resources at least once for executing all test cases against all mutants. Furthermore, given the stochastic nature of the strategies, they are executed n times to provide an average cost and score. Hence, as it happens for most off-line hyper-heuristics [10, 76], the training phase is costly in terms of computational resources.

On the other hand, after trained, Sentinel obtains as result a set of strategies that can be reused without the whole execution of mutants and test cases every time the mutation testing is needed. These strategies can potentially be reused in newer versions of the software, without the need of constant retraining. This is the main advantage of off-line hyper-heuristics (as opposed to on-line hyper-heuristics): the heuristics can be reused in unforeseen instances of the problem, without demanding computational resources to perform more trainings.

We believe that the best use of Sentinel is when the testers perform the mutation testing multiple times during the software development. For example, in open-source software, the rigorousness of the test must be asserted due to the sharing of code from multiple contributors. Take the Apache Commons Lang repository[1] as an example: it has over 5,300 commits and over 80 releases with several contributors. If mutation testing is used to assess the quality of the test cases of such software, the tester might spend a lot of computational resources for actually executing the mutants. Using mutant reduction strategies can greatly decrease the execution time of this task, specially if the mutation testing must be performed on each repository commit or, the worst case scenario, every time a test case is created and must be evaluated. In cases like this, the better the mutant reduction, the more impacting the cost reduction in the long run. By using Sentinel to generate a strategy and then reusing it, not only the tester is going to potentially find the best strategy for their software, but also they will do it automatically. Depending on how often the mutation testing and mutation reduction are executed, the computational cost for training Sentinel will be amortised by the time saved by the generated strategies.

This chapter is organized as follows. Section 3.1 presents the main structure of Sentinel and how its components work. Section 3.2 presents the objective functions used to guide the evolutionary process. Section 3.3 depicts the solution representation used by the GE algorithm. Section 3.4 describes the operations extracted from conventional strategies that are combined to build new strategies. Section 3.5 presents the grammar used by the GE algorithm to perform the GPM procedure. Section 3.6 describes some of the implementation aspects of Sentinel. Finally, Section 3.7 concludes this chapter.

[1] `https://github.com/apache/commons-lang`

3.1 Structure and Functionality

Sentinel uses a hyper-heuristic based on GE to generate, execute and then evaluate strategies for the unit mutation testing of Java programs. The hyper-heuristic implements features/operations from several existing conventional mutant reduction strategies. Instead of generating strategies that are completely different, the idea is to use common operations performed by conventional strategies for building new strategies. These operations are usually context-free and can be applied to any kind of software, not needing information about lines of code, type of statement being mutated, and so on. Examples of operations are selecting a given number or percentage of mutants, grouping mutants by operator, sorting operators by number of generated mutants and executing a given set of operators. The available operations are presented and described in Section 3.4. Figure 3.1 depicts the general structure of Sentinel.

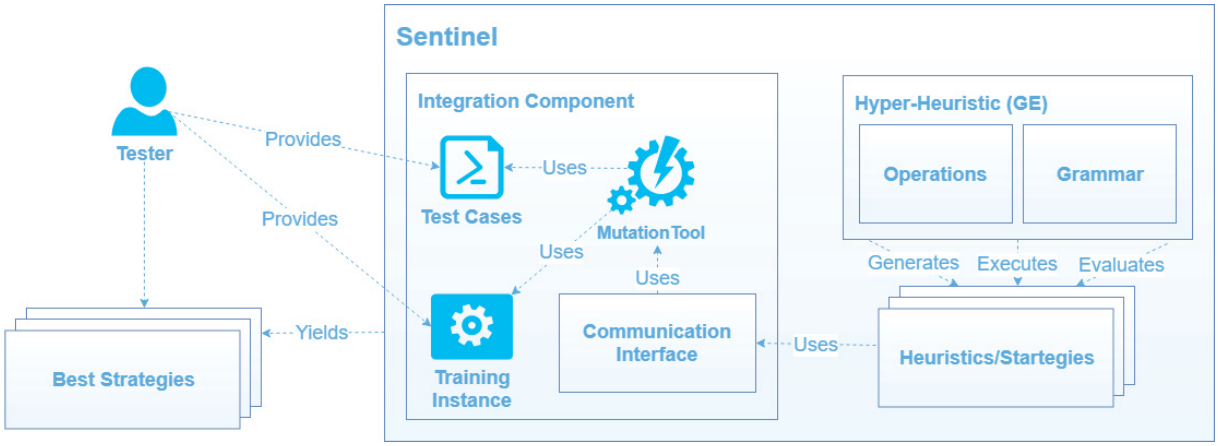


Figure 3.1: Structure of Sentinel

The hyper-heuristic executes each generated strategy on the training instance to generate a reduced set of mutants. The training instance is provided by the tester, alongside its test cases. Each generated strategy during the generation process is evaluated by objective functions that compute the execution cost and mutation score of the obtained reduced set of mutants. In the end of the execution, Sentinel returns the best strategies found during the evolutionary process. These strategies are the ones that presented the best trade-off between cost and mutation score, and that can be reused to generate reduced set of mutants in newer versions of the software under test.

For actually performing the mutation, Sentinel uses a mutation tool that is integrated with it. We implemented a communication interface that receives the commands from the strategies and redirects them to the tool. The tool is decoupled from Sentinel, thus different tools can be used by implementing their corresponding communication interfaces. For instance, Sentinel is implemented in Java and uses PIT for the mutation testing of units of Java programs, but by developing a specific integration component, Sentinel can be adapted to work with Proteum for the mutation testing of C programs. The implemented operations, generated strategies and grammar file are all independent of the mutation context and allow the instantiation of Sentinel in different scenarios by only requiring the implementation of a new integration component.

3.2 Objective Functions

During the evolutionary process, the hyper-heuristic used by Sentinel evaluates the strategies according to two objective functions: i) `TIME` – average CPU time taken by the startegy to be

executed plus the execution time of the selected mutants; and ii) SCORE – average mutation score obtained by T' when applied to M. These two functions are used to reduce the overall time spent performing the mutation testing, while also maintaining a high mutation score. However, as in most multi-objective problems, these two objectives are conflicting. In other words, as we decrease the time spent executing various mutants, we expect to also decrease the overall mutation score. By applying a multi-objective optimization, we expect to find strategies that present acceptable trade-off between those two objectives.

TIME – Average Strategy CPU Execution Time

The TIME function is straightforwardly used to evaluate the average CPU time taken by a given strategy e to execute the mutation testing n times. This function is given by Equation 3.1. Let $cpuTime(e_i)$ be the CPU execution time taken by e in execution i and $cpuTime(M_i)$ be the CPU execution time taken to execute the whole set of mutants M on execution i.

$$\downarrow \texttt{TIME}(e) = \frac{\sum_{i=1}^{n} cpuTime(e_i)}{\sum_{i=1}^{n} cpuTime(M_i)} \tag{3.1}$$

It is important to note that *cpuTime* computes the CPU time taken starting from the strategy execution, until all the selected mutants are executed. Hence, the time function computes not only the time spent selecting operators and mutants, but also the time spent executing operators and time spent executing mutants. Therefore, the lower the TIME value, the faster the mutation testing activity will be performed using the mutants obtained by the strategy, making this a minimization function.

Even though the main objective of mutant reduction strategies is to reduce the number of mutants, we believe that the CPU execution time is a better metric to accurately measure the cost reduction. Some mutants can take more time to be executed than others (e.g. mutants with infinite loops, mutants with bigger memory allocation and mutants that are killed by time out). In addition to that, by measuring the execution time of the strategy, we can also take into account the time taken by the strategy itself to perform the selection and avoid the generation of strategies that perform useless operations during the selection.

For this objective function, the strategy execution time is divided by the time taken to execute all the mutants (as in the conventional mutation process). Therefore, TIME measures how well a strategy can reduce the mutation cost in relation to no reduction at all.

SCORE – Average Mutation Score of T' in relation to M

The SCORE objective relates to the quality of the mutant set obtained by the strategy in terms of mutation score. This objective computes the average mutation score obtained by T' when applied to M in n executions. Equation 3.2 presents the function. Let T be the set of all available test cases, T'_i be the set of test cases used to kill M' in the i-th strategy execution, n be the number of strategy executions and $MS(T'_i, M)$ be the mutation score obtained by T'_i when executed on M.

$$\uparrow \texttt{SCORE}(e) = \frac{\sum_{i=1}^{n} MS(T'_i, M)}{MS(T, M) \times n} \tag{3.2}$$

By maximizing this function, we may find strategies that can obtain a reduced set of mutants that demand a test set good enough to also kill the whole set of mutants. In this sense, the greater the SCORE value, the better the mutant set is in revealing faults. From now on, the

term “mutant set score” refers to the global mutation score obtained by a test set used to kill that specific mutant set.

Ultimately, what mutant reduction strategies try to do is to minimize the cost and maintain the global mutation score [62]. The whole premise of mutant reduction strategies is that by reducing the number of mutants, the cost of mutation testing is reduced as a consequence. Furthermore, such strategies assume that cost and score are two conflicting objectives, since the fewer mutants, the less probable it is to maintain the mutation score. Works in the literature use an objective function similar to `SCORE` to compute the global mutation score and a function that computes the number of mutants to assess the cost [41, 42, 70, 71, 105]. `TIME` is an alternative to the number of mutants with the advantage of measuring the actual cost of execution, but with the disadvantage of being more costly to compute. Both `SCORE` and `TIME` functions are conflicting in the sense that while one is optimised, we expect the other to be worsened, as it happens to number of mutants and mutation score. However, one should bear in mind that it is very hard to optimise both objectives simultaneously (to a certain degree), not impossible. Hence, search algorithms such as the one implemented in Sentinel can properly guide the search towards strategies that are able to obtain good trade-off in both objectives.

Because Sentinel uses multiple objectives, at the end of its execution it will return a set of non-dominated strategies. The `TIME` and `SCORE` values can be used by the tester to aid their choice in which non-dominated strategy to use. For example, if the tester considers a test set good enough when it achieves 0.95 of mutation score, then they can select any strategy that obtained $\texttt{SCORE} \geq 0.95$, preferably the fastest one. In this case, the mutants obtained by the selected strategy will be good enough for the tester to create new test cases or to evaluate existing ones, even though they do not achieve 1.0 mutation score.

Furthermore, this choice can be very important and the tester needs can vary according to the program being tested. For example, for systems which the testing must be as rigorous as it possible can (e.g. flying control systems, medical software) a high mutation score cannot be compromised, whereas the tester may want to sacrifice mutation score for a considerable cost reduction for systems with a small testing budget.

It is also important to note that these objectives can be customized by the user. The tester can design objectives that aim at generating strategies that focus on objectives such as reducing the number of equivalent mutants, reducing the number of executed test cases, increasing the number of bugs revealed and so on. In this sense, the strategies can be specifically adjusted to the mutation testing context.

3.3 Solution Representation

The individual representation used by Sentinel is an integer array, which is recombined and mutated by the GE. This is the same representation used by other works in the literature [76, 89] (already discussed in Section 2.1.1) and how it is interpreted depends on the grammar used by the algorithm (presented in Section 3.5). However, there is a specific representation used by the strategies to manipulate the sets of operators and mutants by selecting, grouping, removing and applying other kinds of operations over these elements to actually enable the mutant reduction.

Each strategy can operate over the operator or mutant level. For instance, the selective mutation operates over the operator level by removing or adding operators in the operator pool. On the other hand, selective mutation operates over the mutant level, by selecting mutants in the mutant pool. Sentinel strategies can operate on both at the same time, thus the solution representation must comprise both mutants and operators alike. In this sense, each strategy

manipulates a solution composed by an array of mutants (V_m) and an array of operators (V_o). Figure 3.2 depicts a solution example.

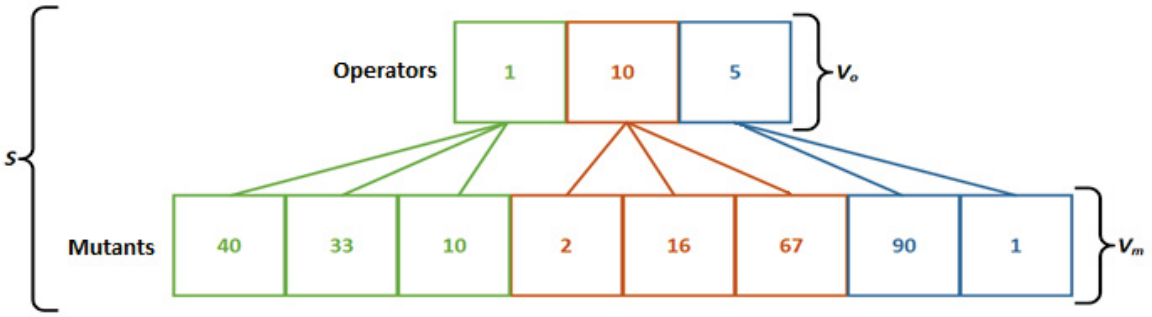


Figure 3.2: Solution representation

A solution S is composed by both V_m and V_o. The value in each array index represents the ID of a mutant or operator. Each mutant is connected to the operator it is generated and each operator is connected to the mutants it generated. These connections enable the grouping of mutants by their operators, and the grouping of operators by the number of mutants they generate. Therefore, the mutants and operators are manipulated as if they share attributes.

This representation was chosen because many works treat operators and their mutants as similar elements. For example, in [5, 6, 21, 22, 80, 82–84] the authors refer to the mutation score obtained by operators. Even though the mutation score is computed for a set of test cases and a set of mutants, operators have "indirect" score in relation to the global mutation score. For computing this score, the function takes into account only the test cases used to kill the set of mutants generated by the given operator.

Even though a strategy is represented by an integer array, this is only its genotype. As seen in Section 2.1.1, a GE algorithm uses a grammar to perform the genotype-phenotype mapping (GPM). Figure 3.3 depicts an example of a strategy phenotype after GPM. The nodes represent operations, rectangles represent solutions S used by the strategy with the given operation, edges represent transitions between operations and edge bifurcations represent the creation of a new tree path.

The strategy phenotype can be seen as a top-down execution tree. In each tree path, a solution S containing the V_o operators and V_m mutants is manipulated independently. Each tree node consists in an operation applied to that solution S. For example, the node C of the figure manipulates S, which in turn was previously manipulated by A and B. The solution can be copied into a new solution when a bifurcation is reached, and that copy is manipulated independently in this new tree path. This can be seen in the C node in the figure, where S_1 is copied to node I and the copy becomes S_3. However, solution S_1 continues to be manipulated by the strategy until the leaf node F. In this sense, solution S_1 is manipulated throughout the $\{A, B, C, D, E, F\}$ path, whereas S_3 is manipulated only throughout $\{A, B, C, I, J\}$.

The first node of a strategy is always the selection of all available mutation operators. In this sense, a solution S always starts with an empty V_m and all the available operators in V_o. From this node forward, the strategy will select, discard, group and perform other kinds of operation on these elements.

At the end of each tree path, in the leaf nodes, the strategy stores the mutants in V_m into an external array called "final mutant set". For example, in the example of the figure, the nodes

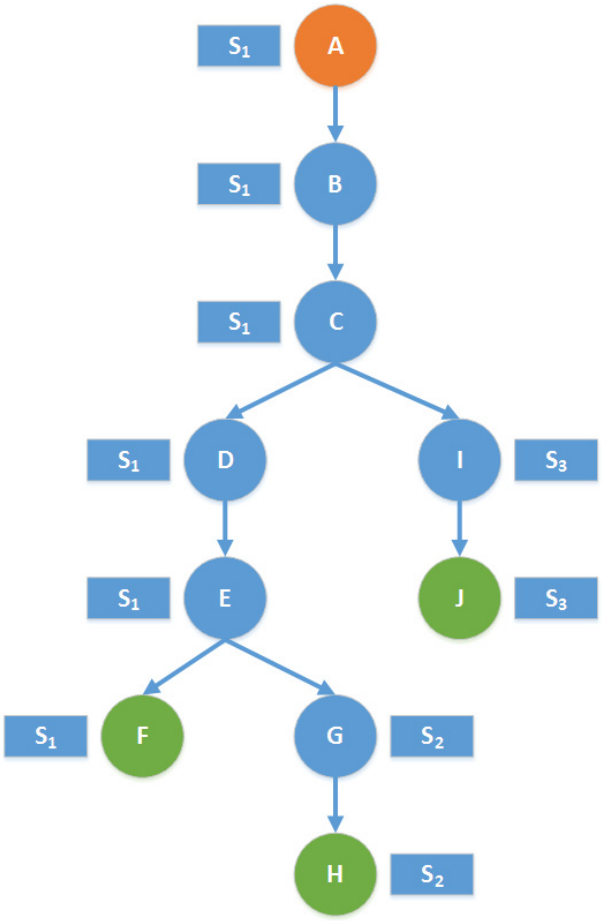


Figure 3.3: Strategy phenotype

F, H and J are leaf nodes that store the mutants in their respective solutions into this final mutant set. After executing all the tree paths, the strategy then returns the final mutant set as the reduced set of mutants M', which in turn is executed to actually perform the mutation.

3.4 Operations

The operations that compose strategies are divided into four main types: operators operations, mutants operations, bifurcation and storing mutants. Operators operations act over V_o of a solution S, whereas mutants operations act over V_m. Bifurcation operations create a new execution path and copy the current solution to this new path. Storing Mutants operations, as the name suggests, store the mutants contained in V_m into the final mutant set M'. This is the path execution stop for the strategy. If such an operation is executed, the strategy will execute another path (if there is any bifurcation).

Furthermore, operator and mutant operations are subdivided into four main subtypes: retention, discard, grouping and execution.

Retention operations select elements and keep only the selected ones in the V_o and V_m arrays. This retention can be done on single elements or whole groups of elements. Furthermore, a sorting operation can rearrange the elements before the retention. On the other hand, discard operations select elements or group of elements and discard only the selected ones. Sorting can also be used before selecting elements for discarding.

Grouping operations are used to group elements in the arrays according to their attributes. This grouping can be done in order to select whole groups of mutants and operators, or only to cluster a set of elements in a certain group to perform operations such as selecting a specific number of elements from each group. Either way, it is possible to group mutants and operators using attributes from the elements of the other group, e.g., grouping mutants by type of operator, grouping mutants by specific operators, grouping operators by number of mutants and so on.

The execution operation is specific to operators. This operation basically selects a set of operators using selection and grouping options, and then executes the selected operators to generate mutants. The generated mutants are stored in the V_m array if they are not present there. Moreover, when an operator is executed, it generates the mutants using the mutation tool according to its rules and the mutants are stored internally in Sentinel for an eventual retrieval. Hence, an operator is never executed more than once during the strategy execution.

For retaining, discarding and executing, the operations first select some elements. This selection can be done in three ways: sequential, random or last to first. All of these selections can be done over a fixed number of elements or a percentage. Sequential selection selects elements starting from the first one and follows the array index. Random selection, as the name states, selects elements randomly. With the last to first selection, elements are selected starting from the beginning of the array and interchanging from the ending of the array: first element, last element, second element, penultimate element, third element and so on.

As mentioned, sorting (ascending or descending) of arrays is also possible before selecting elements. Operators can be sorted by their types or by the number of generated mutants. Mutants can be sorted by their operators or operator types. Sorting is specially useful for sequential or last to first selections.

Another kind of selection can be done on whole groups of elements. For that purpose, the grouping operations can be applied on operators according to their type or number of generated mutants. The mutants grouping is done based on their operators or operator types. Similarly, operator groups can be sorted by their size or by the number of mutants in the group, whereas mutant groups can be sorted by their size.

The aforementioned operations were extracted from three common strategies found in the literature: Selective Mutation, Random Mutant Sampling and Mutant Clustering. For example, Selective Mutation firstly sorts the operators by the number of generated mutants in descending order, and then discards a given number of operators and their mutants selecting them with a sequential selection operation. In another example, Random Mutant Sampling executes all operators and then retains a given percentage of randomly selected mutants. Mutant Clustering groups mutants in clusters and then selects a given number of mutants from each cluster. By deriving those operations and allowing their combination, Sentinel can generate hybrid strategies unimagined by engineers.

Even though Sentinel is applied to the unit test of source-code of Java programs, these operations can work with other contexts of testing, such as integration testing, model testing, testing of programs in different languages and others. In fact, these operations are generic because the strategies from which they were extracted are also generic. In other words, the operations that compose the strategies do not take into account context specific attributes such as line number in which the mutation was applied (source-code testing), if the operators are applied on method or class level (OOP testing), size of the mutated unit (unit testing), and so on. All of these operations are only concerned with context-free attributes, e.g., grouping operators by number of generated mutants, selecting from the array of mutants a given number or percentage of mutants, grouping operators by their category. We have not tested the effectiveness of the strategies in different contexts, but they still can be instantiated and used if Sentinel is integrated to these new contexts.

3.5 Grammar

The operations presented in the previous section can be encapsulated into grammar rules. Moreover, using a context-free grammar enables an easy extension of Sentinel by adding new rules in the grammar and implementing the added operations. Grammar 3.1 presents the first part out of three of the grammar used by Sentinel.

⟨*strategy*⟩ ::= `All Operators` ⟨*defaultOperation*⟩

⟨*defaultOperation*⟩ ::= ⟨*operatorOperation*⟩ ⟨*defaultOperation*⟩
| ⟨*mutantOperation*⟩ ⟨*defaultOperation*⟩
| `Bifurcation` ⟨*defaultOperation*⟩ ⟨*defaultOperation*⟩
| `Storing Mutants`

⟨*selectionType*⟩ ::= `Sequential|Random|LastToFirst`

⟨*sortingDirection*⟩ ::= `Ascending|Descending`

⟨*quantity*⟩ ::= `1|2|3|4|5|6|7|8|9|10`

⟨*percentage*⟩ ::= `0.1|0.2|0.3|0.4|0.5|0.6|0.7|0.8|0.9|1.0`

Grammar 3.1: Sentinel grammar (part 1). Standard operations.

The initial rule ⟨*strategy*⟩ defines the start of the strategy. As mentioned before, all strategies start with all operators in V_o and then an operation ⟨*defaultOperation*⟩ is executed. The ⟨*defaultOperation*⟩ rule represents one of the main 4 types of operations: ⟨*operatorOperation*⟩, ⟨*mutantOperation*⟩, Bifurcation and Storing Mutants. For the former three, the subsequent inclusion of the rule ⟨*defaultOperation*⟩ represents the next operation to be executed in the execution path.

The ⟨*selectionType*⟩ rule defines the three types of selection: sequential, random and last to first. The ⟨*quantity*⟩ and ⟨*percentage*⟩ rules define, respectively, constant numbers or percentages for selecting elements in steps of 1 (constant) and 10%. ⟨*sortingDirection*⟩ defines the two possible directions for sorting elements: ascending and descending. These rules are used as auxiliary rules for several others.

Grammar 3.2 presents the second part of the grammar, regarding the operator operations represented by the rule ⟨*operatorOperation*⟩. λ represents `null`/empty.

The ⟨*operatorOperation*⟩ rule has three main options: retain, discard and execute. The grouping operation can be performed during the selection of operators (⟨*selectOperators*⟩) for performing these three main operations and is represented by the ⟨*operatorGrouping*⟩ rule. As seen in the rule ⟨*operatorSelectionType*⟩, there are two types of operator selection: selection by element or selection by group. If the former is used, then operators are selected individually, otherwise the strategy first selects a group of operators according to a grouping operation and then apply another selection operation in each group.

There is only one type of operator execution in Sentinel, which is the default execution of operators provided by most mutation tools. However, this can be extended by adding new rules in the grammar, specifically in ⟨*operatorExecutionType*⟩, and then implementing new operations based on the types of executions provided by the tools. For instance, in future work,

⟨*operatorOperation*⟩ ::= `Retain Operators` ⟨*selectOperators*⟩
| `Discard Operators` ⟨*selectOperators*⟩
| `Execute Operators` ⟨*operatorExecutionType*⟩ ⟨*selectOperators*⟩

⟨*selectOperators*⟩ ::= ⟨*operatorSelectionType*⟩ ⟨*quantity*⟩
| ⟨*operatorSelectionType*⟩ ⟨*percentage*⟩

⟨*operatorSelectionType*⟩ ::= `Select Operators` ⟨*selectionType*⟩ ⟨*operatorSorting*⟩
| `Select Operators by Groups` ⟨*operatorGroupSelectionType*⟩ ⟨*selectOperators*⟩

⟨*operatorSorting*⟩ ::= ⟨*operatorAttribute*⟩ ⟨*sortingDirection*⟩ | λ

⟨*operatorGroupSelectionType*⟩ ::= ⟨*operatorGrouping*⟩ ⟨*selectionType*⟩ ⟨*operatorGroupSorting*⟩

⟨*operatorGrouping*⟩ ::= ⟨*operatorAttribute*⟩

⟨*operatorGroupSorting*⟩ ::= ⟨*operatorGroupAttribute*⟩ ⟨*sortingDirection*⟩ | λ

⟨*operatorAttribute*⟩ ::= `Type|Mutant Quantity`

⟨*operatorGroupAttribute*⟩ ::= `Quantity in Group|Mutant Quantity in Group`

⟨*operatorExecutionType*⟩ ::= `Conventional`

Grammar 3.2: Sentinel grammar (part 2). Rules for operator operations.

we can extend Sentinel to work with HOMs and then implement operations to execute mutation operators several times in a single mutant.

Similarly, Grammar 3.3 presents the third part of Sentinel grammar, which relates to mutant operations.

The mutant rules are very similar to the operator rules. The main differences are on the attributes used for grouping and sorting, and that there is no mutant execution rule. Mutants are executed after the execution of the strategy.

This grammar also allows the instantiation of most conventional mutant reduction strategies. To exemplify that, consider the following integer array: {0, 2, 1, 0, 0, 1, 9, 1, 0, 1, 0, 1, 1, 0, 3} used as a genotype for building a strategy. The result of the GPM for this array would be a Random Mutant Sampling of 10% of all available mutants. Figure 3.4 depicts how the GPM interprets the integer array and gradually builds the mentioned strategy.

In Figure 3.4, each non-terminal node is followed by a gene consumed for choosing an option (⟨*node*⟩ – *gene*), e.g., the first rule ⟨$defaultOperation$⟩ with the consumed gene 0. The selected option for a node is given right below that node. Some rules do not consume genes, because they only have one option, such as ⟨$strategy$⟩ and ⟨$operatorExecutionType$⟩. Still using the first ⟨$defaultOperation$⟩ node as an example, the selected option is ⟨$operatorOperation$⟩⟨$defaultOperation$⟩. Thus, there are two decisions to be made in this step, one for choosing an option for ⟨$operatorOperation$⟩ and another one for choosing an option for ⟨$defaultOperation$⟩. The gene 2 is consumed for selecting an option for ⟨$operatorOperation$⟩, resulting in the option “Execute Opera-

⟨*mutantOperation*⟩ ::= `Retain Mutants` ⟨*selectMutants*⟩
| `Discard Mutants` ⟨*selectMutants*⟩

⟨*selectMutants*⟩ ::= ⟨*mutantSelectionType*⟩ ⟨*quantity*⟩
| ⟨*mutantSelectionType*⟩ ⟨*percentage*⟩

⟨*mutantSelectionType*⟩ ::= `Select Mutants` ⟨*selectionType*⟩ ⟨*mutantSorting*⟩
| `Select Mutants by Groups` ⟨*mutantGroupSelectionType*⟩ ⟨*selectMutants*⟩

⟨*mutantSorting*⟩ ::= ⟨*mutantAttribute*⟩ ⟨*sortingDirection*⟩ | λ

⟨*mutantGroupSelectionType*⟩ ::= ⟨*mutantGrouping*⟩ ⟨*selectionType*⟩
⟨*mutantGroupSorting*⟩

⟨*mutantGrouping*⟩ ::= ⟨*mutantAttribute*⟩

⟨*mutantGroupSorting*⟩ ::= ⟨*mutantGroupAttribute*⟩ ⟨*sortingDirection*⟩ | λ

⟨*mutantAttribute*⟩ ::= `Operator Type` | `Operator`

⟨*mutantGroupAttribute*⟩ ::= `Quantity in Group`

Grammar 3.3: Sentinel grammar (part 3). Rules for mutant operations.

tors ⟨$operatorExecutionType$⟩⟨$selectOperators$⟩". Because this option has non-terminal rules, then more genes must be consumed to decide which options will be chosen for ⟨$operatorExecutionType$⟩ and ⟨$selectOperators$⟩. One should keep in mind that the gene consuming is done like a depth tree, i.e, reaching a terminal rule, then stepping back one level to the previous rule and checking for more non-terminal rules to make decisions. This whole process continues until there are no more decisions to be made or when the GE wrapping limit is reached, in which case the solution is considered infeasible.

Hence, the strategy generated with the chromosome {0, 2, 1, 0, 0, 1, 9, 1, 0, 1, 0, 1, 1, 0, 3} will perform the following operations: i) sequentially select 100% of the operators without sorting; ii) execute the selected operators with conventional operator execution; iii) randomly select 10% of the generated mutants without sorting; iv) retain the selected mutants in the mutant array; and v) store the retained mutants in the final mutant set M'.

We expect to generate strategies that are very similar to conventional ones, but also unforeseen strategies combining features from different existing strategies. Using this grammar, a possible unforeseen strategy could be one that randomly discards 50% of the operators, executes the remaining, group the generated mutants by operator type and then sequentially retains 80% of the mutants in each group. Moreover, the grammar is customizable, thus it can be changed to focus on specific operations. That way, Sentinel can work as a tool to automatically configure existing strategies by simply fixing some operations and allowing the variation of conventional strategies parameters.

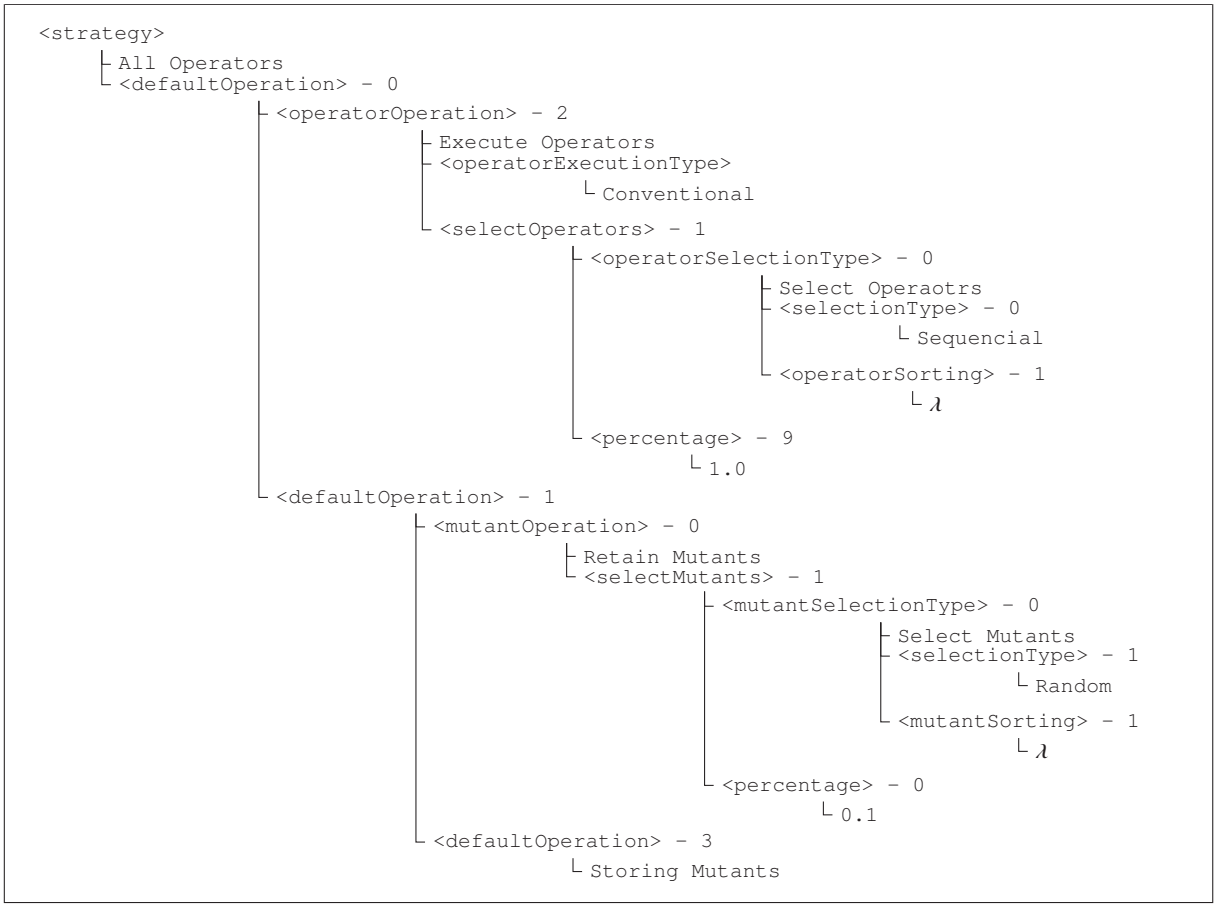


Figure 3.4: GPM example using the proposed grammar

3.6 Implementation Aspects

Sentinel focuses on generating strategies for the mutation testing of units in Java programs. We decided to firstly implement Sentinel for testing Java programs given the great popularity of this programming language among works on mutation testing. For that purpose, Sentinel uses PIT[2] [16]. Sentinel was integrated with the command line framework of PIT 1.2.0, which provides classes and objects for manipulating mutants, test cases and executing them. We did not change the default configuration of PIT for integrating it with Sentinel. We chose to use PIT because it is one of the fastest Java mutation tools available and because it is widely used in mutation testing works [42, 65, 70].

The GE algorithm used by Sentinel implements several multi-objective mechanisms, because the strategies are evaluated considering the two objectives presented in this chapter. Therefore, MOEA specific procedures such as parent selection, population replacement and fitness assignment will be used to support the GE execution. We use jMetal 5 [30] as the main framework for implementing the GE algorithm, because it already implements several MOEAs and is easily extended to other kinds of optimization algorithms.

[2]http://pitest.org/

3.7 Final Remarks

This chapter presented the proposed approach and its main implementation aspects. We presented the main structure of Sentinel, the objective functions used to evaluate the generated strategies, the solution representation, the main operations used by Sentinel to build strategies, the proposed grammar and some implementation aspects.

The main advantage of using Sentinel is the automation of the selection and configuration of strategies. With the proposed grammar, Sentinel can generate unforeseen strategies mixing operations from existing ones. Furthermore, this grammar can be changed to encompass only specific operations in order to allow the generation of conventional strategies from the literature, i.e., Sentinel can be customized to work as an automatic configuration approach for existing strategies. Either way, because the evolutionary process focuses on minimizing the execution time and maximizing the mutation score, we expect that the generated strategies are capable of reducing the mutation cost while maintaining the test efficacy.

Sentinel is an original approach since, as far as we are aware, there are no other works in the literature that automatically generate mutant reduction strategies. Furthermore, Sentinel uses the actual execution time of strategies and generated mutants as an objective for optimization. It also allows the usage of other objectives, giving the possibility of generating strategies tailored to the needs of the tester. Other works from the literature in mutant reduction only use number of mutants and mutation score. To the best of our knowledge, this is the first work actually measuring mutants execution time for a more accurate assessment of cost reduction.

As a secondary contribution, we make the grammars and source code of Sentinel publicly available at `https://github.com/GiovaniGuizzo/Sentinel`.

The next chapter presents the experimental evaluation for assessing the feasibility of Sentinel. These experiments aim at comparing the performance of the generated strategies with the performance of conventional strategies. For doing so, we applied Sentinel in several real world systems and used multiple quality indicators and statistical tests for the result analysis.

Chapter 4

Experiments

Before proceeding, it is important to recapitulate the hypothesis of this work: **"An approach based on hyper-heuristics is capable of generating strategies that contribute to reduce the cost of the mutation testing activity without losing efficacy when compared to conventional strategies from the literature"**. Hence, the main goal of this experimental evaluation is to determine the capability of the strategies generated by Sentinel in reducing the overall execution time of mutation testing and in maintaining the mutation score. For performing this assessment, we used 10 real world open-source software and 4 versions of each system. We then compared the results of the strategies generated by Sentinel to three conventional strategies of the literature to answer the Research Questions (RQs) and then answer the main hypothesis of this work.

This chapter is organized as follows. Section 4.1 presents the RQs and how we designed the experiments to answer them. Section 4.2 presents the experimental subjects and their characteristics. Section 4.3 describes the parameters used by the GE algorithm implemented in Sentinel. Section 4.4 shows the results and the answers to the RQs. In Section 4.5 we discuss the feasibility of Sentinel considering its training time and the main scenarios where it should be used. Section 4.6 shows some examples of generated strategies and how they work to reduce mutant sets. The threats to the validity of these experiments are shown in Section 4.7. Finally, Section 4.8 concludes this chapter.

4.1 Research Questions

Considering the objective of our experiment, we designed three RQs to guide our evaluation, each of which concerned with a particular subject. RQ1 is a sanity check to determine if Sentinel does not work at random, i.e., if the GE implemented with Sentinel is actually contributing to the generation of good strategies. RQ2 regards the quality of the strategies generated by Sentinel and is used to assess if Sentinel is actually capable of generating strategies that are better than conventional ones from the literature. RQ3 is concerned with the reusability of strategies, i.e, by answering this question we intend to evaluate if Sentinel is capable of doing what it was proposed for: generating strategies that can be reused in subsequent versions of the software. All of these RQs were specially designed to answer the main hypothesis of this work.

4.1.1 RQ1 – Sanity Check

Is Sentinel better than a random strategy generation algorithm?

We want to check if the hyper-heuristic mechanism used with Sentinel is indeed responsible for the generation of good strategies during the training phase. Hence, the null

hypothesis to be rejected for this RQ is as follows: **"The strategies generated by Sentinel are not better than strategies generated by a random hyper-heuristic"**.

To answer this question and evaluate our hypothesis, we executed Sentinel with a GE algorithm (default Sentinel configuration) and Sentinel with a random generation hyper-heuristic. In other words, we tested the generation efficacy of Sentinel when using a GE to its efficacy when using a random search algorithm with the same components (objective functions, operations, grammar file, and so on). For succinctness, the GE version is herein called "Sentinel" and the random version is herein called just "Random".

Both versions were executed for 30 independent runs on 10 different systems used in the literature [42, 63, 70, 105, 106]. Since Sentinel is a multi-objective approach, each independent run generates an approximated Pareto front, where each solution of the front is a generated strategy. We need multi-objective quality indicators [109] for comparing the quality of the fronts. In this work we use hypervolume (HV) and Inverted Generational Distance (IGD) [109], both described in Section 2.1.2. Because the true Pareto front for the problem we investigate herein is unknown and infeasible to discover, we create such a front with all the non-dominated solutions found by all algorithms.

We also use the Kruskal-Wallis [67] and Vargha-Delaney A_{12} effect size [99] statistical tests for assessing if there is any statistical difference between the algorithms and for computing the magnitude of the difference, respectively, as suggested in [4]. We chose these statistical tests because they are non-parametric, i.e., they must be used when we cannot assume the normal distribution of our data.

Kruskal-Wallis [67] is a statistical test that can show if the data of two groups are statistically different given a confidence interval. In this work we assume the confidence of 95%, hence if the computed *p-value* is lower than 0.05, then there is statistical difference between the groups. Kruskal-Wallis is actually an extension of Wilcoxon rank sum test [75] by allowing comparisons between three or more groups. It works as the Wilcoxon test when only two groups are being compared.

While Kruskal-Wallis determines if there is statistical difference between groups, the Vargha-Delaney A_{12} effect size [99] gives the magnitude of the difference between two groups: A and B, or left and right. According to [99], the A_{12} value varies between $[0, 1]$, where 0.5 represents absolute no difference between the two groups, values below 0.5 represent that group B obtains greater values than group A, and values above 0.5 represent that A obtains greater values than B. Values in $]0.44, 0.56[$ represent negligible differences, values in $[0.56, 0.64[$ and $]0.36, 0.44]$ represent small differences, values in $[0.64, 0.71[$ and $]0.29, 0.44]$ represent medium differences, and values in $[0.0, 0.29]$ and $[0.71, 1.0]$ represent large differences.

4.1.2 RQ2 – Strategies Generated by Sentinel versus Conventional Strategies

How are the strategies generated by Sentinel compared to the ones from the literature?

After assessing if the strategies generated by Sentinel are better than random guessing, we compare these strategies to conventional mutant reduction strategies commonly used in the literature: Selective Mutation (SM), Random Operator Selection (ROS) and Random Mutant Sampling (RMS).

For RQ2, the null hypothesis is: **"The strategies generated by Sentinel are not better than conventional strategies from the literature"**. By rejecting the null hypothesis, we can assert that using strategies generated by Sentinel is better than using conventional strategies in terms of execution time and mutation score. Given the multiple Pareto fronts

generated by the strategies, we applied the same indicators (hypervolume and IGD) and statistical tests (Kruskal-Wallis and Vargha-Delaney A_{12}) as RQ1 to analyse the results and evaluate our hypothesis.

To answer this question, we executed all the non-dominated strategies generated by Sentinel in the training phase. Then we compared with 3 types of conventional strategies, which in turn are simple heuristics that do not need training. Each type of strategy has multiple strategy variations that differ on their configuration parameters. For example, RMS can sample 90%, 80%, 70% mutants and so on, whereas SM can remove 1, 2, 3 operators and so on. Each strategy variation is considered a single strategy of the given strategy type. Those strategies are grouped by type and are compared as a group. In this sense, after executing the strategies of a given type, we obtain a Pareto front where each solution is a strategy variation with their respective objective values for the given problem instance. When comparing RMS, ROS, SM and Sentinel, we actually execute the strategies of each group, compute their objectives and then compare. Hence, when we state that Sentinel is better than ROS for example, we mean that the strategies of a group are, in overall, better than the strategies of the other group based on the HV and/or IGD indicators.

For each of the 10 systems, we selected 4 different versions for the strategies to be tested on. Hence, all strategies were executed over 4 different versions of the 10 systems with 30 independent runs each, for a total of 1200 independent runs for each type of strategy. Bear in mind that the strategies were generated by Sentinel in the training phase using only the first version of each system and then tested on all versions of the same system. Hence, we only performed the training once per system.

The conventional strategies were implemented using the same framework with which Sentinel was implemented, thus they are represented in the same way as the generated strategies (as described in Section 3.3). A set of strategies with parameter variations was created for each of the 3 types of conventional strategies: i) RMS randomly selecting from 10% to 90% (steps of 10%) of mutants; ii) ROS selecting from 10% to 90% (steps of 10%) of the operators; and iii) SM excluding from 1 to 6 (steps of 1) of the operators that generate the most mutants. Each of the 4 types of strategies (Sentinel and the 3 conventional ones) yielded a Pareto front for each of the 30 independent runs and for each system version.

4.1.3 RQ3 – Sentinel Strategies Reusability

Are the strategies generated by Sentinel reusable in subsequent versions of the software?

With this RQ we intend to evaluate if strategies generated by Sentinel can be reused over newer versions (versions newer than the one in which Sentinel was trained) of the software before they start to become ineffective. The null hypothesis for RQ3 is: **"The strategies generated by Sentinel cannot be reused in newer versions of the software in which Sentinel was trained without becoming ineffective"**. This analysis can give us insights about when we need to perform the training again to maintain the effectiveness of the generated strategies, and about the cost-benefit of Sentinel when compared to the conventional strategies.

To answer this question, we used the same data collected for answering RQ2. For these experiments we purposely selected subsequent versions of each system to emulate a scenario in which Sentinel was designed to be used. In this scenario the tester would train Sentinel on a given version, use the strategies on this version and then reuse the generated strategies on newer versions of the same system. With that in mind, the training was done on the first version of the systems, tested on the same versions and then tested on 3 subsequent versions. With that data we

can analyse if the strategies can maintain their effectiveness in terms of mutation score and cost reduction on newer and unforeseen versions of the software.

The result evaluation followed the previous RQs, with hypervolume and IGD as indicators, and Kruskal-Wallis and Vargha-Delaney as statistical tests.

4.2 Subjects

In preliminary experiments we observed that the experiments designed to answer the RQs would take way too long to execute given the great cost associated to the execution of mutants. In this sense, we had to limit the number of subjects and versions to a reasonable amount in order to perform the experiments in feasible time. We selected 10 real world open source Java programs used in mutant reduction works of the literature [42, 63, 70, 104]. During the analysis of the subjects, we only considered the systems for which: i) the source code is compiled with no errors; ii) the test cases are compiled with no errors; iii) the test cases pass without failing; iv) there are several versions; and v) PIT is capable of applying the mutation. The subjects are:

1. Apache Commons Beanutils[1] [42] – a library for wrapping around reflection and introspection;
2. Apache Commons Codec[2] [42] – an encoder and decoder for several formats such as Base64 and Hexadecimal;
3. Apache Commons Collections[3] [42, 70] – a library for collections and arrays manipulation developed and maintained by the Apache Foundation;
4. Apache Commons Lang[4] [42] – utility classes for the *java.lang* package;
5. Apache Commons Validator[5] [42] – client and server validation library;
6. JFreeChart[6] [42, 63, 70] – a framework for manipulating charts in Java;
7. JGraphT[7] [42] – a library with classes and algorithms for graphs;
8. Joda-Time[8] [42, 63, 70] – a date and time library for Java;
9. Object-Graph Navigation Language (OGNL)[9] [42] – a simple expression language for Java object's properties; and
10. Wire[10] [104] – a project for Protocol Buffers on the Android platform.

[1] https://github.com/apache/commons-beanutils
[2] https://github.com/apache/commons-codec
[3] https://github.com/apache/commons-collections
[4] https://github.com/apache/commons-lang
[5] https://github.com/apache/commons-validator
[6] https://github.com/jfree/jfreechart
[7] https://github.com/jgrapht/jgrapht
[8] https://github.com/JodaOrg/joda-time
[9] https://github.com/jkuhnert/ognl
[10] https://github.com/square/wire

Because these programs have several minor and major versions and it would be impracticable to test Sentinel on all of these versions, we decided to use only the oldest major versions of each program that could be mutated by PIT for the training phase and 3 subsequent versions for the testing phase. This allows the experimentation on different versions for a more reliable comparison, while also allowing us to answer RQ3.

Furthermore, all of those systems have test suites along with their source code that are used by the developers. None of the testing suites obtain 1.0 mutation score by default, but the objective function `SCORE` presented in Section 3.2 aims at finding strategies that obtain a similar score regardless of whether it is 1.0 or not. Hence, we considered the mutation score of such systems as they are provided by the developers.

Some of those systems presented some minor compilation problems and few test cases that would not pass by default. Instead of discarding them, we decided to solve these minor issues by doing minor changes to building scripts (e.g. fixed missing dependencies, Java version used for compilation, directory paths) and by removing not more than 3 failing test cases from the test suites. The dataset with those fixed issues can be found at `http://bit.ly/SentinelDataset`.

The selected systems vary on size and type. The properties of these programs are presented in Table 4.1. The first column shows the name and version of the program; the second column presents the number of Classes and Interfaces; the third column contains the number of logical lines of code (LLOC) of the source code; the fourth column shows the number of available test cases; the fifth column shows the LLOC for the test cases; and the sixth column presents the number of mutants generated by PIT.

4.3 Setup

During the integration of Sentinel with PIT, we maintained the default options for the tool. Therefore, we applied Sentinel using only the default mutation operators of PIT (7 operators). Even though it may be a low number of operators when compared to other tools and seem like that it is already well optimized, our intention here is to improve even more the optimization already implemented in PIT. The GE parameters used in the experiment are presented in Table 4.2. These parameters were chosen based on other works of the literature [45, 50, 76, 77].

It is important to restate that the strategies cost here relates to CPU Time for executing the reduced set of mutants. Therefore, we did not compare the strategies based on number of mutants such as in other works of the literature [42, 71, 106], but rather compared the mutation execution time for a better accuracy of cost measurement. Hence, one can deduce that the experiments were expensive to perform. For that reason, we used machines available on Microsoft Azure, a cloud computing platform. All the experiments were executed in similar machines for a fair measurement of time.

4.4 Results

This section summarizes and discusses the results obtained in the experimentation. Subsection 4.4.1 presents the results regarding RQ1, where Sentinel is compared to a Random hyper-heuristic generation. Subsection 4.4.2 presents the results regarding RQ2, where the strategies generated by Sentinel were compared to conventional strategies found in the literature. Finally, Subsection 4.4.3 presents the results for answering RQ3, where the reusability of the strategies on newer versions of the systems is evaluated.

Table 4.1: Program versions used in the experiments.

Program	Classes	LLOC	Test Cases	Tests LLOC	Mutants
beanutils-1.8.0	134	11,279	877	20,486	2,827
beanutils-1.8.1	134	11,362	892	20,934	2,855
beanutils-1.8.2	134	11,362	893	20,966	2,856
beanutils-1.8.3	134	11,376	896	21,033	2,857
codec-1.4	30	2,994	284	6,846	1,587
codec-1.5	43	3,551	380	8,307	1,895
codec-1.6	74	4,554	421	9,034	2,196
codec-1.11	94	8,109	875	11,774	3,473
collections-3.0	422	22,842	1,896	22,828	7,488
collections-3.1	438	25,372	2,346	25,734	8,298
collections-3.2	449	26,323	2,566	29,076	8,637
collections-3.2.1	449	26,323	2,566	29,076	8,632
lang-3.0	148	18,997	1,902	31,008	9,072
lang-3.0.1	149	19,495	1,964	31,804	9,328
lang-3.1	150	19,499	1,976	32,446	9,333
lang-3.2	187	22,532	2,390	38,963	10,970
validator-1.4.0	66	5,411	414	6,367	1,811
validator-1.4.1	67	6,031	442	7,389	1,917
validator-1.5.0	72	6,669	481	7,922	1,979
validator-1.5.1	72	7,014	486	8,051	1,982
jfreechart-1.0.0	480	68,796	1,023	26,823	23,417
jfreechart-1.0.1	481	68,663	1,027	27,016	23,490
jfreechart-1.0.2	528	73,162	1,073	28,504	24,091
jfreechart-1.0.3	575	77,621	1,234	32,825	26,401
jgrapht-0.9.0	252	12,978	188	7,030	2,976
jgrapht-0.9.1	264	13,822	647	8,184	3,147
jgrapht-0.9.2	294	15,661	728	10,046	3,825
jgrapht-1.0.0	303	16,417	1,201	13,609	4,270
joda-2.8	246	28,479	2,967	54,645	10,225
joda-2.8.1	246	28,479	2,967	54,645	10,225
joda-2.8.2	246	28,479	2,967	54,645	10,225
joda-2.9	246	28,624	2,985	54,985	10,321
ognl-3.1	129	16,103	54	6,229	5,650
ognl-3.1.1	129	16,102	53	6,223	5,652
ognl-3.1.2	129	16,103	56	6,252	5,652
ognl-3.1.3	129	16,109	57	6,268	5,654
wire-2.0.0	33	1,354	70	1,776	513
wire-2.0.1	33	1,354	70	1,776	513
wire-2.0.2	33	1,353	70	1,776	513
wire-2.0.3	34	1,405	71	1,794	524

4.4.1 RQ1 – Sanity Check

Table 4.3 presents the mean hypervolume and mean IGD results for the comparison. The standard deviation is presented in parentheses. The best indicator values, and *p-values* lower than 0.05 are highlighted in bold. If two indicator values are highlighted in bold in the same row, then for that given system the statistical test showed no difference. For the effect size presented in the table, Sentinel is group A and the Random algorithm is group B. For hypervolume, an effect size value closer to 1 is favourable to Sentinel (greater hypervolume values more often), whereas for IGD, an effect size value closer to 0 is better for Sentinel (lower IGD values more often). Greater HV

Table 4.2: Parameters for the GE used by Sentinel.

Parameter	Value
Independent Runs	30
Strategy Repetitions	5
Maximum Evaluations	10,000
Population Size	100
Crossover Operator	Single Point Crossover
Crossover Probability	100%
Mutation Operator	Random Integer Mutation
Mutation Probability	1%
Prune Probability	10%
Duplicate Probability	10%
Lower Gene Bound	0
Upper Gene Bound	179
Maximum Chromosome Length	100
Minimum Chromosome Length	15
Maximum Wraps	10

and lower IGD values are better for Sentinel. Large effect size differences in favour of Sentinel are also highlighted in bold. L stands for large, M for medium, S for small and N for negligible.

Table 4.3: RQ1: HV and IGD mean results for the training data.

Program	Ind.	Sentinel	Random	*p-value*	effect size
beanutils-1.8.0	HV	**0.87 (0.001)**	0.83 (0.004)	**2.87E-11**	**1 (L)**
	IGD	**1.98E-4 (6.59E-6)**	3.24E-4 (3.58E-5)	**2.87E-11**	**0 (L)**
codec-1.4	HV	**0.91 (0.001)**	0.89 (0.002)	**2.87E-11**	**1 (L)**
	IGD	**2.58E-4 (8.78E-5)**	3.13E-4 (4.89E-5)	**0.003**	0.29 (M)
collections-3.0	HV	**0.83 (0.004)**	0.81 (0.003)	**2.87E-11**	**1 (L)**
	IGD	**2.32E-4 (4.05E-5)**	3.31E-4 (3.78E-5)	**1.48E-9**	**0.05 (L)**
lang-3.0	HV	**0.88 (0.002)**	0.86 (0.003)	**2.87E-11**	**1 (L)**
	IGD	**1.83E-4 (1.49E-5)**	3.08E-4 (3.58E-5)	**2.87E-11**	**0 (L)**
validator-1.4.0	HV	**0.90 (0.002)**	0.87 (0.005)	**2.87E-11**	**1 (L)**
	IGD	**2.11E-4 (1.37E-5)**	4.12E-4 (3.60E-5)	**2.87E-11**	**0 (L)**
jfreechart-1.0.0	HV	**0.92 (0.002)**	0.89 (0.01)	**2.87E-11**	**1 (L)**
	IGD	**2.36E-4 (4.04E-5)**	5.48E-4 (9.08E-5)	**3.17E-11**	**0.001 (L)**
jgrapht-0.9.0	HV	**0.93 (0.001)**	0.90 (0.05)	**2.87E-11**	**1 (L)**
	IGD	**2.78E-4 (7.15E-5)**	4.96E-4 (5.47E-5)	**1.15E-10**	**0.02 (L)**
joda-time-2.8	HV	**0.85 (0.002)**	0.82 (0.004)	**2.87E-11**	**1 (L)**
	IGD	**1.88E-4 (1.64E-5)**	3.62E-4 (3.46E-5)	**2.87E-11**	**0 (L)**
ognl-3.1	HV	**0.97 (0.002)**	0.95 (0.003)	**2.87E-11**	**1 (L)**
	IGD	**5.15E-4 (1.64E-4)**	6.18E-4 (1.09E-4)	**0.004**	0.29 (M)
wire-2.0.0	HV	**0.92 (0.004)**	0.90 (0.004)	**2.87E-11**	**1 (L)**
	IGD	**4.89E-4 (1.39E-4)**	**4.89E-4 (5.99E-5)**	0.35	0.43 (N)

As shown in Table 4.3,Sentinel is able to obtain better results than the Random hyper-heuristic for all programs and for both quality indicators. The only statistical equivalence in these results is for IGD in wire-2.0.0, where Kruskal-Wallis yielded no difference and Vargha-Delaney A_{12} showed a negligible effect size. We already expected a negative or equal result for this system, as it is the smallest one in terms of number of mutants, LLOC and number of test LLOC. For such a system, there is not much that can be optimized. However, the hypervolume comparison

for this system yielded statistical difference and a large effect size, thus Sentinel still managed to outperform the Random hyper-heuristic in relation to this indicator.

In overall, the Kruskal-Wallis statistical test showed statistical difference with 95% confidence (*p-value* < 0.05) for 19 out of 20 comparisons. The Vargha-Delaney A_{12} effect size showed large differences in 17 out of 20 comparisons in favour of Sentinel, and only 2 medium and 1 negligible differences for the IGD comparisons.

> **RQ1:** We can reject the null hypothesis formulated for RQ1 and assert that Sentinel is indeed capable of generating strategies in a better way than a Random hyper-heuristic.

4.4.2 RQ2 – Sentinel versus Conventional Strategies

Similar to the previous subsection, Tables 4.4 and 4.5 present, respectively, the hypervolume results and the IGD results for the comparisons. The standard deviation is presented in parentheses. The best values and *p-values* lower than 0.05 are highlighted in bold. If two or more indicator values are highlighted in bold in the same row, then for that given system the statistical test showed no difference between the best strategy and the other highlighted strategies. For simplicity, from now on RMS stands for Random Mutant Sampling, ROS stands for Random Operator Selection, and SM stands for Selective Mutation. Greater HV values are better. Lower IGD values are better.

We can observe that overall, the strategies generated by Sentinel are able to outperform the conventional strategies according to both indicators. More precisely, for 70 out of 80 statistical comparisons, Sentinel strategies outperformed the conventional strategies with significant differences, and for 6 comparisons Sentinel presented statistically equivalent results. Hence, for 76 out of 80 comparisons (95%), Sentinel strategies outperformed or presented equivalent results to the other ones. Only for wire SM obtained significantly better results in terms of HV, but not for IGD. Furthermore, for codec-1.5/1.6/1.11 and beanutils-1.8.1/1.8.2/1.8.3, SM showed no statistical significant difference regarding HV, but fell behind in the IGD comparison.

Table 4.6 shows the Vargha-Delaney A_{12} effect size results for the HV and IGD values. In the comparison, Sentinel is subject A and the conventional strategy represented by the given column is subject B. Greater HV effect size values (> 0.5) and lower IGD effect size values (< 0.5) are favourable to Sentinel. Values highlighted in bold show large differences in favour of Sentinel. L stands for large, M for medium, S for small and N for negligible.

Regarding the Vargha-Delaney A_{12} effect size, Sentinel obtained favourable large differences in 227 out of 240 comparisons (~95%). Selective Mutation obtained favourable large difference only for the HV comparisons in wire. Furthermore, Random Operator Selection obtained large favourable difference for the hypervolume comparison in wire-2.0.3.

In overall, Sentinel strategies are able to outperform the conventional strategies with large differences in approximately 95% of the cases. The only exceptions are codec-1.5/1.6/1.11 and beanutils-1.8.1/1.8.2/1.8.3 for which the generated strategies obtained statistically equivalent results to SM regarding hypervolume, and wire for which the generated strategies performed worse than SM regarding hypervolume. As observed in the previous subsection, the strategies generated by Sentinel are not very good at reducing the cost and maintaining the mutation score for the wire program. This is probably due to the small search space of mutant sets, thus the time taken by the strategies to find a reduced set is not worth the results. However, when considering IGD, the strategies generated by Sentinel always presented statistically better results with large effect size differences, even for wire.

Table 4.4: RQ2: HV results comparing Sentinel and the conventional strategies.

Program	Sentinel	SM	RMS	ROS	p-value
beanutils-1.8.0	**0.81 (0.02)**	0.79 (0.02)	0.76 (0.03)	0.77 (0.04)	**3.3E-10**
beanutils-1.8.1	**0.79 (0.01)**	**0.79 (0.03)**	0.73 (0.02)	0.75 (0.04)	**1.6E-11**
beanutils-1.8.2	**0.79 (0.02)**	**0.78 (0.03)**	0.74 (0.03)	0.76 (0.04)	**1.9E-09**
beanutils-1.8.3	**0.78 (0.03)**	**0.78 (0.03)**	0.73 (0.03)	0.75 (0.04)	**1.9E-10**
codec-1.4	**0.89 (0.01)**	0.87 (0.02)	0.78 (0.005)	0.84 (0.02)	< **2.2E-16**
codec-1.5	**0.90 (0.005)**	**0.89 (0.01)**	0.80 (0.005)	0.84 (0.02)	< **2.2E-16**
codec-1.6	**0.88 (0.003)**	**0.89 (0.02)**	0.78 (0.004)	0.82 (0.02)	< **2.2E-16**
codec-1.11	**0.91 (0.01)**	**0.89 (0.01)**	0.83 (0.01)	0.83 (0.03)	< **2.2E-16**
collections-3.0	**0.92 (0.01)**	0.74 (0.01)	0.86 (0.01)	0.79 (0.03)	< **2.2E-16**
collections-3.1	**0.89 (0.01)**	0.72 (0.02)	0.84 (0.01)	0.77 (0.02)	< **2.2E-16**
collections-3.2	**0.90 (0.01)**	0.73 (0.02)	0.84 (0.01)	0.77 (0.02)	< **2.2E-16**
collections-3.2.1	**0.90 (0.01)**	0.73 (0.01)	0.85 (0.01)	0.77 (0.02)	< **2.2E-16**
lang-3.0	**0.95 (0.004)**	0.88 (0.01)	0.90 (0.01)	0.85 (0.02)	< **2.2E-16**
lang-3.0.1	**0.94 (0.005)**	0.86 (0.01)	0.90 (0.01)	0.84 (0.02)	< **2.2E-16**
lang-3.1	**0.94 (0.01)**	0.86 (0.02)	0.90 (0.01)	0.84 (0.02)	< **2.2E-16**
lang-3.2	**0.95 (0.01)**	0.87 (0.02)	0.91 (0.01)	0.86 (0.02)	< **2.2E-16**
validator-1.4.0	**0.86 (0.004)**	0.79 (0.01)	0.75 (0.005)	0.76 (0.02)	< **2.2E-16**
validator-1.4.1	**0.87 (0.01)**	0.80 (0.02)	0.76 (0.01)	0.75 (0.02)	< **2.2E-16**
validator-1.5.0	**0.86 (0.004)**	0.77 (0.01)	0.74 (0.01)	0.74 (0.03)	< **2.2E-16**
validator-1.5.1	**0.85 (0.01)**	0.76 (0.02)	0.74 (0.01)	0.74 (0.02)	< **2.2E-16**
jfreechart-1.0.0	**0.97 (0.001)**	0.84 (0.001)	0.94 (0.001)	0.84 (0.02)	< **2.2E-16**
jfreechart-1.0.1	**0.97 (0.003)**	0.85 (0.01)	0.94 (0.005)	0.85 (0.02)	< **2.2E-16**
jfreechart-1.0.2	**0.97 (0.004)**	0.85 (0.01)	0.94 (0.01)	0.85 (0.02)	< **2.2E-16**
jfreechart-1.0.3	**0.97 (0.003)**	0.86 (0.01)	0.94 (0.005)	0.85 (0.02)	< **2.2E-16**
jgrapht-0.9.0	**0.91 (0.01)**	0.86 (0.01)	0.86 (0.01)	0.84 (0.02)	< **2.2E-16**
jgrapht-0.9.1	**0.91 (0.01)**	0.87 (0.01)	0.85 (0.01)	0.83 (0.02)	< **2.2E-16**
jgrapht-0.9.2	**0.90 (0.004)**	0.87 (0.01)	0.85 (0.01)	0.87 (0.02)	< **2.2E-16**
jgrapht-1.0.0	**0.89 (0.005)**	0.87 (0.02)	0.85 (0.01)	0.87 (0.01)	< **2.2E-16**
joda-time-2.8	**0.91 (0.01)**	0.72 (0.01)	0.83 (0.01)	0.80 (0.02)	< **2.2E-16**
joda-time-2.8.1	**0.90 (0.005)**	0.71 (0.01)	0.82 (0.01)	0.79 (0.02)	< **2.2E-16**
joda-time-2.8.2	**0.90 (0.005)**	0.71 (0.01)	0.81 (0.01)	0.78 (0.02)	< **2.2E-16**
joda-time-2.9	**0.90 (0.01)**	0.70 (0.01)	0.81 (0.01)	0.78 (0.02)	< **2.2E-16**
ognl-3.1	**0.99 (0.003)**	0.98 (0.003)	0.97 (0.01)	0.92 (0.02)	< **2.2E-16**
ognl-3.1.1	**0.99 (0.003)**	0.97 (0.01)	0.92 (0.01)	0.92 (0.03)	< **2.2E-16**
ognl-3.1.2	**0.99 (0.001)**	0.97 (0.01)	0.97 (0.002)	0.92 (0.02)	< **2.2E-16**
ognl-3.1.3	**0.98 (0.001)**	0.96 (0.01)	0.91 (0.002)	0.91 (0.02)	< **2.2E-16**
wire-2.0.0	0.80 (0.01)	**0.83 (0.01)**	0.62 (0.01)	0.80 (0.01)	< **2.2E-16**
wire-2.0.1	0.84 (0.02)	**0.87 (0.01)**	0.65 (0.02)	0.84 (0.02)	< **2.2E-16**
wire-2.0.2	0.83 (0.01)	**0.86 (0.01)**	0.64 (0.01)	0.83 (0.02)	< **2.2E-16**
wire-2.0.3	0.66 (0.01)	**0.74 (0.05)**	0.51 (0.01)	0.67 (0.01)	< **2.2E-16**

RQ2: We can reject the null hypothesis formulated for RQ2 and assert that, in overall, the strategies generated by Sentinel perform better than the conventional strategies used in this work.

4.4.3 RQ3 – Sentinel Strategies Reusability

As observed in Tables 4.4-4.6, the strategies generated by Sentinel were able to outperform the other strategies in most comparisons. In fact, for 7 out of the 10 analysed programs, the strategies generated by Sentinel with the first version of the software were able to outperform the conventional strategies with large statistical differences in all the subsequent versions considering

Table 4.5: RQ2: IGD results comparing Sentinel and the conventional strategies.

Program	Sentinel	SM	RMS	ROS	p-value
beanutils-1.8.0	**1.61E-4 (6.32E-6)**	2.05E-3 (2.45E-4)	2.17E-3 (8.80E-5)	2.02E-3 (5.73E-4)	**2.1E-15**
beanutils-1.8.1	**1.71E-4 (7.66E-6)**	2.25E-3 (2.86E-4)	2.20E-3 (6.13E-5)	2.10E-3 (5.49E-4)	**1.2E-14**
beanutils-1.8.2	**1.94E-4 (1.51E-5)**	2.24E-3 (3.25E-4)	2.20E-3 (1.00E-4)	2.20E-3 (4.81E-4)	**1.8E-14**
beanutils-1.8.3	**1.89E-4 (1.45E-5)**	2.22E-3 (3.05E-4)	2.18E-3 (7.09E-5)	2.19E-3 (5.11E-4)	**1.2E-14**
codec-1.4	**2.03E-4 (9.83E-6)**	2.57E-3 (1.57E-4)	2.95E-3 (8.15E-5)	2.94E-3 (3.48E-4)	< **2.2E-16**
codec-1.5	**1.70E-4 (6.53E-6)**	2.78E-3 (7.13E-5)	3.40E-3 (8.30E-5)	3.44E-3 (3.17E-4)	< **2.2E-16**
codec-1.6	**1.65E-4 (3.44E-6)**	2.67E-3 (1.04E-4)	3.49E-3 (6.22E-5)	3.55E-3 (4.64E-4)	< **2.2E-16**
codec-1.11	**1.57E-4 (6.48E-6)**	2.19E-3 (4.29E-5)	3.15E-3 (4.93E-5)	3.03E-3 (3.93E-4)	< **2.2E-16**
collections-3.0	**1.73E-4 (1.45E-5)**	1.73E-3 (2.34E-5)	2.31E-3 (4.86E-5)	2.70E-3 (8.29E-4)	< **2.2E-16**
collections-3.1	**1.80E-4 (1.43E-5)**	1.64E-3 (1.56E-5)	2.25E-3 (5.11E-5)	1.85E-3 (4.06E-4)	< **2.2E-16**
collections-3.2	**1.91E-4 (1.67E-5)**	1.56E-3 (2.51E-5)	2.11E-3 (6.53E-5)	1.66E-3 (5.41E-4)	< **2.2E-16**
collections-3.2.1	**1.76E-4 (1.09E-5)**	1.54E-3 (1.71E-5)	2.11E-3 (7.23E-5)	1.82E-3 (6.38E-4)	< **2.2E-16**
lang-3.0	**9.67E-5 (7.90E-6)**	1.74E-3 (2.50E-5)	2.41E-3 (3.57E-5)	1.93E-3 (5.54E-4)	< **2.2E-16**
lang-3.0.1	**9.68E-5 (4.41E-6)**	1.75E-3 (1.54E-5)	2.40E-3 (2.82E-5)	1.80E-3 (5.08E-4)	< **2.2E-16**
lang-3.1	**1.43E-4 (1.48E-5)**	1.84E-3 (3.51E-5)	2.39E-3 (3.35E-5)	1.72E-3 (4.25E-4)	< **2.2E-16**
lang-3.2	**1.78E-4 (3.33E-5)**	1.85E-3 (3.45E-5)	2.41E-3 (2.95E-5)	1.84E-3 (4.46E-4)	< **2.2E-16**
validator-1.4.0	**1.72E-4 (6.38E-6)**	1.71E-3 (7.04E-5)	3.19E-3 (8.53E-5)	2.37E-3 (5.94E-4)	< **2.2E-16**
validator-1.4.1	**1.85E-4 (9.46E-6)**	1.77E-3 (9.72E-5)	3.13E-3 (1.04E-4)	2.75E-3 (5.54E-4)	< **2.2E-16**
validator-1.5.0	**1.83E-4 (5.53E-6)**	1.82E-3 (9.32E-5)	3.04E-3 (8.97E-5)	2.55E-3 (8.34E-4)	< **2.2E-16**
validator-1.5.1	**1.90E-4 (1.10E-5)**	1.79E-3 (9.93E-5)	3.07E-3 (8.16E-5)	2.68E-3 (7.56E-4)	< **2.2E-16**
jfreechart-1.0.0	**1.60E-4 (1.52E-6)**	1.74E-3 (1.96E-5)	2.72E-3 (4.59E-5)	1.63E-3 (2.44E-4)	< **2.2E-16**
jfreechart-1.0.1	**1.50E-4 (6.29E-6)**	1.56E-3 (3.65E-5)	2.74E-3 (4.37E-5)	1.54E-3 (2.98E-4)	< **2.2E-16**
jfreechart-1.0.2	**1.68E-4 (7.61E-6)**	1.58E-3 (4.08E-5)	2.76E-3 (3.62E-5)	1.61E-3 (3.63E-4)	< **2.2E-16**
jfreechart-1.0.3	**1.51E-4 (3.52E-6)**	1.53E-3 (3.32E-5)	2.74E-3 (4.91E-5)	1.65E-3 (4.29E-4)	< **2.2E-16**
jgrapht-0.9.0	**2.23E-4 (1.70E-5)**	1.58E-3 (5.61E-5)	3.72E-3 (7.56E-5)	2.43E-3 (6.82E-4)	< **2.2E-16**
jgrapht-0.9.1	**2.15E-4 (1.44E-5)**	1.86E-3 (6.78E-5)	4.06E-3 (8.42E-5)	2.64E-3 (7.86E-4)	< **2.2E-16**
jgrapht-0.9.2	**2.85E-4 (1.56E-5)**	1.72E-3 (4.53E-5)	3.72E-3 (5.88E-5)	2.25E-3 (5.48E-4)	< **2.2E-16**
jgrapht-1.0.0	**2.27E-4 (1.78E-5)**	1.68E-3 (4.42E-5)	3.65E-3 (7.08E-5)	2.29E-3 (6.34E-4)	< **2.2E-16**
joda-time-2.8	**1.26E-4 (9.44E-6)**	1.66E-3 (2.72E-5)	1.66E-3 (2.43E-5)	1.25E-3 (2.44E-4)	< **2.2E-16**
joda-time-2.8.1	**1.11E-4 (5.59E-6)**	1.70E-3 (2.36E-5)	1.66E-3 (3.10E-5)	1.30E-3 (3.02E-4)	< **2.2E-16**
joda-time-2.8.2	**1.13E-4 (8.08E-6)**	1.71E-3 (2.79E-5)	1.65E-3 (3.35E-5)	1.33E-3 (3.46E-4)	< **2.2E-16**
joda-time-2.9	**1.25E-4 (6.08E-6)**	1.68E-3 (2.02E-5)	1.67E-3 (2.77E-5)	1.36E-3 (3.47E-4)	< **2.2E-16**
ognl-3.1	**9.35E-4 (4.91E-5)**	5.87E-3 (1.12E-4)	6.74E-3 (1.06E-4)	5.19E-3 (9.84E-4)	< **2.2E-16**
ognl-3.1.1	**9.22E-4 (5.10E-5)**	5.75E-3 (7.91E-5)	6.53E-3 (8.59E-5)	4.56E-3 (1.02E-3)	< **2.2E-16**
ognl-3.1.2	**1.06E-3 (1.12E-5)**	5.65E-3 (1.05E-4)	6.49E-3 (1.11E-4)	4.51E-3 (1.08E-3)	< **2.2E-16**
ognl-3.1.3	**1.06E-3 (9.70E-6)**	6.01E-3 (1.28E-4)	6.82E-3 (7.98E-5)	5.14E-3 (1.14E-3)	< **2.2E-16**
wire-2.0.0	**5.08E-4 (2.26E-5)**	3.13E-3 (3.11E-4)	5.77E-3 (2.82E-4)	4.13E-3 (1.02E-3)	< **2.2E-16**
wire-2.0.1	**4.68E-4 (2.34E-5)**	3.18E-3 (2.38E-4)	5.69E-3 (3.37E-4)	3.99E-3 (1.07E-3)	< **2.2E-16**
wire-2.0.2	**4.61E-4 (1.48E-5)**	3.05E-3 (1.55E-4)	5.79E-3 (3.66E-4)	4.09E-3 (1.19E-3)	< **2.2E-16**
wire-2.0.3	**4.68E-4 (1.83E-5)**	3.03E-3 (3.49E-4)	5.62E-3 (2.70E-4)	3.76E-3 (1.02E-3)	< **2.2E-16**

both IGD and HV. For 2 of these 10 programs, the generated strategies showed no statistical differences to SM in the three subsequent versions for HV, but still outperformed ROS and RMS with large statistical differences. Looking at the IGD results, for all 10 systems, Sentinel was able to maintain its large statistical differences throughout the evaluated versions.

Therefore, we can state that, in overall, Sentinel strategies can be reused without great degradation in effectiveness when compared to conventional strategies, at least when considering the 3 subsequent versions of the programs used in this work. This is supported by the 20 system comparisons (10 systems with 2 indicators), where Sentinel obtained consistent better or equivalent results for 19 of those comparisons (95%). Thus, these strategies can be used in

Table 4.6: RQ2: Effect size results for the comparison.

Program	HV			IGD		
	SM	RMS	ROS	SM	RMS	ROS
beanutils-1.8.0	**0.78 (L)**	**0.94 (L)**	**0.84 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
beanutils-1.8.1	0.63 (S)	**0.96 (L)**	**0.80 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
beanutils-1.8.2	0.70 (M)	**0.88 (L)**	**0.77 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
beanutils-1.8.3	0.64 (S)	**0.91 (L)**	**0.80 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
codec-1.4	**0.88 (L)**	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
codec-1.5	0.69 (M)	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
codec-1.6	0.45 (N)	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
codec-1.11	**0.88 (L)**	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
collections-3.0	**1 (L)**	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
collections-3.1	**1 (L)**	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
collections-3.2	**1 (L)**	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
collections-3.2.1	**1 (L)**	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
lang-3.0	**1 (L)**	**0.99 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
lang-3.0.1	**1 (L)**	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
lang-3.1	**1 (L)**	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
lang-3.2	**1 (L)**	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
validator-1.4.0	**1 (L)**	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
validator-1.4.1	**1 (L)**	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
validator-1.5.0	**1 (L)**	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
validator-1.5.1	**1 (L)**	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
jfreechart-1.0.0	**1 (L)**	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
jfreechart-1.0.1	**1 (L)**	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
jfreechart-1.0.2	**1 (L)**	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
jfreechart-1.0.3	**1 (L)**	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
jgrapht-0.9.0	**0.99 (L)**	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
jgrapht-0.9.1	**0.97 (L)**	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
jgrapht-0.9.2	**0.95 (L)**	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
jgrapht-1.0.0	**0.92 (L)**	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
joda-time-2.8	**1 (L)**	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
joda-time-2.8.1	**1 (L)**	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
joda-time-2.8.2	**1 (L)**	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
joda-time-2.9	**1 (L)**	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
ognl-3.1	**0.99 (L)**	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
ognl-3.1.1	**1 (L)**	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
ognl-3.1.2	**1 (L)**	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
ognl-3.1.3	**1 (L)**	**1 (L)**	**1 (L)**	**0 (L)**	**0 (L)**	**0 (L)**
wire-2.0.0	0.03 (L)	**1 (L)**	0.57 (N)	**0 (L)**	**0 (L)**	**0 (L)**
wire-2.0.1	0.04 (L)	**1 (L)**	0.40 (S)	**0 (L)**	**0 (L)**	**0 (L)**
wire-2.0.2	0.05 (L)	**1 (L)**	0.45 (N)	**0 (L)**	**0 (L)**	**0 (L)**
wire-2.0.3	0 (L)	**1 (L)**	0.19 (L)	**0 (L)**	**0 (L)**	**0 (L)**

subsequent releases (without need to retrain) providing a significantly better mutation score than conventional strategies.

RQ3: We can reject the null hypothesis of RQ3 and assert that the strategies generated by Sentinel can be reused in newer versions of the software in which Sentinel was trained without becoming ineffective.

4.5 Discussion

As presented in the previous section, Sentinel generates strategies that are better than conventional ones in the literature, thus they are indeed capable of reducing the mutation cost while maintaining the mutation score in a better way. However, there is the caveat related to the training cost of Sentinel. While Sentinel demands a training phase for generating strategies, the conventional strategies are already well-defined in the literature and may not require training if the tester chooses to use them. In this section we evaluate and discuss the results of Sentinel taking its training time into consideration. Table 4.7 presents the average execution times of (respectively) Sentinel training, conventional mutation, Sentinel strategies and conventional strategies considering all program versions.

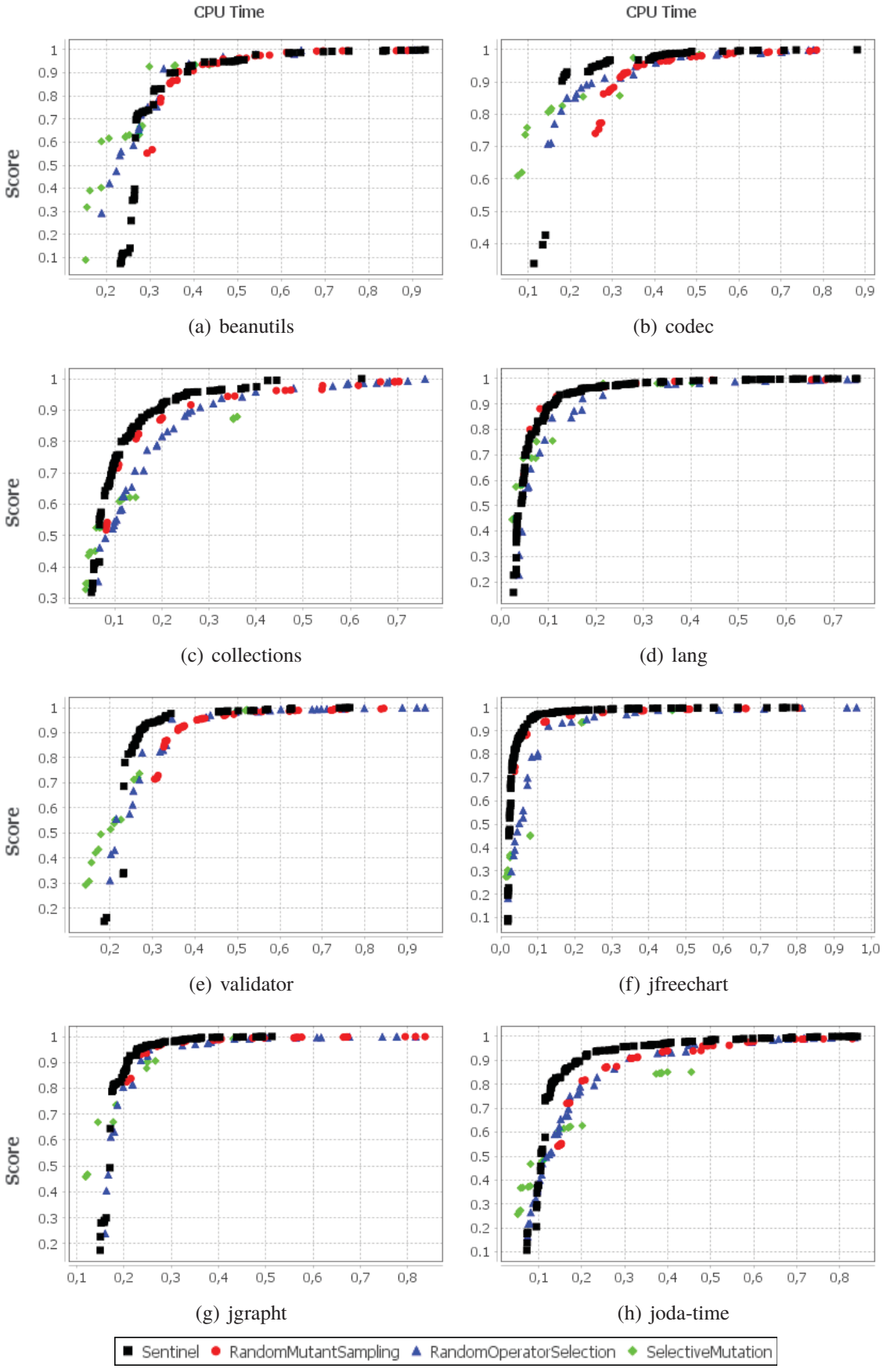
Table 4.7: RQ3: Average execution time of Sentinel and the Conventional Strategies.

Program	Training	All Mut.	Sentinel	RMS	ROS	SM
beanutils	04h12m23s	10m27s	05m14s	05m47s	05m20s	02m54s
codec	01h22m11s	14m05s	05m17s	06m34s	05m41s	03m39s
collections	07h24m22s	26m12s	05m01s	08m46s	08m33s	03m16s
lang	11h23m49s	50m34s	10m43s	17m19s	16m26s	06m05s
validator	38m44s	07m59s	03m04s	04m17s	03m56s	02m18s
jfreechart	07h51m33s	01h57m08s	18m24s	34m04s	32m05s	13m48s
jgrapht	01h49m51s	34m26s	09m23s	16m00s	14m53s	08m06s
joda-time	03h25m33s	01h02m39s	23m08s	25m16s	21m43s	09m55s
ognl	01h19m44s	10m33s	37s	03m26s	03m21s	01m05s
wire	01m41s	58s	29s	41s	35s	28s

As shown in Table 4.7, the training time of Sentinel may seem unsatisfactory at first glance. However, our results show that once a set of strategies is generated with Sentinel, their execution cost and mutation score are at least as same as the conventional strategies, and more often than not, the generated strategies are better. When considering both objectives, Sentinel is the best option, i.e., Sentinel presents the best trade-off between mutation score and execution time. Hence, in the long run, Sentinel training time can be justified by the significantly better execution times and mutation scores.

This trade-off can be seen in the Pareto fronts depicted in Figures 4.1 and 4.2. Each point in the scatter plot is a non-dominated strategy found over the 30 independent runs and it shows where it stands in the objective space. For succinctness, we only show the fronts for the first version of each program.

The Pareto fronts and the indicator analysis of the previous subsection show that, in overall, choosing a strategy generated by Sentinel yields the best results. In a scenario where the tester must achieve a minimum score of x, they can select the non-dominated strategy generated by Sentinel that obtains at least x and with the lowest CPU time objective. By using such a strategy, we can expect that it will provide a set of mutants that is faster to execute than SM, RMS and ROS strategies that achieve at least the x mutation score. For example, considering the set of strategies obtained in Figure 4.1(b) for codec, if the tester chooses 0.9 as minimum mutation score for their mutation testing process, then using a strategy generated by Sentinel would result in a mutant set that takes approximately 18% of the CPU time that it would take to execute all mutants. Respectively, the CPU time in the same scenario for SM is approximately 35%, 28% for ROS and 32% for RMS. This can be better seen in Figure 4.3, for which the relevant part of the scatter plot is highlighted.



(a) beanutils (b) codec

(c) collections (d) lang

(e) validator (f) jfreechart

(g) jgrapht (h) joda-time

Figure 4.1: Pareto Fronts. Part 1

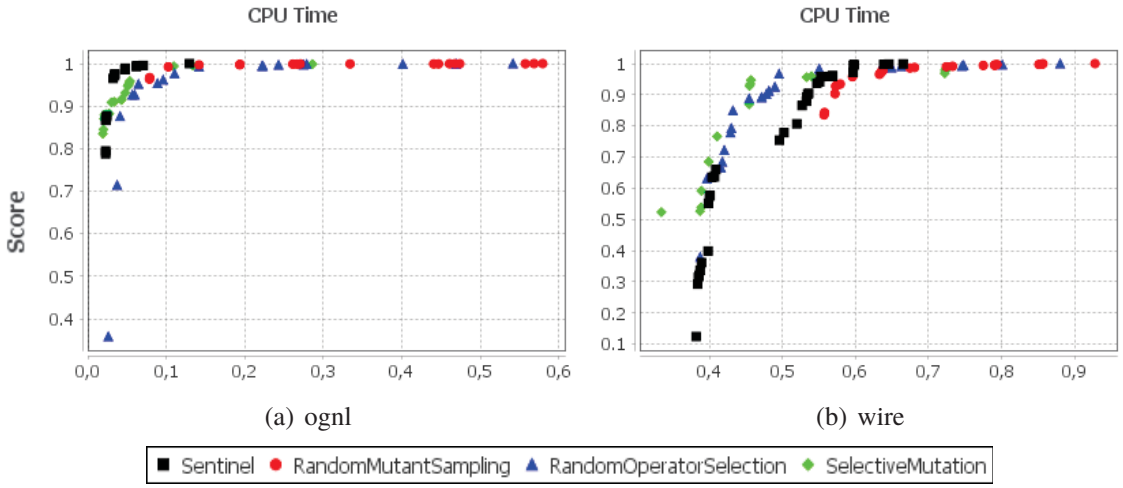


Figure 4.2: Pareto Fronts. Part 2

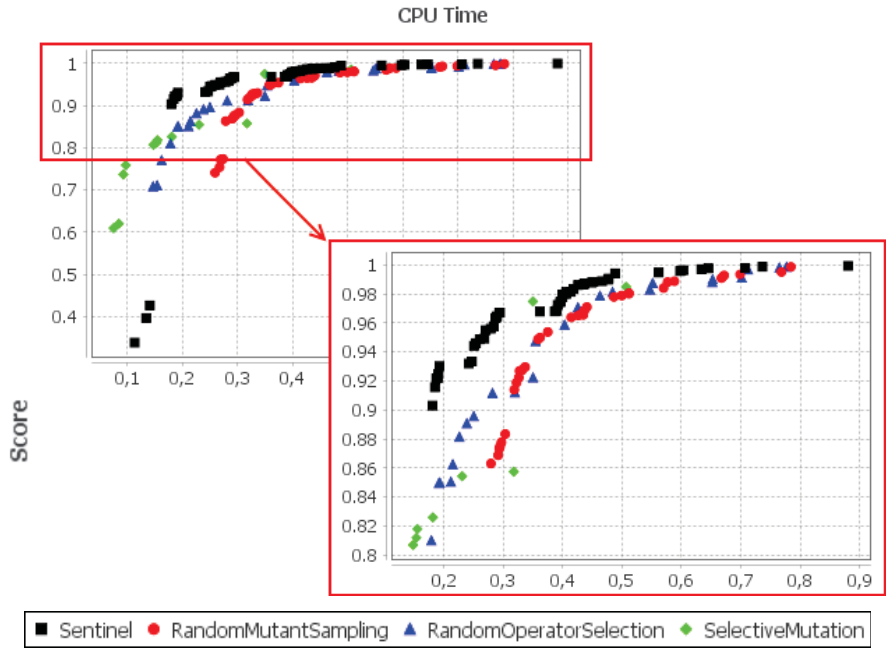


Figure 4.3: Highlighted objectives for codec

In other words, when using a strategy generated by Sentinel, there is a greater chance of this strategy yielding a higher mutation score for the same CPU time than conventional strategies. In this sense, the cost associated to the training phase of Sentinel is a price to pay for significantly better results in both mutation score and mutant execution times.

One should note that there is no training cost regarding the use of a conventional strategy only when the tester already knows which strategy to use or when the choice is arbitrary. If the tester needs to select and configure the best conventional strategy, then they will need to perform experiments to compare the several types of strategies and configurations. The actual cost of such task was not measured in this work, but given the time taken to execute the experiments with those strategies for answering RQ2 and RQ3, we believe that it can be as costly as the training time of Sentinel when we also consider the time taken by the tester to set up the experiments, collect the data and then perform the analysis. All in all, doing it automatically seems preferable

in most cases. However, there is no other work in the literature that addresses this configuration automatically, thus Sentinel can be used as a tool to this end.

In the long run, as far as we could observe in the experiments, Sentinel has a good cost-benefit and the generated strategies hold their good results. Taking into account that Sentinel training time could be further reduced by the use of parallelism and other techniques [25, 36, 37, 40, 92] for which open-source framework are available [93], we believe it is an affordable cost to pay off in order to obtain the best results.

Nevertheless, Sentinel is best applied on systems for which the mutation testing time is too high and when the reliability of the test must be ensured. Possible examples of such systems are JFreeChart, Joda-Time and Apache Commons Lang that take dozens of minutes to execute all mutants and that are modified by several contributors hundreds of times. For cases like these, reducing even a small percentage of CPU time can be impacting for the development cost and maintaining a great mutation score is crucial. If the tester wants to save time and maintain the mutation score at the same time, then Sentinel can generate strategies that are tailored specially for their system and that will be better than conventional ones.

Therefore, we advocate that Sentinel should be used in two situations:

1. When the time taken to execute all mutants is infeasible; and
2. When the mutation testing is performed several times during the software development process.

Indeed, how long is "infeasible time" and how many is "several times" is totally subjective to environmental constraints, software testing budget and even to the required reliability level involved in the testing process. In any case, if the tester finds themselves in at least one of those two situations, then we believe that the training cost of Sentinel can be amortised in the long run by the cost reduction obtained by the generated strategies and by the significantly better mutation scores. Moreover, given the fact that the test suites being evaluated by the reduced mutant sets are usually reused in the newer versions of the software, a good mutation activity is crucial for the reliability of the code, thus using a strategy trained for that purpose is the best option.

4.6 Examples of Generated Strategies

Sentinel is able to generate strategies using operations from several existing conventional strategies. In fact, the generated strategies can be either hybrid containing several kinds of operations, or even very simple by doing one random retention like RMS. In this section we present some examples of strategies generated by Sentinel to glimpse on how different those strategies are from conventional ones. We do not intend to do a thorough assessment of which operations are more meaningful for the good results, nor evaluate which ones appear more often in the strategies, but rather report some functionalities found among them and how they are performed. For this end, we randomly selected 3 strategies generated by Sentinel in the experiments presented in this chapter.

The first selected strategy was generated using jgrapht-0.9.0. It is the 26^{th} best strategy found for that system in terms of mutation score with an average of 0.9764, obtaining such a score using only 31.62% of the CPU time it takes to execute all mutants for the same system. This strategy first executes all operators to generate all mutants. Then it discards 50% of the mutants randomly. After that, it performs the first grouping operation according to the type of operator used to generate them and then does a second level grouping based on the specific

operator used to generate each mutant. Then it sorts the second level groups according to the number of mutants. It then discards 90% of the mutants in the smallest second level group of each first level group. For jgrapht-0.9.0, this strategy first randomly discards 1,488 mutants out of 2,976 total, then discards approximately 349 mutants (it may vary due to the randomness of the strategy), leaving approximately 1,139 mutants (38.27% of the total) in the end to perform the mutation activity.

The second selected strategy was generated using commons-validator-1.4.0. It is the 28^{th} best strategy for that system in terms of mutation score with an average of 0.9791 score with only 50.52% of CPU time. The first step of this strategy is to execute the first two operators in the list of operators. With PIT, the selected operators are Conditionals Boundary Mutator and Negate Conditionals Mutator. Then, the strategy randomly retains 30% of the generated mutants. The final step is to execute the last two operators of the list of available operators, which for PIT are Return Values Mutator and Void Method Call Mutator, and then it stores all the mutants recently generated alongside the 30% retained mutants. For commons-validator-1.4.0, the strategy first generates 929 mutants, retains 278 mutants, and then store those mutants with other 660 mutants generated by Return Values Mutator and Void Method Call Mutator. In the end, the reduced set of mutants contains 938 mutants out of 1,811 total (51.79% of all mutants).

The last analysed strategy was generated using commons-lang-3.0, which is the 64^{th} best strategy for that system in terms of mutations score with an average score of 0.9184 using 14.88% of CPU time to achieve that. The first step of this strategy is to discard the first half of operators from the list of all operators, which for PIT are Conditionals Boundary Mutator, Negate Conditionals Mutator and Math Mutator. Then, it executes the first operator from the remaining operators sorted by type (which for PIT is Return Values Mutator) and stores the generated mutants in the final set of reduced mutants. Then, it executes all remaining available operators (Increments Mutator, Invert Negatives Mutator, Return Values Mutator and Void Method Call Mutator) and discards 80% of the recently generated mutants randomly. In the experiments, first the strategy generates and stores 3,001 mutants using Return Values Mutator. Then the strategy generates 4,116 mutants using Increments Mutator, Invert Negatives Mutator, Return Values Mutator and Void Method Call Mutator, and randomly discards 3,292 mutants. Then it merges the remaining 824 mutants to the list of 3,001 mutants previously generated, excluding repeated ones. On average, the resulting reduced set of mutants has approximately 3,227 mutants out of 9,072 possible mutants for the system (35.57% of all mutants).

All three strategies are hybrid by mixing operations from different types of strategies. The first one uses mutant sampling and clustering of mutants based on their attributes. The second and third strategies use operator selection and mutant sampling techniques. It is interesting to note that, for the analysed strategies, the selection of operators is deterministic, implying that the strategies are trained to execute specific operators. Presumably, these operators are the most suitable ones for the system used in the training process. Moreover, the sampling of mutants is usually done at random, but with other operations involved in the process as well, such as grouping mutants and randomly selecting a given percentage from each group (first strategy), and discarding mutants generated by specific operators (third strategy).

Furthermore, we observed once again that the number of mutants does not always reflect the CPU time spent to perform the mutation. For the first strategy the average CPU time is 31.62% using on average 38.27% of mutants. For the second strategy this gap is smaller, having an average CPU time of 50.52% when using 51.79% of all mutants. The biggest gap between these two metrics was observed when executing the third strategy, where an average CPU time of 14.88% was obtained when executing a set containing on average 35.57% of all mutants. These

strategies are trained to find a set of mutants containing the fastest mutants to execute, thus the execution time usually is lower than the number of mutants.

In light of the findings of this work, it seems that hybrid strategies seem to be the best choice for reducing the number of mutants. This conclusion aligns with the findings of Zhang et al. [105], who observed that combining strategies is better than using them separately.

4.7 Threats to Validity

The threats to validity are divided into categories according to the framework proposed by Wohlin et al. [101].

Conclusion Validity

For answering RQ2 we trained Sentinel on the first version of a program and tested the generated strategies on the same versions and 3 subsequent versions. Training and testing an approach on the same version may lead to overfitting. However, when using Sentinel, the tester might actually train Sentinel on a version of their software and use the generated strategies while developing and testing that same version. Hence, this is not an impossible scenario. Moreover, if we exclude the training instances from the testing, then Sentinel still has better or equal results for 57 out of 60 statistical comparisons (Tables 4.4-4.5) and for 167 out of 180 effect size comparisons (Table 4.6), leading to the same conclusion.

We have not compared the cost of training Sentinel with the cost of executing and comparing conventional strategies manually. As mentioned before, this comparison has a cost related to the strategies execution and the manual effort employed in the experimentation. Considering this time could provide more insights on the actual computational cost and the human effort employed in selecting and configuring strategies.

Internal Validity

The test cases used in the experiments are provided alongside the source code of the systems. We have not tried to evaluate equivalent mutants and to achieve 1.0 mutation score before performing the experiments. However, we are not trying to compare the mutation score of the strategies considering a perfect global mutation score, but rather how similar their mutation score are to the actual mutation score achieved by the available test sets and generated mutants.

Because Sentinel uses PIT, then the results of the experiments depend on the configuration of PIT and its internal features, such as test case prioritization, mutation operator execution and so on. In this sense, using different configurations for PIT and even other mutation tools may lead to different results. Moreover, PIT only uses 7 operators as default and the results are also dependent on those operators. However, if the configuration of PIT changes or another mutation tool is used, then the strategies generated by Sentinel will be trained on this new scenario, thus we expect the same good results.

We have not tuned the GE algorithm, but rather we used some common parameters used in other works of the literature [45, 50, 76, 77] for the execution. A GE tuning can lead to even better results.

External Validity

We tested Sentinel in 10 different programs with 4 different versions of each. Even though the total number of system versions evaluated (40) is similar to the number of systems used in other works of the literature [42, 63, 70, 104], we cannot assert that this is enough to generalize the results to all systems. Furthermore, the size of the programs may not reflect the size of all real world systems. To minimize this threat, we tried to evaluate systems of several sizes and domains, and we used systems extracted from other works of the literature.

4.8 Final Remarks

In this chapter we presented the experiments conducted to evaluate Sentinel. The three RQs were formulated to answer if Sentinel is better than a random hyper-heuristic to generate strategies, if the generated strategies are better or equal to conventional strategies of the literature, and if the strategies can be successfully reused in newer versions of the software.

The experiments encompassed 10 systems and 4 different versions of each. We used two quality indicators and two statistical tests to evaluate the results. The strategies were compared to three conventional strategies of the literature. After analysing the data, we were able to positively answer all three questions. If the tester is willing to spare some computational resources during the training phase of Sentinel, then Sentinel is the best choice.

The main observed advantage of Sentinel is that it automates the selection and configuration of the best mutant reduction strategies for a given system. By letting Sentinel generate good strategies, the tester does not need to waste their time in selecting and configuring the best strategy. Furthermore, optimizing the processes of the software development cycle instead of specific problems is also a trend in SBSE [55] that Sentinel is contributing to.

In this sense, we can accept the main hypothesis presented in the introduction of this work: “An approach based on hyper-heuristics is capable of generating strategies that contribute to reduce the cost of the mutation testing activity without losing efficacy when compared to conventional strategies from the literature”.

Chapter 5

Related Work

This chapter presents the works related to the objective of this work. We present works on hyper-heuristics applied to SBSE, which are the most similar. The main goal of analysing the works presented in this chapter is to identify search based techniques and how hyper-heuristics can be applied to SBSE.

The search for works was done using the following online libraries: IEEE Xplore Digital Library[1], ACM Digital Library[2], Elsevier[3], Springer[4] and Scopus[5]. The search was performed on October 2018 using the following search string:

> (“hyper-heuristic” OR “hyper-heuristics” OR “hyper heuristic” OR “hyper heuristics” OR “hyperheuristic” OR “hyperheuristics”)
>
> AND
>
> (“software engineering”)

The search string was limited to the title, keywords and abstract fields for more concise results. The resulting papers were evaluated and their references checked using the snowballing technique. Furthermore, we have found a few surveys and literature reviews which included related work and also contributed for the development of this chapter [33, 51, 53, 55, 78, 95].

We only included in the results the works that actually propose or evaluate hyper-heuristics for solving software engineering problems, and discarded works that only mentioned hyper-heuristics (e.g. subject of future work and trend surveys). In the end, we have narrowed the results down to 11 different works, which are reported in 19 papers due to extensions and slight variations. These papers are presented in the next section.

5.1 Hyper-Heuristic Works in SBSE

This section is divided by subfield for which the hyper-heuristics are proposed.

5.1.1 Software Project Management

Basgalupp et al. [8] proposed an off-line hyper-heuristic for evolving algorithms that can create decision trees. These trees are then applied to predict software effort during the maintenance

[1] http://ieeexplore.ieee.org/
[2] http://dl.acm.org/
[3] http://www.sciencedirect.com/
[4] http://www.springer.com/br/
[5] http://www.scopus.com/

phase, because effort generates cost and this cost is a subject of interest for stakeholders. Furthermore, with an accurate prediction metric, it is possible to better allocate resources during the software development and maintenance, which reduces the waste of efforts and consequently the overall cost of the software. The proposed hyper-heuristic uses induction code blocks from existing algorithms and builds new algorithms that are submitted to an evolutionary process. In the experiments, the authors observed that the hyper-heuristic generated algorithms capable of obtaining better results than conventional evolutionary algorithms and specialized algorithms that create decision trees.

Sarro et al. [94] propose an adaptive multi-objective approach to select meta-heuristic operators for solving the overtime problem. In such a problem, the goal is to minimize the project duration, risk of overrun and the overtime deployed. The authors tested the proposed hyper-heuristic in 8 real world systems and found out that it outperforms the state of the art algorithms in 93 percent of the experiments with large effect size. Furthermore, when compared to common overtime planning practices, the approach was able to obtain the best results in 100 percent of the experiments with large statistical differences.

5.1.2 Software Design

Kumari e Srinivas [68, 69] proposed an on-line hyper-heuristic based on MOEAs to solve the software modules clustering problem. By clustering the modules in a certain way, the engineer can reduce the software maintenance cost, ease the software understanding and ease the software development. However, given the number of modules and the way they can be clustered, finding the best clustering is a hard task. The hyper-heuristic is called Multi-objective Hyper-heuristic Evolutionary Algorithm (MHypEA) and uses low-level heuristics composed by selection, mutation and crossover operators for the problem in hand. The objective of MHypEA is to maximize the modules cohesion and minimize their coupling. The authors executed MHypEA in six systems [68] and then in 12 systems [69], obtaining good results with one-twentieth of the computational effort needed by other MOEAs in overall.

5.1.3 Software Testing

Jia et al. [61] investigate the use of an off-line hyper-heuristic based on simulated annealing for the selection of combinatorial testing strategies. The authors idea is to provide, based on several problem instances, a single generic approach that is good for the majority of the systems. For that, the hyper-heuristic uses machine learning in runtime and applies the learning to obtain the best combinatorial testing strategy. The experiments done in 26 real world systems prove that the hyper-heuristic is capable of obtaining the best results when compared to the best strategies in the literature.

Jia [60] proposes an approach based on on-line hyper-heuristics for Search Based Software Testing (SBST). According to the author, the idea is to detach the hyper-heuristic from the software detail and apply it independently to the search algorithm. The hyper-heuristic employs an intermediary layer with learning for the decision of which low-level heuristics must be used. The low-level heuristics can be problem specific and can be used by meta-heuristics. The author did not present experiments, just an example of how the hyper-heuristic would be applied to combinatorial testing problem.

Guizzo et al. [43, 45, 48–50] propose *Hyper-Heuristic for the Integration and Test Order Problem* (HITO). HITO is used for the on-line selection of mutation and crossover operators during the MOEA execution for solving the integration and test order problem. This problem

consists in finding the best order of units to be tested and integrated into the system, such that the stubs development cost is minimized. A stub is needed when a unit A depends on a unit B, but B has not yet been tested and integrated into the software. In this case, a stub must be created to emulate the behaviour of B, but this comes with a development cost that can be minimized with search algorithms. In their experiments, the authors used NSGA-II and SPEA2 as MOEAs for HITO and the selection methods Choice Funcion (CF) [74] e Multi-Armed Bandit (MAB) [38]. These selection methods use the current heuristic performance to maximize the convergence, but also consider heuristics that were not used in a while to also introduce diversity in the heuristic selection process. HITO was tested in 8 real world systems and the results were better or equal to the conventional algorithms NSGA-II, SPEA2 and MOEA/DD. In addition to the good results, the advantage of using HITO is that the tester does not need to select the best operators and neither their parameters.

Carvalho et al. [13] proposed a hyper-heuristic called MOCAITO-HH, used for the on-line selection of MOEAs for solving the same problem. Differently to HITO, MOCAITO-HH considers the MOEAs as low-level heuristics and dynamically selects them during the optimization process. The algorithms are selected according to the their results using CF. In the MOCAITO-HH implementation, the MOEAs are evaluated with quality indicators and are switched during the evolutionary process. The experiments show that MOCAITO-HH obtained the best results when compared to conventional MOEAs and competitive results to HITO. Another advantage is that the tester does not need to select the MOEA to be used in each optimization.

Still for solving the same problem, Mariani et al. [76, 77] used GE for the generation of MOEAs. The hyper-heuristic evolves a set of MOEAs and uses a grammar to build new MOEAs based on the best MOEAs of the current generation. The proposed grammar contains several components and parameters common to MOEAs in the literature. These components and parameters are selected during the training phase and, in the end, the best MOEA is returned. Thus, the tester should only inform the available components and parameters and the hyper-heuristic will generate a MOEA that can be reused. This hyper-heuristic is specially useful for testers that do not have comprehensive optimization knowledge, because avoids the manual design and configuration a MOEA. In the experimental evaluation the authors executed the hyper-heuristic in 7 real world problems and compared to NSGA-II, SPEA2 and HITO. The hyper-heuristic was capable of generating better results than the conventional algorithms and obtained competitive results to HITO.

The works of Ferreira et al. [34, 35] and Strickler et al. [98] use online hyper-heuristics based on HITO. Such hyper-heuristics are applied for selecting products of Software Product Lines (SPLs) to be used as test cases during the mutation testing of feature models. During the empirical evaluation, the authors obtained better or competitive results than conventional MOEAs such as MOEA/D-DRA, NSGA-II, IBEA and SPEA2.

Jakubovski Filho et al. [58] also applied a hyper-heuristic for the problem of SPL product selection. In their work, the authors used an offline hyper-heuristic based on GE (proposed by Mariani et al. [76, 77]) to automatically generate MOEAs. The generated MOEAs were compared to NSGA-II and to HITO. The results are competitive to HITO and generally better than NSGA-II.

Within the mutation testing subject, Sentinel differs from [34, 35, 58, 98] on the artefact that is being mutated. While the related work is focused on the mutation of feature models for SPLs, Sentinel is applied to the mutation of Java code. Furthermore, Sentinel actually reduces the set of mutants, whereas their work aims at selecting a subset of test cases. While Sentinel is an off-line generation hyper-heuristic, the hyper-heuristic used in the related work is an on-line selection hyper-heuristic for adaptively selecting evolutionary operators. Only [58] is similar to

Sentinel in the sense of off-line generation of heuristics for mutation testing, but still their work focuses on generating MOEAs instead of mutant reduction strategies.

Lima and Vergilio [72] used HITO to automatically select crossover and mutation operators of a MOEA in order to generate HOMs. The algorithm tries to optimize three objectives: i) minimizing the total number of generated HOMs; ii) maximizing the number of revealed subtle faults (i.e. faults that can only be revealed by the HOM); and iii) maximizing the number of strongly subsuming HOMs (i.e. HOMs that can substitute their constituent mutants). The results showed competitive results to NSGA-II, but with the advantage of avoiding the manual selection of operators when using HITO. This work differs from Sentinel because it focuses on HOMs, whereas Sentinel only applies mutant reduction on FOMs. Furthermore, the objectives used by Lima and Vergilio [72] are concerned with HOM related metrics, such as number of subsuming HOMs and number of revealed subtle faults.

5.2 Final Remarks

15 out of 19 works found are applied to software testing. This majority of works focusing on testing can be explained by the great cost associated to such a phase. Furthermore, the majority of works use on-line hyper-heuristics and hyper-heuristic feedback with learning. Another important observation is that the interest in applying hyper-heuristics in SBSE is rather recent, with the oldest work being published in 2013.

The works mentioned in this subsection present promising results, as almost all of them presented the best results when compared to conventional algorithms from the literature. This indicates that hyper-heuristics can be very effective in SBSE. As shown in the previous chapter, mutant reduction strategies and hyper-heuristics can be explored together in order to minimize the overall mutation cost. However, we have not found any work on the selection or generation of any kind of cost reduction strategy. Hence, as far as we are aware, Sentinel is the first effort on automating the selection and configuration of mutant reduction strategies. The good results of Sentinel can be seen as yet another evidence on the benefits of hyper-heuristics in SBSE.

Chapter 6

Conclusion

This work presented Sentinel, an off-line hyper-heuristic based approach for generating mutant reduction strategies for mutation testing. The main motivation for proposing and implementing Sentinel is to automate the selection and configuration of mutant reduction strategies. The objective is to generate strategies that can reduce the mutation execution time, while maintaining the mutation score.

Sentinel uses a GE algorithm to build strategies using operations from existing strategies from the literature. In this sense, we proposed a context-free grammar containing several rules for supporting the strategy generation. This grammar is used during the multi-objective evolutionary process for mapping a solution into a mutant reduction strategy. Each strategy is evaluated by two objective functions: CPU time taken to be executed along with its mutant execution, and global mutation score. The idea is that, after a strategy is generated, it can be reused every time the tester needs to perform the mutation testing activity without the need of generating and executing all mutants.

For assessing the feasibility of Sentinel, multiple experiments were performed. The experiments were conducted using 10 different systems with 4 versions of each, in a total of 40 systems to answer three research questions.

The first research question is concerned with the actual efficacy of the hyper-heuristic implemented in Sentinel. For that purpose we compared Sentinel to a random hyper-heuristic and concluded that Sentinel is in fact better than the random algorithm.

The second research question is designed to assess if the generated strategies are better or equivalent to conventional strategies from the literature. We compared the generated strategies with Random Mutant Sampling, Selective Mutation and Random Operator Selection. Using hypervolume, IGD, Kruskal-Wallis and Vargha-Delaney A_{12} effect size we observed that Sentinel obtained the best results or results statistically equivalent to the best ones for 76 out of 80 (95%) indicator comparisons. Furthermore, Sentinel obtained 227 out of 240 (~95%) favourable large effect size results. In this sense, we can conclude that Sentinel indeed generates the best strategies when compared to those conventional strategies.

Finally, the third question regards to the reusability of strategies generated by Sentinel. We trained Sentinel on a version of the software and tested the strategies on subsequent versions of the same software. After analysing the data, we observed that for 7 out of the 10 systems, the generated strategies are statistically better than the conventional ones in all versions and for both indicators, and at least equivalent for 9 out of 10 versions. Considering the 20 indicator comparisons, Sentinel presented consistent equivalent or better results for the subsequent versions in 19 out of 20 (95%) comparisons. This indicates that the generated strategies stay effective and

actually keep outperforming the other strategies as the software evolves. Therefore, the strategies can be reused without needing to retrain Sentinel for several versions.

The main advantage of Sentinel is that it can remove the manually intensive and boring task of selecting and configuring strategies. Sentinel automates the experimentation process which the tester would have to perform manually to find out which is the best strategy. In this way, the tester can focus their expertise on actually testing the software and not on optimization. Furthermore, as we observed in the experimentation, the generated strategies are better than common strategies from the literature such as Selective Mutation and Random Mutant Sampling. These generated strategies can also be reused for several new versions of the software, without losing effectiveness, which in the long run is cost-efficient considering the training time.

Therefore, for the cost of a training phase, the tester can automatically obtain a set of good strategies that can be reused and that can significantly reduce the cost of the mutants execution while maintaining the mutation score. This is even more beneficial if the mutation testing is executed multiple times throughout the development of a program version, or if new versions are being constantly released. In light of this, we advocate that Sentinel should be used when the time taken to execute all mutants is too high, when the mutation testing is performed several times during the development process or specially when the mutation score must be maintained. In such cases, the significantly better results obtained by the strategies generated by Sentinel will be more meaningful.

Considering the evaluation done and the answers found for the research questions, we can accept the main hypothesis of this work presented in the introduction. Sentinel, an approach based on hyper-heuristics, is indeed capable of generating strategies that contribute to reduce the cost of the mutation testing activity without losing efficacy in terms of mutation score when compared to conventional strategies from the literature.

Finally, the main contributions of this work can be summarized as follows:

- Investigation of a multi-objective hyper-heuristic approach for solving a problem not yet solved by hyper-heuristics;
- Automation of the selection and configuration of mutant reduction strategies;
- Generation of strategies that can minimize the execution time and maximize the mutation score;
- Usage of a multi-objective evolutionary algorithm with an objective not yet explored by other works: execution time;
- Proposal of a context-free grammar with multiple features of several conventional strategies to support the generation of unforeseen strategies and to allow the automatic configuration of existing ones;
- Implementation and provision of Sentinel as an open-source software;
- Experimental evaluation and comparison of Sentinel with conventional strategies of the literature in real-world open-source software.

6.1 Limitations

This section presents the limitations of the proposed approach. These limitations will be addressed in future work.

The first observed limitation is the training cost of Sentinel. As it happens in off-line hyper-heuristics [10], there is a training phase for Sentinel to generate reusable strategies. Given the fact that the results in the literature using conventional strategies are reportedly good (as discussed in Section 2.4.2), the cost of the training phase of Sentinel must be considered by the tester and it might actually prevent the usage of the proposed approach.

We have implemented only one version of the context-free grammar that is used with the GE algorithm. We have not explored other grammars or other variations of the same grammar. Furthermore, the grammar is limited to operations from 3 types of strategies: Selective Mutation, Mutant Clustering and Mutant Sampling. Other conventional strategies might be evaluated and other operations may be added to the generation of strategies.

The only hyper-heuristic used by Sentinel is a variation of a GE algorithm with predefined parameters. We have not implemented other grammar-based algorithms to serve as hyper-heuristic during the generation of strategies, even though there are other types in the literature [17, 86] and other possible parameters for them.

Sentinel is only applied to the unit test of Java programs so far. For using it with other programming languages, other testing tools and other testing scenarios, then new integration components must be implemented for it to properly work. In addition, the effectiveness of Sentinel in such cases is unclear, since we have not performed experiments for the assessment.

Sentinel proposes a solution for a problem related to the manual selection and configuration of strategies for reducing the number of mutants. We are not sure if mutation testing is used during the development of big systems such as the ones used in the experiments, let alone mutant reduction. Furthermore, it is not clear how many times the mutation testing activity is performed if it is ever used by the tester in such systems. The tester might execute the mutation only once and then reuse the test cases. In this sense, if the mutation is rarely applied or if the mutation testing is not an option to the tester in the first place, then Sentinel is also not an option.

6.2 Future Work

As future work, we intend to expand the operations used by Sentinel and extend the grammar to encompass such operations. More operations such as grouping, sorting and execution types can be added as rules to the grammar to improve the results and flexibility of Sentinel. An interesting investigation would be to enable the generation of search based strategies, such as [58, 76, 77] for generating MOEAs, or even hybrid strategies with search based and simpler heuristics. All of this can be done by extending the grammar to work with such operations and strategies, and then by implementing the extracted operations in Sentinel. This investigation can provide more insights on the how different kinds of operations can interact to provide good results.

Additionally, other objectives for generating strategies can be used, such as number of mutants, number of equivalent mutants, number of revealed faults, and so on. This can also reduce the training phase, given the fact that the great cost of this phase is due to the `TIME` objective evaluation.

Generated strategies can be reused in newer versions of the same software, but it is still unclear if they are effective in other software. For evaluating that, we intend cross validate the strategies in several programs and assess their reusability. We believe that strategies trained on a given system may be reused on different systems of the same domain or even systems similar in some specific features. Machine Learning algorithms can be used to assess the similarity of the systems and the cross validation can be guided according to this assessment.

We intend to evaluate if the number of mutants is actually a good surrogate for the cost metric of mutation testing. As far as we are aware, our work is the only one that evaluates the

cost reduction of mutant reduction strategies in terms of CPU time instead of number of selected mutants. The reason behind that is probably the great cost of the experimentation process. However, in preliminary experiments during the development of Sentinel, we observed that some mutants take more time to execute than a set of several other mutants. By comparing the correlation between mutation score and number of mutants and the correlation between mutation score and execution time, we can assess if the number of mutants metric actually reflects cost reduction. Perhaps it can be used only to reduce the cost of the training phase.

We did not explicitly investigate the use of "do faster" and "do smarter" techniques in this work. PIT already employs some cost reduction strategies, but we did no compare how they interact with Sentinel. In future work, we intend to combine such strategies with the automatic generation implemented in Sentinel and assess how further we can reduce the cost. Strategies based on Machine Learning algorithms (such as in [104]) can significantly improve the cost reduction and maybe they can be used as a complement to mutant reduction strategies.

An interesting approach would be to extend Sentinel for the generation of HOMs. In this case, HOM specific objectives can be used to search for specific kinds of HOMs and tackle difficulties in this type of mutation.

There are other mutation tools that can be used with Sentinel, and consequently other mutation operators. Depending on the tool integrated with Sentinel, other artefacts can be mutated and Sentinel can generate strategies for other programming languages and other testing scenarios as well. For instance, we can evaluate Sentinel for the mutation testing of C programs, for mutating software models and also for integration testing instead of unit testing.

Finally, we intend to compute the execution time and the manual effort of creating, selecting and/or configuring strategies manually. This effort can be measured and compared to the training cost of Sentinel to assess at to which extent the usage of an automatic tool for this task is cost saving. We believe that by automating such a task, the tester can actually spend less time doing optimization and experimentations manually, and let Sentinel do that hard work while they test the software.

Bibliography

[1] Acree, A. T. (1980). *On Mutation*. Phd thesis, Georgia Institute of Technology.

[2] Adamopoulos, K., Harman, M., and Hierons, R. M. (2004). How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *Proceedings of the 13th Genetic and Evolutionary Computation Conference*, pages 1338–1349.

[3] Ammann, P. and Offutt, J. (2008). *Introduction to Software Testing*. Cambridge University Press, 1st edition.

[4] Arcuri, A. and Briand, L. (2014). A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250.

[5] Banzi, A. S., Nobre, T., Pinheiro, G. B., Árias, J. a. C. G., Pozo, A., and Vergilio, S. R. (2012). Selecting Mutation Operators with a Multiobjective Approach. *Expert Systems with Applications*, 39(15):12131–12142.

[6] Barbosa, E. F., Maldonado, J. C., and Vincenzi, A. M. R. (2001). Toward the determination of sufficient mutant operators for C. *Software Testing, Verification and Reliability*, 11(2):113–136.

[7] Barros, R. C., Basgalupp, M. P., Cerri, R., da Silva, T. S., and de Carvalho, A. C. P. L. F. (2013). A Grammatical Evolution Approach for Software Effort Estimation. In *Proceedings of the 22nd Genetic and Evolutionary Computation Conference*, pages 1413–1420.

[8] Basgalupp, M. P., Barros, R. C., da Silva, T. S., and Carvalho, A. C. P. L. F. (2013). Software Effort Prediction: A Hyper-heuristic Decision-tree Based Approach. In *Proceedings of the 28th Symposium on Applied Computing*, pages 1109–1116.

[9] Budd, T. A. (1980). *Mutation Analysis of Program Test Data*. Phd thesis, Yale University.

[10] Burke, E. K., Gendreau, M., Hyde, M., Kendall, G., Ochoa, G., Özcan, E., and Qu, R. (2013). Hyper-heuristics: A survey of the state of the art. *Journal of the Operational Research Society*, 64(12):1695–1724.

[11] Burke, E. K., Hyde, M., Kendall, G., Ochoa, G., Ozcan, E., and Qu, R. (2009). A survey of hyper-heuristics. Technical report, School of Computer Science and Information Technology, University of Nottingham.

[12] Burke, E. K., Hyde, M., Kendall, G., Ochoa, G., Özcan, E., and Woodward, J. R. (2010). *Handbook of Metaheuristics*, volume 146, chapter A Classification of Hyper-heuristic Approaches, pages 449–468. Springer US.

[13] Carvalho, V. R., Vergilio, S. R., and Pozo, A. T. R. (2015). Uma hiper-heurística de seleçao de meta-heurísticas para estabelecer sequências de módulos para o teste de software. In *Proceedings of the VI Workshop de Engenharia de Software Baseada em Busca*. In portuguese.

[14] Chakhlevitch, K. and Cowling, P. (2008). *Adaptive and Multilevel Metaheuristics*, volume 136, chapter Hyperheuristics: Recent Developments, pages 3–29. Springer Berlin Heidelberg.

[15] Coello, C. A. C., Lamont, G. B., and Veldhuizen, D. A. V. (2007). *Evolutionary Algorithms for Solving Multi-Objective Problems Second Edition*. Springer Science, 2nd edition.

[16] Coles, H., Laurent, T., Henard, C., Papadakis, M., and Ventresque, A. (2016). PIT: A Practical Mutation Testing Tool for Java (Demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 449–452.

[17] de Mingo López, L. F., Blas, N. G., Peñuela, J. C., and Albert, A. A. (2018). *Enjoying Natural Computing*, chapter ACORD: Ant Colony Optimization and BNF Grammar Rule Derivation, pages 99–113. Springer International Publishing.

[18] de Oliveira, A. A. L., Camilo-Junior, C. G., and Vincenzi, A. M. R. (2013). A coevolutionary algorithm to automatic test case selection and mutant in Mutation Testing. In *Proceedings of the 2013 IEEE Congress on Evolutionary Computation*, pages 829–836.

[19] Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197.

[20] Delamaro, M. E., Barbosa, E. F., Vicenzi, A. M. R., and Maldonado, J. C. (2007). *Introdução ao Teste de Software*, chapter Teste de Mutação, pages 77–118. Elsevier. In portuguese.

[21] Delamaro, M. E., Deng, L., Durelli, V. H. S., Li, N., and Offutt, J. (2014). Experimental Evaluation of SDL and One-Op Mutation for C. In *Proceedings of the 7th International Conference on Software Testing, Verification and Validation*, pages 203–212.

[22] Delamaro, M. E., Maldonado, J. C., and Vincenzi, A. M. R. (2001). *Mutation Testing for the New Century*, chapter Proteum/IM 2.0: An Integrated Mutation Testing Environment, pages 91–101. Springer US.

[23] Delgado-Pérez, P., Medina-Bulo, I., and Merayo, M. G. (2017a). Using Evolutionary Computation to Improve Mutation Testing. In *Proceedings of the 14th Advances in Computational Intelligence*, pages 381–391.

[24] Delgado-Pérez, P., Medina-Bulo, I., Segura, S., García-Domínguez, A., and José, J. (2017b). GiGAn: Evolutionary Mutation Testing for C++ Object-oriented Systems. In *Proceedings of the 32nd Symposium on Applied Computing*, pages 1387–1392.

[25] Di Martino, S., Ferrucci, F., Maggio, V., and Sarro, F. (2013). Towards Migrating Genetic Algorithms for Test Data Generation to the Cloud. In *Software Testing in the Cloud: Perspectives on an Emerging Discipline*, pages 113–135. IGI Global.

[26] Di Penta, M. (2012). Sbse meets software maintenance: Achievements and open problems. In *Proceedings of the 4th International Symposium on Search Based Software Engineering*, volume 7515, pages 27–28. Springer.

[27] Domínguez-Jiménez, J. J., Estero-Botaro, A., García-Domínguez, A., and Medina-Bulo, I. (2010). GAmera: A Tool for WS-BPEL Composition Testing Using Mutation Analysis. In *Proceedings of the 10th International Conference on Web Engineering*, pages 490–493.

[28] Domínguez-Jiménez, J. J., Estero-Botaro, A., García-Domínguez, A., and Medina-Bulo, I. (2011). Evolutionary mutation testing. *Information and Software Technology*, 53(10):1108–1123.

[29] Domínguez-Jiménez, J. J., Estero-Botaro, A., and Medina-Bulo, I. (2009). A Framework for Mutant Genetic Generation for WS-BPEL. In *Proceedings of the 35th Conference on Current Trends in Theory and Practice of Computer Science*, pages 229–240.

[30] Durillo, J. J. and Nebro, A. J. (2011). jMetal: A Java framework for multi-objective optimization. *Advances in Engineering Software*, 42:760–771.

[31] Eiben, A. E. and Smit, S. K. (2011). Parameter tuning for configuring and analyzing evolutionary algorithms. *Swarm and Evolutionary Computation*, 1(1):19–31.

[32] Eiben, A. E. and Smith, J. E. (2003). *Introduction to evolutionary computing*. Springer Science & Business Media.

[33] Ferrari, F. C., Pizzoleto, A. V., and Offutt, J. (2018). A Systematic Review of Cost Reduction Techniques for Mutation Testing: Preliminary Results. In *Proceedings of the 11th International Conference on Software Testing, Verification and Validation Workshops*, pages 1–10.

[34] Ferreira, T. N., Kuk, J. N., Pozo, A., and Vergilio, S. R. (2016). Product selection based on upper confidence bound MOEA/D-DRA for testing software product lines. In *Proceedings of the 2016 IEEE Congress on Evolutionary Computation*, pages 4135–4142.

[35] Ferreira, T. N., Lima, J. A. P., Strickler, A., Kuk, J. N., Vergilio, S. R., and Pozo, A. (2017). Hyper-Heuristic Based Product Selection for Software Product Line Testing. *IEEE Computational Intelligence Magazine*, 12(2):34–45.

[36] Ferrucci, F., Salza, P., Kechadi, M. T., and Sarro, F. (2015). A parallel genetic algorithms framework based on Hadoop MapReduce. In *Proceedings of the 30th Annual Symposium on Applied Computing*, pages 1664–1667.

[37] Ferrucci, F., Salza, P., and Sarro, F. (2017). Using Hadoop MapReduce for Parallel Genetic Algorithms: A Comparison of the Global, Grid and Island Models. *Evolutionary Computation*, pages 1–33.

[38] Fialho, A., Da Costa, L., Schoenauer, M., and Sebag, M. (2010). Analyzing bandit-based adaptive operator selection mechanisms. *Mathematics and Artificial Intelligence*, 60(1–2):25–64.

[39] Garey, M. R. and Johnson, D. S. (1990). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman.

[40] Geronimo, L. D., Ferrucci, F., Murolo, A., and Sarro, F. (2012). A Parallel Genetic Algorithm Based on Hadoop MapReduce for the Automatic Generation of JUnit Test Suites. In *Proceedings of the 5th International Conference on Software Testing, Verification and Validation*, pages 785–793.

[41] Gopinath, R., Alipour, A., Ahmed, I., Jensen, C., and Groce, A. (2015). How hard does mutation analysis have to be, anyway? In *Proceedings of the 26th International Symposium on Software Reliability Engineering*, pages 216–227.

[42] Gopinath, R., Alipour, M. A., Ahmed, I., Jensen, C., and Groce, A. (2016). On The Limits of Mutation Reduction Strategies. *2016 IEEE/ACM 38th International Conference on Software Engineering*, pages 511–522.

[43] Guizzo, G., Bazargani, M., Paixao, M., and Drake, J. H. (2017a). A Hyper-Heuristic for Multi-Objective Integration and Test Ordering in Google Guava. In *Proceedings of the 9th International Symposium on Search Based Software Engineering*, pages 168–174.

[44] Guizzo, G., Colanzi, T. E., and Vergilio, S. R. (2017b). Applying design patterns in the search-based optimization of software product line architectures. *Software & Systems Modeling*, pages 1–26.

[45] Guizzo, G., Fritsche, G. M., Vergilio, S. R., and Pozo, A. T. R. (2015a). A Hyper-Heuristic for the Multi-Objective Integration and Test Order Problem. In *Proceedings of the 24th Genetic and Evolutionary Computation Conference*.

[46] Guizzo, G. and Vergilio, S. R. (2016). Metaheuristic Design Pattern: Visitor for Genetic Operators. In *Proceedings of the 5th Brazilian Conference on Intelligent Systems*.

[47] Guizzo, G. and Vergilio, S. R. (2018). A pattern-driven solution for designing multi-objective evolutionary algorithms. *Natural Computing*, pages 1–14.

[48] Guizzo, G., Vergilio, S. R., and Pozo, A. T. R. (2015b). Evaluating a Multi-Objective Hyper-Heuristic for the Integration and Test Order Problem. In *Proceedings of the 5th Brazilian Conference on Intelligent Systems*.

[49] Guizzo, G., Vergilio, S. R., and Pozo, A. T. R. (2015c). Uma Solução Baseada em Hiper-Heurística para Determinar Ordens de Teste na Presença de Restrições de Modularização. In *Proceedings of the VI Workshop de Engenharia de Software Baseada em Busca*.

[50] Guizzo, G., Vergilio, S. R., Pozo, A. T. R., and Fritsche, G. M. (2017c). A Multi-Objective and Evolutionary Hyper-Heuristic Applied to the Integration and Test Order Problem. *Applied Soft Computing*, 56:331–344.

[51] Harman, M., Burke, E., Clark, J., and Yao, X. (2012a). Dynamic Adaptive Search Based Software Engineering. In *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–8.

[52] Harman, M., Jia, Y., and Langdon, W. B. (2010). A Manifesto for Higher Order Mutation Testing. In *Proceedings of the 3rd International Conference on Software Testing, Verification, and Validation Workshops*, pages 80–89.

[53] Harman, M., Jia, Y., and Zhang, Y. (2015). Achievements, open problems and challenges for search based software testing. In *Proceedings of the 8th International Conference on Software Testing, Verification and Validation*.

[54] Harman, M. and Jones, B. F. (2001). Search-Based Software Engineering. *Information and Software Technology*, 43(14):833–839.

[55] Harman, M., Mansouri, S. A., and Zhang, Y. (2012b). Search-based Software Engineering: Trends, Techniques and Applications. *ACM Computing Surveys*, 45(1):11–61.

[56] Howden, W. E. (1982). Weak Mutation Testing and Completeness of Test Sets. *IEEE Transactions on Software Engineering*, 8(4):371–379.

[57] Hussain, S. (2008). *Mutation clustering*. Master's thesis, King's College London.

[58] Jakubovski Filho, H. L., Lima, J. A. P., and Vergilio, S. R. (2017). Automatic Generation of Search-Based Algorithms Applied to the Feature Testing of Software Product Lines. In *Proceedings of the 31st Brazilian Symposium on Software Engineering*, pages 114–123.

[59] Ji, C., Chen, Z., Xu, B., and Zhao, Z. (2009). A Novel Method of Mutation Clustering Based on Domain Analysis. In *Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering*, volume 9, pages 422–425.

[60] Jia, Y. (2015). Hyperheuristic Search for SBST. In *Proceedings of the 8th International Workshop on Search-Based Software Testing*, pages 15–16.

[61] Jia, Y., Cohen, M., Harman, M., and Petke, J. (2015). Learning Combinatorial Interaction Test Generation Strategies using Hyperheuristic Search. In *Proceedings of the 37th International Conference on Software Engineering*.

[62] Jia, Y. and Harman, M. (2011). An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, 37(5):649–678.

[63] Just, R., Jalali, D., Inozemtseva, L., Ernst, M. D., Holmes, R., and Fraser, G. (2014). Are Mutants a Valid Substitute for Real Faults in Software Testing? In *Proceedings of the 22nd International Symposium on Foundations of Software Engineering*, pages 654–665.

[64] King, K. N. and Offutt, A. J. (1991). A Fortran Language System for Mutation-based Software Testing. *Software – Practice and Experience*, 21(7):685–718.

[65] Kintis, M., Papadakis, M., Papadopoulos, A., Valvis, E., and Malevris, N. (2016). Analysing and Comparing the Effectiveness of Mutation Testing Tools: A Manual Study. In *Proceedings of the 16th International Working Conference on Source Code Analysis and Manipulation*, pages 147–156.

[66] Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press.

[67] Kruskal, W. H. and Wallis, W. A. (1952). Use of Ranks in One-Criterion Variance Analysis. *Journal of the American Statistical Association*, 47(260):583–621.

[68] Kumari, A. C. and Srinivas, K. (2013). Software Module Clustering using a Fast Multi-objective Hyper-heuristic Evolutionary Algorithm. *International Journal of Applied Information Systems*, 5(6):12–18.

[69] Kumari, A. C. and Srinivas, K. (2016). Hyper-heuristic approach for multi-objective software module clustering. *Journal of Systems and Software*, 117:384–401.

[70] Laurent, T., Papadakis, M., Kintis, M., Henard, C., Traon, Y. L., and Ventresque, A. (2017). Assessing and Improving the Mutation Testing Practice of PIT. In *Proceedings of the 10th International Conference on Software Testing, Verification and Validation*, pages 430–435.

[71] Lima, J. A. P., Guizzo, G., Vergilio, S. R., Silva, A. P. C., Filho, H. L. J., and Ehrenfried, H. V. (2016). Evaluating Different Strategies for Reduction of Mutation Testing Costs. In *Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing*, pages 1–10.

[72] Lima, J. A. P. and Vergilio, S. R. (2017). A Multi-objective optimization approach for selection of second order mutant generation strategies. In *Proceedings of the 2nd Brazilian Symposium on Systematic and Automated Software Testing*.

[73] Ma, Y.-S., Offutt, J., and Kwon, Y. R. (2005). MuJava: an automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133.

[74] Maashi, M., Özcan, E., and Kendall, G. (2014). A multi-objective hyper-heuristic based on choice function. *Expert Systems with Applications*, 41(9):4475–4493.

[75] Mann, H. B. and Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60.

[76] Mariani, T., Guizzo, G., Vergilio, S. R., and Pozo, A. T. R. (2016a). A Grammatical Evolution Hyper-Heuristic for the Integration and Test Order Problem. In *Proceedings of the 25th Genetic and Evolutionary Computation Conference*.

[77] Mariani, T., Guizzo, G., Vergilio, S. R., and Pozo, A. T. R. (2016b). Automatic Design of Algorithms Applied to the Multi-Objective TSP Problem. In *Proceedings of the XIII Encontro Nacional de Inteligência Artificial e Computacional*.

[78] Mariani, T. and Vergilio, S. R. (2017). A systematic review on search-based refactoring. *Information and Software Technology*, 83:14–34.

[79] Mateo, P. R. and Usaola, M. P. (2012). Bacterio: Java mutation testing tool: A framework to evaluate quality of tests cases. In *Proceedings of the 28th International Conference on Software Maintenance*, pages 646–649.

[80] Mathur, A. P. (1991). Performance, effectiveness, and reliability issues in software testing. In *Proceedings of the 15th Computer Software and Applications Conference*, pages 604–605.

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

[82] Namin, A. S., Andrews, J., and Murdoch, D. (2008). Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 30th International Conference on Software Engineering*, pages 351–360.

[83] Offutt, A. J., Lee, A., Rothermel, G., Untch, R. H., and Zapf, C. (1996). An Experimental Determination of Sufficient Mutant Operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99–118.

[84] Offutt, A. J., Rothermel, G., and Zapf, C. (1993). An experimental evaluation of selective mutation. In *Proceedings of the 15th International Conference on Software Engineering*, pages 100–107.

[85] Offutt, A. J. and Untch, R. H. (2001). *Mutation Testing for the New Century*, chapter Mutation 2000: Uniting the Orthogonal, pages 34–44. Springer US.

[86] O'Neill, M. and Brabazon, A. (2006). Grammatical Swarm: The generation of programs by social programming. *Natural Computing*, 5(4):443–462.

[87] Pressman, R. (2010). *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Inc., 7th edition.

[88] Quyen, N. T. H., Tung, K. T., Hanh, L. T. M., and Binh, N. T. (2016). Improving mutant generation for Simulink models using genetic algorithm. In *Proceedings of the 15th International Conference on Electronics, Information, and Communications*, pages 1–4.

[89] Ryan, C., Collins, J. J., and Neill, M. O. (1998). Grammatical evolution: Evolving programs for an arbitrary language. In *Genetic Programming*, volume 1391, pages 83–96. Springer Berlin Heidelberg.

[90] Räihä, O. (2010). A survey on search-based software design. *Computer Science Review*, 4(4):203–249.

[91] Sahinoglu, M. and Spafford, E. H. (1990). Sequential Statistical Procedures for Approving Test Sets Using Mutation-Based Software Testing. Technical report, Purdue University.

[92] Salza, P., Ferrucci, F., and Sarro, F. (2016a). Develop, Deploy and Execute Parallel Genetic Algorithms in the Cloud. In *Proceedings of the 24th Genetic and Evolutionary Computation Conference*, pages 121–122.

[93] Salza, P., Ferrucci, F., and Sarro, F. (2016b). elephant56: Design and Implementation of a Parallel Genetic Algorithms Framework on Hadoop MapReduce. In *Proceedings of the 25th Genetic and Evolutionary Computation Conference*, pages 1315–1322.

[94] Sarro, F., Ferrucci, F., Harman, M., Manna, A., and Ren, J. (2017). Adaptive Multi-Objective Evolutionary Algorithms for Overtime Planning in Software Projects. *IEEE Transactions on Software Engineering*, 43(10):898–917.

[95] Silva, R. A., do Rocio Senger de Souza, S., and de Souza, P. S. L. (2016). A systematic review on search based mutation testing. *Information and Software Technology*.

[96] Sommerville, I. (2011). *Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., 9th edition.

[97] Storn, R. and Price, K. (1997). Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces. *Journal of Global Optimization*, 11(4):341–359.

[98] Strickler, A., Prado Lima, J. A., Vergilio, S. R., and Pozo, A. T. R. (2016). Deriving products for variability test of Feature Models with a hyper-heuristic approach. *Applied Soft Computing Journal*, 49:1232–1242.

[99] Vargha, A. and Delaney, H. D. (2000). A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132.

[100] Vincenzi, A. M. R., Maldonado, J. C., Barbosa, E. F., and Delamaro, M. E. (1999). Operadores Essenciais de Interface: Um Estudo de Caso. In *Simpósio Brasileiro de Engenharia de Software*, pages 373–391.

[101] Wohlin, C., Runeson, P., Hst, M., Ohlsson, M. C., Regnell, B., and Wessln, A. (2012). *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated.

[102] Wong, W. E., Mathur, A. P., and Maldonado, J. C. (1995). *Software Quality and Productivity: Theory, practice, education and training*, chapter Mutation versus All-uses: An Empirical Evaluation of Cost, Strength and Effectiveness, pages 258–265. Springer US.

[103] Woodward, M. R. and Halewood, K. (1988). From weak to strong, dead or alive? an analysis of some mutation testing issues. In *Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis*, pages 152–158.

[104] Zhang, J., Wang, Z., Zhang, L., Hao, D., Zang, L., Cheng, S., and Zhang, L. (2016). Predictive Mutation Testing. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 342–353.

[105] Zhang, L., Gligoric, M., Marinov, D., and Khurshid, S. (2013). Operator-based and random mutant selection: Better together. In *Proceedings of the 28th International Conference on Automated Software Engineering*, pages 92–102.

[106] Zhang, L., Hou, S.-S., Hu, J.-J., Xie, T., and Mei, H. (2010). Is Operator-based Mutant Selection Superior to Random Mutant Selection? In *Proceedings of the 32nd International Conference on Software Engineering*, pages 435–444.

[107] Zhang, Q. and Li, H. (2007). MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition. *IEEE Transactions on Evolutionary Computation*, 11(6):712–731.

[108] Zitzler, E., Laumanns, M., and Thiele, L. (2001). SPEA2: improving the strength Pareto evolutionary algorithm. Technical report, Department of Electrical Engineering, Swiss Federal Institute of Technology.

[109] Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C. M., and da Fonseca, V. G. (2003). Performance assessment of multiobjective optimizers: an analysis and review. *IEEE Transactions on Evolutionary Computation*, 7(2):117–132.