Many-Objective Test Case Generation for Graphical User Interface Applications via Search-Based and Model-Based Testing

Valdivino Alexandre de Santiago Júnior[a,*], Ender Özcan[b], Juliana Marino Balera[a]

[a] *Instituto Nacional de Pesquisas Espaciais (INPE)*
Coordenação de Pesquisa Aplicada e Desenvolvimento Tecnológico (COPDT) - Av. dos Astronautas, 1758, São José dos Campos, SP, Brazil, 12227-010
[b] *University of Nottingham*
Computational Optimisation and Learning (COL) Lab, School of Computer Science, Wollaton Road, Nottingham, United Kingdom, NG8 1BB

[*] Corresponding author
Email addresses: valdivino.santiago@inpe.br (Valdivino Alexandre de Santiago Júnior), Ender.Ozcan@nottingham.ac.uk (Ender Özcan), juliana.balera@inpe.br (Juliana Marino Balera)

Abstract

The majority of the studies that generate test cases for graphical user interface (GUI) applications are based on or address functional requirements only. In spite of the fact that interesting approaches have been proposed, they do not address functional and non-functional requirements of the GUI systems, and non-functional properties of the created test suites altogether to generate test cases. This is called a many-objective perspective where several desirable and different characteristics are considered together to generate the test cases. In this study, we show how to combine search-based (optimisation) with model-based testing to generate test cases for GUI applications taking into account the many-objective perspective. We rely on meta and hyper-heuristics and we address two particular issues (problems) considering code-driven and use case-driven GUI testing. As for the code-driven testing, we target desktop applications and automatically read the C++ source code of the system, translate it into an event flow graph (EFG), and use objective functions that are graph-based measures. As for the use case-driven testing, EFGs are created directly via use cases. A rigorous evaluation was performed using 32 problem instances where we considered three different multi-objective evolutionary algorithms and six different selection hyper-heuristics using those algorithms as low-level (meta)heuristics. The performance of the algorithms was compared based on five different indicators, and also a new Multi-Metric Indicator (MMI) utilising multiple indicators and providing a unique measure for all algorithms. Results show that the metaheuristics obtained better performances overall, particularly NSGA-II, while Choice Function was the most outstanding hyper-heuristic approach.
Keywords: Many-Objective Optimisation, Search-Based Software Testing, Model-Based Testing, Graphical User Interface, Metaheuristics and Hyper-Heuristics

1. Introduction

A graphical user interface(GUI)-driven software application is a program with a GUI used as the main entity for interaction with users (Nguyen et al., 2014). Nowadays, many software systems (desktop, mobile, smart TV applications, etc.) possess a GUI front-end. GUI testing is challenging, since the approaches for test case generation must deal with a range of uncertainties associated with the varying level and nature of users (for example, a user could try to add two letters on a calculator) (Banerjee et al., 2013). This challenge is even greater in the context of non-trivial systems where professionals from different backgrounds can make use of the same product. In this sense, it is essential to ensure that the GUI application does not present critical defects

so that it does neither stop working (e.g. crash) nor present erroneous information displayed via the interface. Note that defects in a GUI software application may be found not only in the GUI-related code itself but also in the underlying application code (Robinson & Brooks, 2009).

Test case[1] generation for GUI applications has long been studied and there are various methods and techniques for this purpose with tool support. Model-based GUI testing (Belli et al., 2017; Banerjee et al., 2013; Farto & Endo, 2017; Gove & Faytong, 2011; Arlt et al., 2011; Memon & Nguyen, 2010; Huang et al., 2010; Yuan & Memon, 2010a,b; Chinnapongse et al., 2009; Xie & Memon, 2008), capture and replay testing (Amalfitano et al., 2019; Herbold et al., 2012; Ariss et al., 2010), and monitoring and replay testing (Herbold et al., 2011) are some examples where methods and techniques have been created.

However, the majority of these studies generate test cases based on or addressing functional properties only. For instance, they address the coverage of multiway interactions between events of a GUI model (Yuan & Memon, 2010a,b), the coverage of all edges at least once of an event sequence graph (ESG) (Farto & Endo, 2017), try to unlock gate GUIs (Amalfitano et al., 2019), among others. They do not envisage an approach where functional and non-functional (e.g. energy consumption (Sahar et al., 2019), performance (Seo et al., 2012), security (Mohanta et al., 2020)) requirements of the GUI applications, and non-functional properties of the created test suites (e.g. effectiveness[2] and cost) are considered altogether to generate test cases. Note that some previous studies, during evaluation, perform some analysis related to effectiveness and non-functional requirements (e.g. perfomance) but these are considered the consequence (effect) of the test cases and not the cause to create them. We denote this perspective where several points of view (functional, non-functional requirements of GUI systems and test suites) are the causes to generate the test cases as a *many-objective perspective*[3]. As reported in the literature (Harman et al., 2015; Balera & Santiago Júnior, 2019), taking all of these test objectives together as the main cause to generate test cases is important because not only GUI applications but any other non-trivial software system should be tested in accordance with this many-objective approach.

Therefore, the reasoning of the many-objective perspective is that the test cases are, by design,

[1] A test case is characterised by a sequence of test steps. Note that a sequence differs from a set because repetition of elements is allowed and order matters. The test steps may have test input data and expected result related to them, or they may simply represent actions that must be taken. In this study, we consider a test case as being a sequence of test steps where each test step is characterised by test input data only, omitting the expected result, or an action. Specifically in terms of GUI applications, this definition above may be reformulated where a test case is simply a sequence of events of the GUI.

[2] In this article, we use "effectiveness" to mean the ability of the test suite to detect defects in the software system. Therefore, effectiveness here does not mean the non-functional property in using the GUI.

[3] In the optimisation community, multi-objective and many-objective problems are distinguished based on the number of objectives, where "many" means more objective functions than "multi". Hence, many-objective optimisation problems are the ones with more than three objectives (four or more), and hence multi-objective would be those with three or two objective functions (Gómez & Coello, 2015). One objective function means single objective optimisation. In this study, we deal with four-objective optimisation problems and, hence, we may state that we are addressing many-objective problems. This is the reason to denote a many-objective perspective.

already suitable according to different test objectives. In a typical practical setting, if a professional needs to decide between several test case generation strategies for its new GUI application, eventually he/she has to generate the test suites according to each strategy and, at least considering part of the system, execute the test cases, and perceive the effectiveness, feasibility of the test suites. Then, he/she can decide about the strategy to test the entire software system. With the many-objective perspective, we can have a good indication about the effectiveness, feasibility of the test cases of the different approaches without necessarily demanding their execution, if we add this property as one of the causes to create the test suites.

It is worth mentioning that exhaustive test case generation is infeasible for basically any type of software system considering the immense search space. More importantly, the test case generation problems can be formulated as constraint satisfaction problems which are, in general, NP-complete and so computationally difficult to solve in a reasonable amount of time using exact methods in practice (McMinn, 2004; Khari & Kumar, 2019). Thus, inexact/(meta)heuristic algorithms are often preferred to address such problems.

As the core idea of search-based software testing (SBST) is that the problem of testing a software system is formulated as an optimisation problem (Harman et al., 2015; Saeed et al., 2016; Balera & Santiago Júnior, 2019; Khari & Kumar, 2019), precisely because test objectives resemble objective functions, it is then straightforward to think that metaheuristics (Dokeroglu et al., 2019) and hyper-heuristics (Burke et al., 2019; Drake et al., 2019) can help a lot in this regard.

Our hypothesis is that it is possible to combine SBST (optimisation) with other traditional type of testing adopted for GUI applications, so that we can generate test cases according to the many-objective perspective. Few approaches have been proposed in the context of this hypothesis where metaheuristics such as evolutionary algorithms/genetic algorithms (Rauf & Ramzan, 2018; Latiu et al., 2013b; Menninghaus et al., 2017; Mahmood et al., 2014; Latiu et al., 2013a), particle swarm optimisation (Rauf & Ramzan, 2018; Rauf et al., 2010), local search algorithms (Menninghaus et al., 2017), and ant colony optimisation (Bauersfeld et al., 2011b,a) were used. However, these previous approaches relied on a couple (at maximum) objectives, and no previous study presented a robust evaluation considering several quality (performance) indicators, a very important aspect under the optimisation point of view. Furthermore, none of these previous studies considered hyper-heuristics to generate test cases for GUI applications.

In this study, we show how to combine search-based with model-based testing to generate test cases for GUI applications taking into account the many-objective perspective. We rely on meta and hyper-heuristics and we address two particular issues (problems) considering *code-driven testing* and *use case-driven testing*. As for the code-driven testing, we target desktop applications and automatically read the C++ source code of the system, translate it into an event flow graph (EFG) (Banerjee et al., 2013; Nguyen et al., 2014), and use objective functions that are graph-

based measures to generate the test cases. As for the use case-driven testing, EFGs are created directly where interactions between an actor (user) and the software are defined (i.e. via use cases), and we use the same previously defined objective functions to generate the test cases. We suggest four objective functions addressing not only typical functional requirements of GUI applications, such as coverage of edges of the EFG, but also non-functional properties of the created test suites, i.e. effectiveness and cost.

A rigorous evaluation was performed using 32 many-objective problem instances from these two problems. In our evaluation, we considered: i) three metaheuristics, i.e. multi-objective evolutionary algorithms whose are also the low-level (meta)heuristics (LLHs) of the hyper-heuristics: Nondominated Sorting Genetic Algorithm-II (NSGA-II) (Deb et al., 2002), Indicator-Based Evolutionary Algorithm (IBEA) (Zitzler & Künzli, 2004), and Strength Pareto Evolutionary Algorithm-2 (SPEA2) (Zitzler et al., 2001); and ii) six selection hyper-heuristics: Hyper-Heuristic based on Reinforcement LearnIng, Balanced Heuristic Selection and Group Decision AccEptance - Responsibility (HRISE_R) and Majority (HRISE_M) rules (Santiago Júnior et al., 2020), Hyper-Heuristic based on Random LLH Selection and Random Choice of Move Acceptance Methods (HRMA) (Santiago Júnior et al., 2020; Santiago Júnior & Özcan, 2019), Choice Function hyper-heuristic (HH-CF) (Maashi et al., 2014), a random choice (meta)heuristic selection hyper-heuristic with All Moves acceptance (HH-ALL), and a Learning Automata-based Hyper-Heuristic with a Ranking Scheme Initialisation (HH-RILA) (Li et al., 2019). Note that even if we evaluated NSGA-II, IBEA, and SPEA2 considering many-objective problem instances, they are typically classified as multi-objective algorithms.

We took into account five quality indicators as follows: three convergence-diversity metrics, i.e. hypervolume (s-metric) (Zitzler & Thiele, 1999), inverted generational distance (IGD) (Li & Zhang, 2009), modified inverted generational distance (IGD+) (Ishibuchi et al., 2015); ii) a convergence metric, i.e. ϵ indicator (Zitzler et al., 2003); and iii) a diversity indicator, i.e. generalised spread (Δ^*) (Zhou et al., 2006). Moreover, we also propose a new Multi-Metric Indicator (MMI) where we consider all these quality indicators together providing a unique performance measure for all algorithms.

Furthermore, the well-known **no free lunch theorem** informally asserts that any two search algorithms are equivalent when their performances are averaged across all possible problems (Wolpert & Macready, 2005). However, the empirical results showing how the performance of those approaches compare or which approach performs the best on some particular problems have always been of interest to the relevant researchers and practitioners.

In summary, the main contributions of this study are:

1. We show how optimisation algorithms and classical model-based testing can be combined to generate test cases for GUI applications taking into account the many-objective perspective.

We consider four objective functions which are measures defined over an EFG that represents the GUI application, and hence addressing the problems in a more realistic and many-objective manner;

2. We carry out an extensive rigorous evaluation with three metaheuristics and six selection hyper-heuristics, where the latter have never been used within the GUI test case generation problem;

3. We use five quality (performance) indicators aiming at having a thorough experimental assessment of all approaches;

4. We propose a new general-purpose multi-metric analysis indicator to give a unique performance measure so that professionals can rely on this single outcome to decide which is the best optimisation algorithm for their problem(s).

This article is organised as follows. In Section 2, we present related work. Section 3 shows how we combine search-based with model-based testing to generate test cases in accordance with the many-objective perspective. The experimental design and results are shown in Sections 4 and 5, respectively. In Section 6, we discuss some relevant points related to our research. Conclusions and future research directions are presented in Section 7.

2. Related Work

GUI test case generation is a very active research subfield where many approaches and tools have been proposed, since the majority of software applications have a friendly (some times not so friendly) graphical front-end. In a previous systematic mapping (Banerjee et al., 2013), authors identified three main classes to generate test cases where methods and techniques have been created: model-based GUI testing (Belli et al., 2017; Banerjee et al., 2013; Farto & Endo, 2017; Gove & Faytong, 2011; Arlt et al., 2011; Memon & Nguyen, 2010; Huang et al., 2010; Yuan & Memon, 2010a,b; Chinnapongse et al., 2009; Xie & Memon, 2008), capture and replay testing (Amalfitano et al., 2019; Herbold et al., 2012; Ariss et al., 2010), and random testing (Yang et al., 2014). Out of them, the majority of the contributions are on model-based methods, and within this category, EFGs and finite state machines (Santiago et al., 2008) were the most commonly used models. We divide this section in two parts: non-optimisation and optimisation approaches.

2.1. Non-optimisation Approaches

There are several studies that rely on non-optimisation methods and techniques to generate GUI test cases. In this section, we mention only some recent and other relevant studies within

this category, and identify the similarities and differences of previously proposed approaches and ours.

In (Amalfitano et al., 2019), the authors proposed an approach, named Gate gui UnLocking for AndRoid (juGULAR), for unlocking gate GUIs, type of GUIs that need to be solicited by specific user input event sequences to allow the exploration of parts of the application that cannot be reached otherwise. juGULAR automatically detects the occurrence of a gate GUI and demands human intervention to unlock it at runtime. It is considered an hybrid exploration technique that combines automated GUI exploration with capture and replay, because the user behaviour to unlock a gate GUI is captured by juGULAR and replayed when the same gate GUI is detected, via a machine learning approach, again during the exploration. In terms of test case generation, we may say that it is at first a manual approach via the human interaction which is later replayed when a gate GUI is detected. The evaluation addressed covered activities, covered lines of code, and generated network traffic bytes which are all related to functional properties.

In the context of smart TV systems, model-based testing approaches have been proposed as in (Bures et al., 2020). This study aimed at usability (non-functional property) testing of smart TV applications based on the automated generation of an interaction model, via a crawler which analyses the application's user interface and detects the actual clickable elements. Based on this model, defined user tasks in the smart TV application can be evaluated automatically in terms of the usability of the application. One limitation of their approach is that there are many parameters and thresholds to tune. They manually performed update of thresholds, without no clear or sound directives on how practitioners must do it, in order that their strategy could be significantly better than users executing test scenarios. Note that metaheuristics and hyper-heuristics usually have parameters to tune too, but we believe that to be completely fair when comparing strategies we must not demand that parameters should be tuned for every different application, but rather to rely on already defined values (or approaches should rely on self-tuning of their parameters). In this study, we used the same values of parameters defined for the algorithms in their original articles, as we show in Section 4.5.

Mobile applications testing was addressed in (Farto & Endo, 2017) where test models were reused to reduce the effort on concretisation and verify other characteristics such as device-specific events, unpredictable users' interaction, among others. The models are focused on system testing, mainly events of users and GUIs. Test case generation is based on ESGs which describe the expected behaviour of mobile applications. A test case is indeed a complete event sequence of the ESG. Our method is similar to theirs in this respect since a test case in our approach is a simple circuit of the EFG that represents the GUI application. However, in their case, the single coverage criterion states that all the edges of the ESG must be covered at least once by a test suite. Thus, this is again related to functional aspects of the application. Eventually, their approach may be

able to address robustness testing (unpredictable interactions) but this is not clear in the article.

A strategy to avoid infeasible test cases by predicting which test cases are indeed infeasible using two supervised machine learning approaches, support vector machines (SVMs) and grammar induction, was proposed in (Gove & Faytong, 2011). Their approach helps the professional by predicting which test cases are infeasible, and hence he/she may remove the predicted infeasible test cases before they are executed, prioritise the test suite, or examine the test cases to determine the reasons why they are classified as infeasible.

Another model-based testing approach aimed at producing most suitable GUI models for test case generation (Arlt et al., 2011). They took into account two observations to create such model. The first is that shared event handlers may imply the creation of redundant test cases, and the second is that user interactions are context-sensitive. Then, they generate a model for GUI testing that analyses parts of the source code to detect sequences of user interactions that are suitable for testing. Test case generation is performed under the functional point of view where, starting in the initial state s_0 of the model and with an empty test case, events are randomly added into the test case until the initial state s_0 of the model is reached again, and the length of the test case is larger than a specified threshold. Two basic differences between our approach and theirs is that firstly, in the code-driven problems, our model may have multiple initial states (we call them nonterminal vertices). Secondly, our approach does not demand the user to specify the length of the test case (threshold). Rather, we define constraints which automatically decide the minimum and maximum number of events (vertices) of the test case based on the number of vertices of the EFG. Hence, we avoid creating meaningless test cases in practical terms.

Creating a model of a GUI that can be used to generate potentially problematic test cases (sequences of events) was the main objective in (Xie & Memon, 2008). The authors defined the minimal effective event context (MEEC) of an event e. This MEEC concept was used to empirically demonstrate that, for defect (fault) detection, the MEEC is short and has a well-defined structure, which may be represented by four compact regular expressions. This result was used to automatically develop a model, called event interaction graph (EIG), which was used to generate and execute test cases. The regular expressions provided four testing coverage criteria which, altogether, basically demand the coverage of vertices, edges, and paths of the model.

Studies have been proposed to address the coverage of multiway interactions between events of a GUI model (Yuan & Memon, 2010a,b). In (Yuan & Memon, 2010a), a technique to test multiway interactions among GUI events was presented which is based on analysis of feedback obtained from the runtime state of GUI widgets. They define a model called event semantic interaction graph (ESIG) which is considered for test case generation, and which is derived based on event semantic interaction relationships. In (Yuan & Memon, 2010b), a technique to generate GUI test cases in batches was proposed and named as ALT. Because of its "alternating" nature, ALT enhances

the next batch by using GUI run-time information from the current batch. It is also based on the idea of computing event semantic interaction relationships, and it incrementally goes deeper in generating test cases to cover multiway interactions. Test case generation is then performed considering functional properties.

In order to support defect reproduction, a generic, non-intrusive GUI usage monitoring mechanism was described in (Herbold et al., 2011). The main motivation is to support defect fixing as the reproduction of the defect is the first important step towards its correction. The test case generation approach may be classified as monitoring and replay, similar to capture and replay where the monitoring mechanism resembles the capture phase of capture and replay.

Table 1 summarises the main characteristics of the non-optimisation strategies presented above and ours. The meaning of the columns are:

1. *FUN*: functional requirements;
2. *EFF*: effectiveness of the test suites;
3. *ONF*: other non-functional requirements of the GUI systems and/or of the test suites other than effectiveness and cost of the test suites;
4. *MBT*: relied on model-based testing;
5. *SBST*: relied on search-based software testing;
6. *MNOB*: in accordance with the many-objective perspective;
7. *HH*: supported by hyper-heuristics.

Table 1: Main characteristics of the non-optimisation approaches and ours

Article	FUN	EFF	ONF	MBT	SBST	MNOB	HH
(Amalfitano et al., 2019)	✓			✓			
(Bures et al., 2020)			✓	✓			
(Farto & Endo, 2017)	✓	✓		✓			
(Gove & Faytong, 2011)			✓	✓			
(Arlt et al., 2011)	✓			✓			
(Xie & Memon, 2008)	✓	✓		✓			
(Yuan & Memon, 2010a)	✓	✓		✓			
(Yuan & Memon, 2010b)	✓	✓		✓			
(Herbold et al., 2011)			✓	?			
Ours	✓	✓		✓	✓	✓	✓

In Table 1, a question mark (?) means that it is not clear whether the characteristic is present or not in the proposed approach. We can clearly see the benefits of our approach where we should particularly emphasise the fact that, although these other studies are interesting, they do not meet the many-objective perspective as we have defined, and as we have used to create GUI test suites.

2.2. Optimisation Approaches

In this section, we show some studies that rely on optimisation approaches to generate GUI test cases. A strategy based on particle swarm optimisation to derive test cases for GUI applications was described in (Rauf et al., 2010). The authors addressed a single objective problem, maximising the coverage of test paths. In addition to address a single objetive, the experimental assessment was very shallow with no statistical support.

In (Rauf & Ramzan, 2018), authors used NSGA-II and multi-objective particle swarm optimisation to generate test suites (sets of test cases) for testing context-free GUI applications. They considered two objectives: minimisation of the number of test cases and maximisation of the coverage of test cases. In our work, we also defined similar objectives, along with two additional objective functions based on the concept of test case diversity. Moreover, their method is based on capture and replay and it is not automated as ours, particularly when dealing with the code-driven problem where we automatically create an EFG (model) based on the source code of the application.

In (Latiu et al., 2013b), a test case generation approach based on an evolutionary algorithm/-genetic algorithm was presented. A tool, named EvoGuiTest, was implemented. Unfortunately, there is no detailed information on the algorithmic features of the proposed evolutionary algorithm/genetic algorithm. It is not clear if it is a novel algorithm, or else if it is an adoption of an existing one. The authors solved a single objective problem only and, again, the evaluation was very preliminary with no statistical tests, no use of quality indicators, and no comparison to other algorithms. There is another study from the same authors which it is basically an adaptation for testing GUIs of water monitoring applications (Latiu et al., 2013a).

Hill climbing and simulated annealing as well as multi-objective evolutionary algorithms, including NSGA-II, SPEA2, and Pareto Envelope based Selection - II (PESA-II) (Corne et al., 2001), were used to generate test cases in (Menninghaus et al., 2017). Three objectives were considered: maximisation of branch coverage in the source code, maximisation of EFG coverage, and minimisation of sequence length. However, hill climbing and simulated annealing were used to solve a single objective problem, i.e. maximisation of branch or EFG coverage. Multi-objective evolutionary algorithms were applied via different configurations (setups) where there were one or two-objective problems by combining the maximisation of branch or EFG coverage plus minimisation of sequence length (in some cases, this objective was not considered). The metrics to evaluate the approaches were average runtime and average overall coverage on code. No evaluation was carried out based on quality indicators (hypervolume, ϵ indicator, etc.) and no statistical test was applied to the results.

EvoDroid is an evolutionary testing framework designed particularly for Android applications (Mahmood et al., 2014). This framework combines two main techniques: (i) an Android-specific

program analysis technique that identifies the independent segments of the code amenable to search, and (ii) an evolutionary algorithm that given information of such segments performs a stepwise search for test cases reaching deep into the code. Similar to our study, the approach automatically creates models, interface model and call graph model, reading the source code. However, this approach considers only a single objective of maximising code coverage to solve only one of the objectives that we tackle. We believe that our approach is much more flexible and can be applied in different phases of the software development lifecycle.

In (Bauersfeld et al., 2011b), an approach to find test cases for GUI applications using ant colony optimisation and a metric referred to as maximum call stacks for use within the objective function was introduced. The approach generates test cases online, executing the software system and repeatedly choosing from a set of possible actions. Thus, it is not required to create a model of the GUI. It reads the code and deals with Java SWT systems, but not C++ applications as in our case for the code-driven problem. Again, this work is another example of a single objective problem. There is another study from the same authors which is in the same context (Bauersfeld et al., 2011a).

Another approach addressing GUI test case generation aiming to automatically repair GUI test suites, which have infeasible test cases by generating new test cases that are indeed feasible, was presented in (Huang et al., 2010). A genetic algorithm was used to evolve new test cases that increase the coverage of the test suite while avoiding infeasible sequences. It is a single objective problem but we might say that this unique objective function addresses two characteristics: the feasibility of the test case and the new coverage a test case can contribute based on the coverage already achieved. We might regard feasibility as a sort of non-functional property of the test suites. But, there is no evidence that the authors considered effectiveness to create the test suites as we did. The authors say that their technique generates smaller test suites with better coverage on the longer test sequences. But note that this conclusion of the smaller cost is obtained a posteriori (consequence) and it is not a cause (objective function) to derive the test cases. Moreover, they compared their approach to a random algorithm only while we compared nine approaches in total.

Table 2 presents the main characteristics of the optimisation approaches and ours. The meaning of the columns have already been defined and, as earlier, a question mark (?) means that it is not clear whether the approach presents or not the characteristics.

Thus, in addition to all the differences that we have pointed out above between our approach and these related optimisation studies, where we can also see them in Table 2, there is one additional point to stress: none of them considered hyper-heuristics to generate test cases. So it would be interesting to investigate these general-purpose optimisation algorithms that operate at a higher abstraction level for GUI test case creation, since they have been successfully applied to several continuous multi/many-objective optimisation problems (Maashi et al., 2014; Li et al.,

Table 2: Main characteristics of the optimisation approaches and ours

Article	FUN	EFF	ONF	MBT	SBST	MNOB	HH
(Rauf et al., 2010)	✓			?	✓		
(Rauf & Ramzan, 2018)	✓			?	✓		
(Latiu et al., 2013b)	✓			?	✓		
(Menninghaus et al., 2017)	✓			✓	✓		
(Mahmood et al., 2014)	✓			✓	✓		
(Bauersfeld et al., 2011b)	✓				✓		
(Huang et al., 2010)	✓		✓	✓	✓		
Ours	✓	✓		✓	✓	✓	✓

2019; Santiago Júnior et al., 2020), usually outperforming metaheuristics. The hyper-heuristic research also investigates how high the level of generality of a strategy can be raised. So, it is important to evaluate them against metaheuristics in discrete optimisation problems as well, as explained in the next section.

3. Many-Objective Test Case Generation for GUI Applications

In this section, we show how search-based (optimisation) and model-based testing can be combined to generate GUI test cases in accordance with the many-objective perspective.

An EFG is indeed the main artefact for test case generation. Therefore, this is where model-based testing fits into our proposal. An EFG is a directed graph representing all possible event interactions on a GUI (Nguyen et al., 2014). Each vertex in an EFG represents a GUI event (e.g. click-on-exit-button). A directed edge from vertex x to vertex y represents a follows relationship between x and y, indicating that y follows x, that is event y is allowed to occur immediately after event x. Figure 1 shows an example of a GUI application and its EFG is in Figure 2.

We now provide some important definitions related to our approach.

Definition 3.1. ***Decision variable as a test case****. A simple circuit (Hawick & James, 2008) of an EFG is a test case composed of a sequence of events (vertices) of such an EFG that represents the GUI application. A decision variable is one element of a solution. Thus, each decision variable of a solution is an integer which identifies a simple circuit, i.e. a test case*[4]*.*

Definition 3.2. ***Solution as a test suite****. A solution is formed by a sequence of decision variables. A solution in a population generated by an optimisation algorithm is a test suite, i.e. a sequence of test cases.*

Note that our definitions of test suite and test case differ from others. For instance, in (Huang et al., 2010), a solution is indeed a test case and each decision variable is a test step (event). In

[4]Precisely, the test cases created are indeed **abstract** test cases but it is easy to translate them into **executable** test cases. We we will denote here an abstract test case simply as a test case for simplicity.

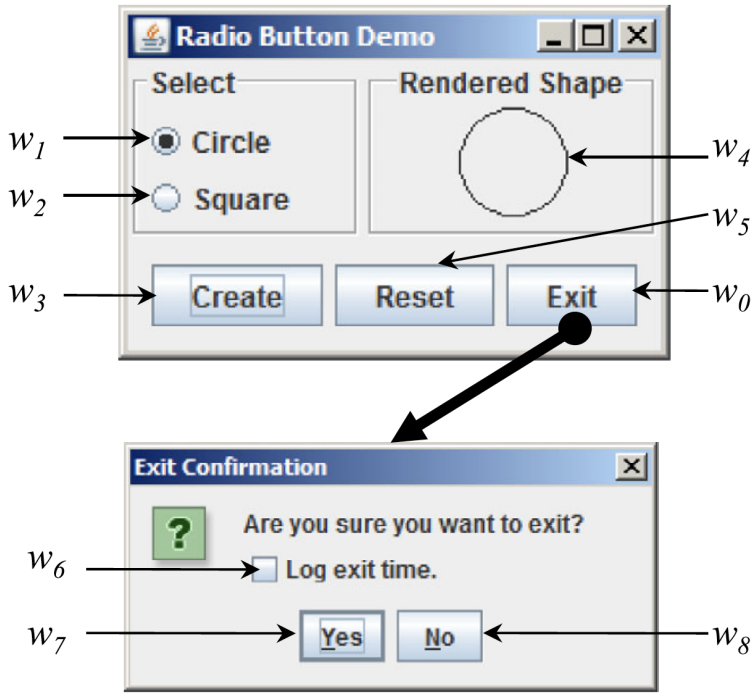


Figure 1: Example of a GUI application. Adapted from (Nguyen et al., 2014)

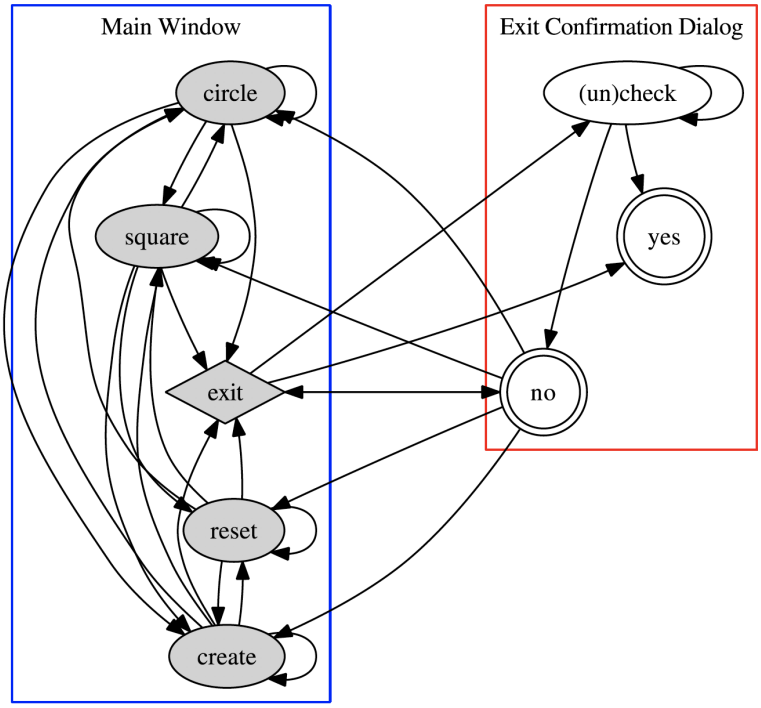


Figure 2: Example of EFG. Adapted from (Nguyen et al., 2014)

their case, a population is indeed a test suite while in our case a population is a set of test suites. There are two reasons behind this decision. Firstly, in our case, the final population after executing an algorithm is composed of nondominated solutions only. This means that each solution (test suite) has at first the same strength in terms of Pareto dominance, and a professional may select any of them to be executed. But, there are methods proposed by the optimisation community to decide which of the nondominated solutions to select in a population (Ferreira et al., 2007). Hence, we provide flexibility to the professional to select a test suite according to his/her preference.

Secondly, as each decision variable identifies a test case (simple circuit) we may have longer test cases and these can expose faults undetectable by shortest ones (Xie & Memon, 2008). In our approach, constraints automatically decide the minimum and maximum number of events (vertices) of the test case based on the number of vertices of the EFG (see Section 4.7). This implies that longer test cases can be created without increasing significantly the number of decision variables. Usually, as higher the number of decision variables in a solution, more time is required for the algorithms to finish.

Hence, this is a discrete optimisation problem. The code-driven and use case-driven problems are considered in this study. We detail each problem in the following sections.

3.1. Code-Driven Problem

In this problem, our approach reads the code related to GUI features of a software system and automatically generates an EFG. We focused on the applications developed in the C++ programming language since it is still one of the most used programming languages in the global development scenario, as recently corroborated by the TIOBE[5] index (Eras et al., 2019). Precisely, the type of GUI application we considered is written in Qt-extended C++, where Qt[6] is a cross-platform application development framework.

The automated generation of an EFG is based on the identification of the interactive components (e.g. buttons, text boxes, etc.) that make up the GUI. This identification is given by reading the source code related to the GUI and depends on the implementation of each of these components. Once identified, each component becomes a vertex in the graph. We then define two types of components:

1. **nonterminal**. The component allows to be clicked multiple times and moreover it allows, after being selected, that another component is clicked too;
2. **terminal**. The component closes the GUI or, after being clicked, opens another GUI.

[5]https://www.tiobe.com/tiobe-index/. Access on: May 10, 2022.
[6]https://www.qt.io/. Access on: May 10, 2022.

After the identification of components/vertices, a graph is built where each vertex is connected to every other vertex and also to itself (in this case, the origin and destination of the edge are the same). Note that the creation of the edges respects the follows relationships that we have previously defined. However, the generation of a simple circuit (i.e. a test case) follows the rule in which its first vertex is always a nonterminal one, and the penultimate vertex of the simple circuit is always of type terminal. Hence, we guarantee that the last event to be fired according to the test case is always a terminal component (note that in a simple circuit the first and the last vertices are the same, and hence the last vertex (component) to be fired is the penultimate one). With no lack of generality, these assumptions are valid for many types of GUI systems.

Figure 3 illustrates a problem instance (GUI) based on the TerraAmazon[7] software product, a C++ geographic information system (GIS) multi-user editor for vector geographic data which has been under development at the *Instituto Nacional de Pesquisas Espaciais* (INPE). This problem instance is referred to as GUI4, which is one of the 24 problem instances from the TerraAmazon software, considered in our study (see Section 4.3) and representing the code-driven problem to generate test cases.

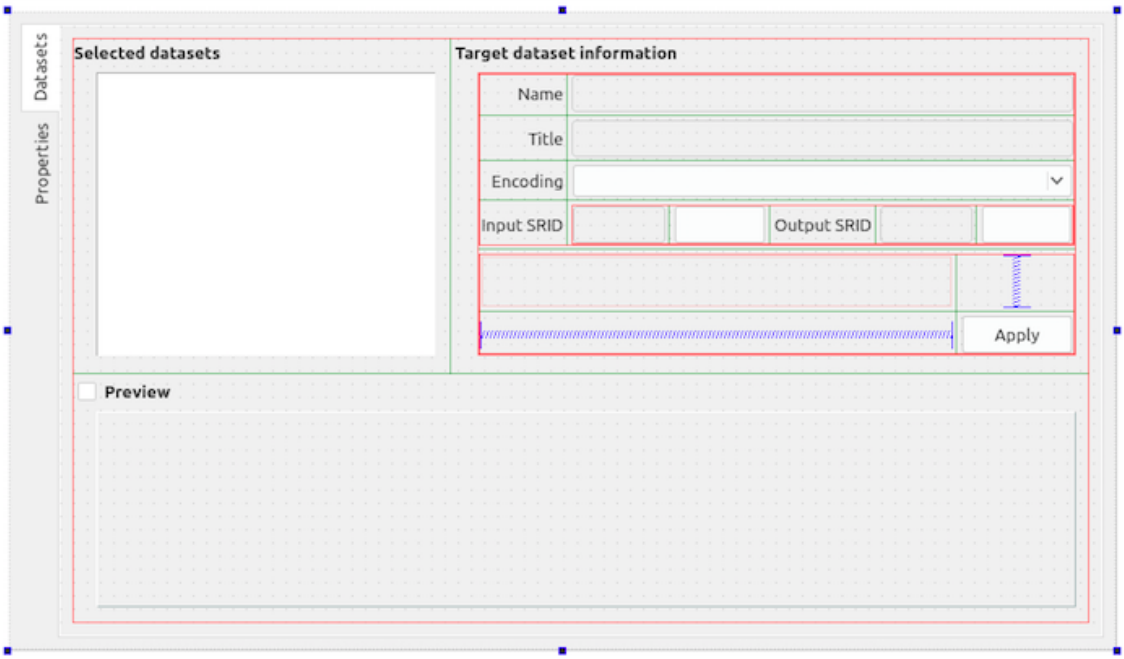


Figure 3: TerraAmazon, GUI4: print screen

Figure 4 shows part of the Qt-extended C++ code, that is the implementation of GUI4 while the generated EFG produced parsing this code is in Figure 5. The EFG has a single terminal vertex, m_applypushbutton, while all the others are nonterminal vertices. In this case, we can

[7]http://www.terraamazon.dpi.inpe.br/. Access on: May 10, 2022.

clearly see that we have edges between each pair of vertices including self-edges. An example of test case (simple circuit) is presented below:

$$tc_i = \{m\_selecteddatasetlistwidget, m\_datasetnamelineedit, m\_datasettitlelineedit, m\_encodingcombobox, m\_sridoutputlineedit, m\_sridoutputpushbutton, m\_sridinputpushbutton, m\_sridinputlineedit, m\_applypushbutton, m\_selecteddatasetlistwidget\}. \quad (1)$$

```
// TerraLib
#include "../../../dataaccess/dataset/DataSetAdapter.h"
...
#include "ui_DataSetOptionsWizardPageForm.h"
...
// Qt
#include <QIcon>
#include <QMessageBox>

te::qt::widgets::DataSetOptionsWizardPage::DataSetOptionsWizardPage(QWidget* parent)
  : QWizardPage(parent),
    m_ui(new Ui::DataSetOptionsWizardPageForm)
{
// setup controls
  m_ui->setupUi(this);

//build form
  ...

// connect signals and slots
  connect(m_ui->m_sridInputPushButton, SIGNAL(clicked()), this,
SLOT(sridInputSearchToolButtonPressed()));
  connect(m_ui->m_sridOutputPushButton, SIGNAL(clicked()), this,
SLOT(sridOutputSearchToolButtonPressed()));
  connect(m_ui->m_applyPushButton, SIGNAL(clicked()), this, SLOT(applyChanges()));
  connect(m_ui->m_dataPreviewGroupBox, SIGNAL(clicked(bool)), this,
SLOT(onDataPreviewGroupBoxClicked()));
  connect(m_constraintWidget.get(), SIGNAL(constraintsChanged()), this,
SLOT(onDataPreviewGroupBoxClicked()));
  connect(m_dataSetAdapterWidget.get(), SIGNAL(dataSetAdapterChanged()), this,
SLOT(onDataPreviewGroupBoxClicked()));
  connect(m_ui->m_selectedDatasetListWidget, SIGNAL(itemPressed(QListWidgetItem*)), this,
SLOT(datasetPressed(QListWidgetItem*)));
}

te::qt::widgets::DataSetOptionsWizardPage::~DataSetOptionsWizardPage() =
    default;

void te::qt::widgets::DataSetOptionsWizardPage::set(const
std::list<te::da::DataSetTypePtr>& datasets,
                                                    const te::da::DataSourceInfoPtr&
datasource,
                                                    const te::da::DataSourceInfoPtr&
targetDatasource)
{
  ScopedCursor wcursor(Qt::WaitCursor);
  m_ui->m_selectedDatasetListWidget->clear();
  clearForm();
  m_datasets.clear();
...
}
...
```

Figure 4: TerraAmazon, GUI4: part of the Qt-extended C++ code

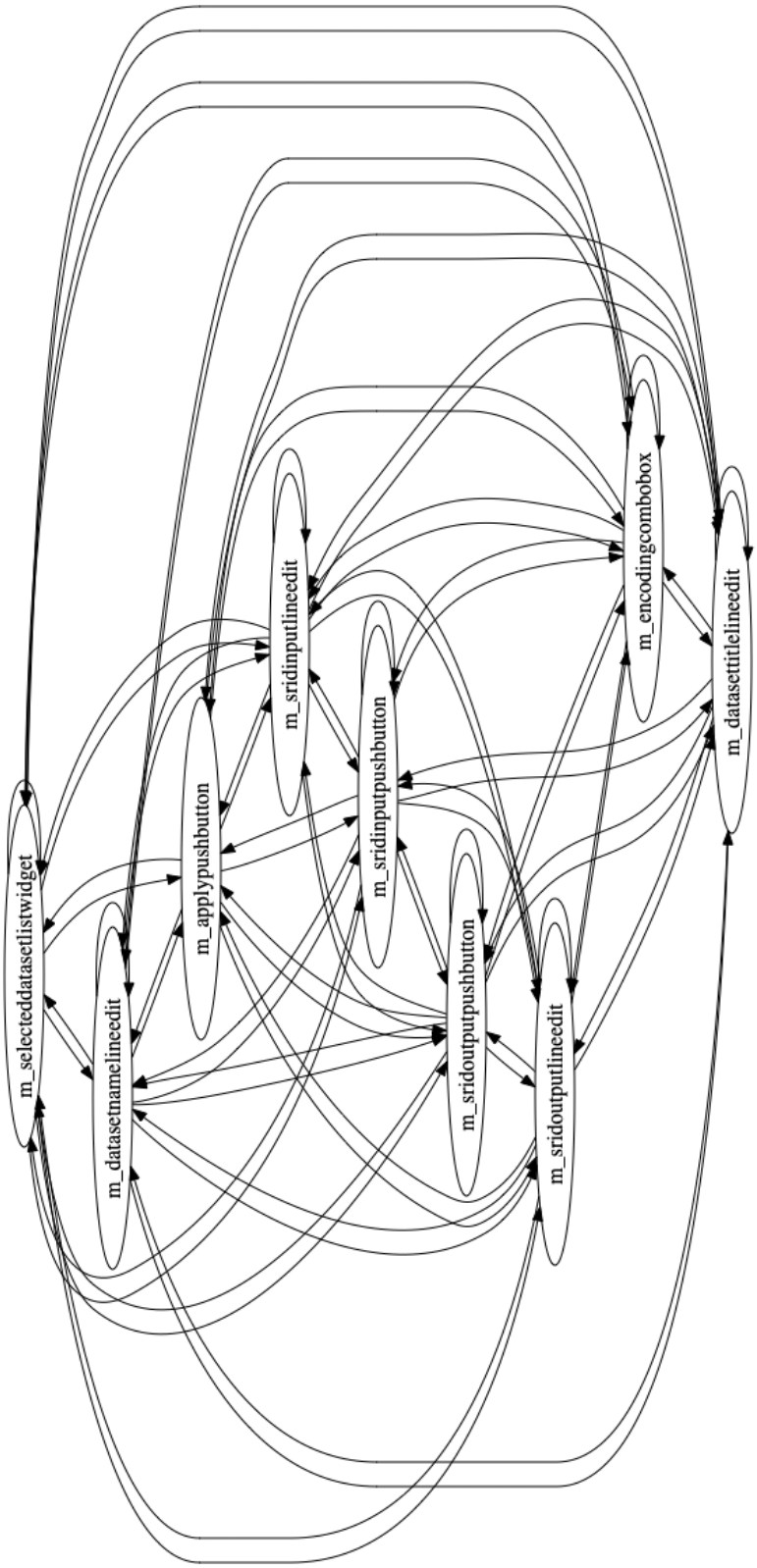


Figure 5: TerraAmazon, GUI4: EFG

3.2. Use Case-Driven Problem

GUI testing is essentially a type of system or acceptance testing. Our motivation behind the use case-driven problem is to create test cases according to a more user-centered perspective, and not necessarily based on all the possibilities that reading the source code may allow (code-driven problem).

In this scenario, ultimately, we would like to still produce test cases (simple circuits) based on EFGs. However, this time, while the simple circuit still "ends" in a terminal vertex, not all nonterminal vertices are enabled to start a simple circuit. This occurs because this is a use case perspective where a user defines a series of steps to interact with the software system via its GUI. This sequence of steps is usually created based on the requirements specification or even some sort of tutorial, user manual of the application under consideration. Hence, having terminal vertices, we differentiate between two types of nonterminal vertices: **initial**, where a simple circuit may begin, and **intermediate**, where a simple circuit neither starts nor ends.

We assume that there is a single terminal vertex for the use case-driven problem instances in this study. However, the graphs can contain several initial and intermediate nonterminal vertices. Moreover, there is an edge whose source is the single terminal vertex and destination is an initial nonterminal vertex. Thus, we have as many as these edges as the number of initial nonterminal vertices. But, no edge starts in the terminal vertex and ends in an intermediate nonterminal node.

Figure 6 shows the EFG for a problem instance obtained from the social media application WhatsApp. We directly created this EFG based on a tutorial of the application[8]. This problem instance is referred to as WUP1, which is one of the eight instances considered in this study representing the use case-driven problem (see Section 4.3) for generating test cases. In this EFG, the single terminal vertex is closeWU, there is a single initial nonterminal vertex, selectChat, and all remaining vertices (typeText, playVideo, ...) are of type intermediate nonterminal. An example of test case (simple circuit) is presented below:

$$tc_i = \{selectChat, playVideo, selectEmoji, typeTextEmoji, \\ typeEmojiText, typeTextAgain, closeWU, selectChat\}. \quad (2)$$

3.3. Objective Functions

In order to properly characterise an optimisation problem so that algorithms (e.g. metaheuristics and hyper-heuristics) can be applied for test case generation, we need to define the objective functions. The objective functions are measures over the EFG regardless if we consider the code-driven or use case-driven perspectives. We defined four objective functions as described below:

[8] https://www.youtube.com/watch?v=y3EdIiJeTXk. Access on: May 10, 2022.

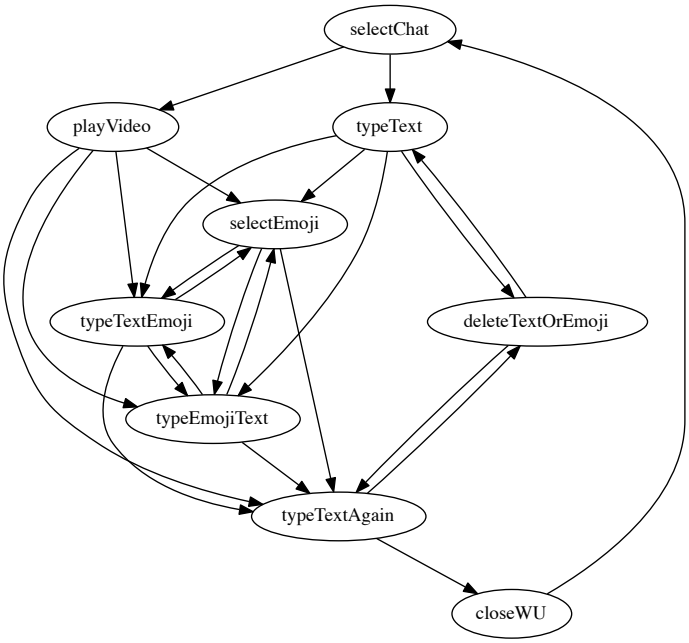


Figure 6: WhatsApp, WUP1: EFG

1. Size of the test suite. This is a *cost* measure which is simply the sum of the number of vertices (which means events of a GUI to be fired) of all test cases (decision variables) of a test suite (solution). It is to be minimised since, in general, the fewer events required to be stimulated by the test suite, the less demand for its execution. We denote this objective function as $f_1(\widehat{x})$ where $\widehat{x}$ is a solution;

2. Test case diversity. Test case diversity aided by similarity measures (Hemmati et al., 2013) is an *effectiveness* metric the testing community has been addressing. The idea here is to minimise the pairwise similarity between test cases of a test suite. It is a less costly effectiveness measure compared to mutation analysis (Silva et al., 2017) where we must usually create a considerable number of mutants, execute the test suites of different approaches, and calculate the mutation score in order to realise which approach is the best. Note that even if mutation testing is indeed a test case generation strategy, it is common, in the testing community, to use the mutation score as a means of evaluating the effectiveness of different test case generation approaches. In this case, we have two objective functions where one relies on Gower-Legendre (dice) measure, and this is objective function $f_2(\widehat{x})$. The other relies on Sokal-Sneath (anti-dice) measure and this is $f_3(\widehat{x})$;

3. Edge coverage. This is a more traditional functional testing objective related to the GUI

application where the goal is to cover as many as possible edges of an EFG. It is to be maximised since, in general, the more graph edges are covered by the test suite, the better. This is objective function $f_4(\widehat{x})$.

Hence, we follow the many-objective perspective where the test cases are, by design, already suitable according to different objective functions (test objectives). In formal terms, our many-objective optimisation problem can be formulated as follows:

$$\begin{aligned} \text{minimise} \quad & F(\widehat{x}) = (f_1(\widehat{x}), f_2(\widehat{x}), f_3(\widehat{x}), f_4(\widehat{x}))^T \\ \text{subject to} \quad & \widehat{x} \in \Omega \end{aligned}$$

where Ω is the decision variable space, $F : \Omega \rightarrow R^4$ consists of the four objective functions we have just described, and R^4 is the objective space.

In order to make it clear how the values of the objective functions are obtained, let us assume the problem instance WUP1 (Figure 6) and also that we define that each solution has three decision variables. Moreover, let us say that a test suite, i.e. a solution $\widehat{x_1}$, is composed by the three test cases (simple circuits) below:

$$\begin{aligned} tc_1 = \{&selectChat, typeText, \\ &deleteTextOrEmoji, typeTextAgain, closeWU, selectChat\} \end{aligned} \tag{3}$$

$$\begin{aligned} tc_2 = \{&selectChat, typeText, selectEmoji, \\ &typeEmojiText, typeTextAgain, closeWU, selectChat\} \end{aligned} \tag{4}$$

$$\begin{aligned} tc_3 = \{&selectChat, playVideo, selectEmoji, typeTextEmoji, \\ &typeEmojiText, typeTextAgain, closeWU, selectChat\}. \end{aligned} \tag{5}$$

Notice that $\widehat{x_1} = \langle 1, 2, 3 \rangle$ but each integer value represents one of the test cases, i.e. tc_1, tc_2, and tc_3. Hence, $f_1(\widehat{x_1})$ is simply the sum of all vertices of all test cases, i.e. $f_1(\widehat{x_1}) = 21$.

Gower-Legendre (dice) and Sokal-Sneath (anti-dice) similarities measures are given by the general formulation below (Hemmati et al., 2013):

$$sim(tc_i, tc_j) = \frac{|tc_i \cap tc_j|}{|tc_i \cap tc_j| + w \times (|tc_i \cup tc_j| - |tc_i \cap tc_j|)} \tag{6}$$

where tc_i, tc_j are two test cases, and $w = 1/2$ means the Gower-Legendre (dice) measure while $w = 2$ is the Sokal-Sneath (anti-dice) one. Then, we calculate the similarity between every pair of test cases of a test suite (solution) and take the average of such similarity measures. In the example above, we have:

$$\begin{aligned} |tc_1 \cap tc_2| &= 5; \\ |tc_1 \cup tc_2| &= 8; \\ |tc_1 \cap tc_3| &= 4; \\ |tc_1 \cup tc_3| &= 10; \\ |tc_2 \cap tc_3| &= 6; \\ |tc_2 \cup tc_3| &= 9. \end{aligned} \tag{7}$$

The Gower-Legendre measures are $sim_{gl}(tc_1, tc_2) = 0.77$, $sim_{gl}(tc_1, tc_3) = 0.57$, and $sim_{gl}(tc_2, tc_3) = 0.8$. Thus, the average Gower-Legendre measure is $f_2(\widehat{x_1}) \approx 0.71$. The Sokal-Sneath measures are $sim_{ss}(tc_1, tc_2) = 0.46$, $sim_{ss}(tc_1, tc_3) = 0.25$, and $sim_{ss}(tc_2, tc_3) = 0.5$. The average Sokal-Sneath value is then $f_3(\widehat{x_1}) \approx 0.4$.

The edge coverage must be maximised but we turn this into a minimisation problem too as shown below:

$$f_4(\widehat{x_1}) = 1 - \frac{|CE|}{|E|} \tag{8}$$

where $|CE|$ is the number of covered edges due to all test cases of a test suite, and $|E|$ is the total number of the edges of the EFG. A covered edge is counted only once if more than one test case traverses it. In the example above, the EFG (Figure 6) has 24 edges and the three test cases cover 12 out of them. Hence, $f_4(\widehat{x_1}) = 0.5$.

Notice that our approach is useful not only to test GUI software applications. In fact, one can generate test cases for any software system (desktop, mobile, web applications, critical or noncritical software, etc.) whose structure and/or behaviour can be represented by a graph.

4. Experimental Design

In this section, we describe the main design options of the rigorous evaluation, a controlled experiment, we carried out to assess the performance of the metaheuristics and hyper-heuristics for GUI test case generation.

4.1. Objective

The objective of this evaluation is to identify which out of nine optimisation algorithms is the best regarding GUI test case generation. We considered three metaheuristics in our evaluation: NSGA-II, IBEA, and SPEA2. Moreover, six selection hyper-heuristics were assessed: HRISE_R, HRISE_M, HRMA, HH-CF, HH-ALL, and HH-RILA. Details about the reasons to select these algorithms are in Section 4.4.

It is important to stress that this experiment is designed from the point of view of the optimisation community evaluating the population-based approaches for the code-driven and use case-driven problems. The use of several different quality indicators is usual and recommended in this setting to gain insights into different search properties of the algorithms, such as convergence and diversity. Therefore, we rely on five quality indicators: hypervolume, IGD, IGD+, ϵ indicator, and generalised spread (Δ^*). In addition, we propose a new indicator obtained by combining all these quality indicators together providing a unique performance measure for all algorithms. Details about all these indicators are in Section 4.6.

We also wonder whether, considering the same problem instances (see Section 4.3), the algorithms are too sensitive to different configurations of decision variables/maximum number of simple circuits. Hence, we define two configurations known as less and more. More information about such configurations is in Section 4.7.

4.2. Research Questions and Variables

The research questions (RQs) related to this experiment are shown below:

1. **RQ_1** - Which out of the nine algorithms is the best with respect to the configuration less? Overall, which one performs better: metaheuristics or hyper-heuristics?

2. **RQ_2** - Which out of the nine algorithms is the best with respect to the configuration more? Overall, which one performs better: metaheuristics or hyper-heuristics?

3. **RQ_3** - Is there an algorithm that is clearly superior than all the others regarding both configurations?

The independent variables are the optimisation algorithms. The dependent variables are the values of the quality indicators which may take different formats: front-normalised, normalised (front-normalised), and our new proposed indicator (see Section 4.6).

4.3. Problems

As for the code-driven problem, we considered 24 problem instances (GUIs) of the TerraAmazon software product where GUIx uniquely identifies a problem instance, that is the code related to a particular TerraAmazon GUI. In our study, the TerraAmazon product has been preferred over

other software systems, simply because we believe that the TerraAmazon GUIs cover a variety of functional components, including classical widgets such as buttons, combo boxes, text boxes, etc. that a typical Qt/C++ application would have. We have shown an example of such GUIs in Figure 3 while part of the respective code is in Figure 4. Table 3 summarises the characteristics of the code-driven problem instances in terms of the number of vertices and edges of the EFGs.

Table 3: Characteristics of the code-driven problem instances

Problem Instance	**#Vertices**	**#Edges**
GUI1	8	63
GUI2	35	1,224
GUI3	12	143
GUI4	9	80
GUI5	12	140
GUI6	13	165
GUI7	10	96
GUI8	14	192
GUI9	16	252
GUI10	20	396
GUI11	26	672
GUI12	17	285
GUI13	24	572
GUI14	11	120
GUI15	18	323
GUI16	10	99
GUI17	9	77
GUI18	24	572
GUI19	23	525
GUI20	13	165
GUI21	32	1,020
GUI22	24	572
GUI23	11	117
GUI24	9	77

We created eight problem instances related to the use case-driven problem where four are use cases for the LibreOffice Writer application (identified as LOFx), three are for WhatsApp (identified as WUPx), and one is for YouTube (identified as YOUx). Recall that in this problem, we directly defined the EFG that represents the problem instance, i.e. interactions with the GUIs of the applications. We have shown an example of such problem instances in Figure 6. Table 4 shows the characteristics of the use case-driven problem instances in terms of the number of vertices and edges of the EFGs.

4.4. Algorithms

Firstly, all algorithms we selected were implemented/adapted based on the jMetal framework (Durillo & Nebro, 2011), version 5.6. The three metaheuristics (NSGA-II, IBEA, and SPEA2) we selected are indeed multi-objective evolutionary algorithms, and they are also the LLHs of the six

Table 4: Characteristics of the use case-driven problem instances

Problem Instance	**#Vertices**	**#Edges**
LOF1	37	467
LOF2	37	608
LOF3	37	479
LOF4	37	560
WUP1	9	24
WUP2	34	191
WUP3	37	196
YOU1	26	108

selection hyper-heuristics (HRISE_R, HRISE_M, HRMA, HH-CF, HH-ALL, and HH-RILA) that make part of the evaluation. We chose these metaheuristics due to their reported success in multi-objective optimisation and they still have competitive performances in continuous optimisation problems. The first reason to opt for the six hyper-heuristics is that previous studies (Santiago Júnior et al., 2020; Santiago Júnior & Özcan, 2019; Li et al., 2019) considered these hyper-heuristics using the same three metaheuristics as LLHs for continuous optimisation problems. Hence, we would like to contrast the conclusions of these previous articles with the outcomes of these discrete optimisation problems. Secondly, some of these algorithms are very recent and thus it is important to perceive their performances in distinct types of problems. We provide a brief description of each algorithm below.

NSGA-II (Deb et al., 2002) is one of the most popular multi-objective evolutionary algorithm up-to-date. It is a sort of benchmark algorithm to be challenged when new population-based optimisation search methods are proposed. It is a fast nondominated sorting approach with lower computational complexity which also addresses elitism. Its authors also presented a crowded-comparison operator which guides the selection process at the various stages of the algorithm towards a uniformly spread-out Pareto Optimal Front.

IBEA (Zitzler & Künzli, 2004) is another classical algorithm based on the idea of flexible integration of preference information. It is a general indicator-based multi-objective evolutionary algorithm, and the main reasoning is to first define the optimisation goal in terms of a binary performance measure (indicator) and then to directly use this measure in the selection process.

In SPEA2 (Zitzler et al., 2001), the authors proposed an algorithm which is an improved version of SPEA and has, as additional characteristics, a fine-grained fitness assignment strategy, a density estimation technique, and an enhanced archive truncation method.

As for the selection hyper-heuristics, HRISE_R and HRISE_M (Santiago Júnior et al., 2020) are two of the recent such optimisation algorithms which embed a heuristic selection method based on roulette wheel supported by reinforcement learning followed by a balanced exploitation/exploration procedure. Moreover, they use a two-level move acceptance strategy: Only Improving plus a group-decision framework where several move acceptance methods are considered. Within the

group-decision framework, HRISE_R relies on the **responsibility** rule where a single move acceptance method takes the responsibility/authority for the decision to accept a population, while HRISE_M is based on the **majority** rule which counts the votes for the accept/reject decision, and the majority leads to the final decision.

HRMA (Santiago Júnior et al., 2020; Santiago Júnior & Özcan, 2019) is a simpler version of the two previous selection hyper-heuristics where we have a simple random heuristic selection method. There is still a two-level move acceptance approach with Only Improving but now we have a random choice among the available move acceptance methods under the group decision-making: hence the one chosen decides to accept/reject a population. Sometimes, we will denote HRISE family these three hyper-heuristics (HRISE_R, HRISE_M, HRMA) since they were developed based on the same general structure.

In (Maashi et al., 2014), the authors presented a hyper-heuristic based on Choice Function, HH-CF, in which NSGA-II, SPEA2, and the Multi-Objective Genetic Algorithm (MOGA) (Fonseca & Fleming, 1998) are the LLHs. In our case, we adapted HH-CF to consider IBEA and not MOGA as a third LLH. The heuristic selection method is based on a two-stage ranking scheme and four quality indicators: algorithm effort, ratio of nondominated individuals (RNI), hypervolume, and uniform distribution (Tan et al., 2002). HH-ALL is a rather simple selection hyper-heuristic with a random LLH selection method and All Moves acceptance approach.

HH-RILA (Li et al., 2019) is another recent selection hyper-heuristic. This algorithm is based on learning automata, a sort of reinforcement learning method, and it uses a ranking scheme initialisation. LLHs are NSGA-II, IBEA, and SPEA2. The authors defined an LLH selection method, named ϵ-RouletteGreedy, in which they apply roulette wheel in the initial stage and, after that, they select between greedy and roulette wheel based on ϵ, the probability[9] of applying the greedy heuristic selection method. Such a probability is increased linearly during the execution.

4.5. Parameters of the Algorithms and Execution

With respect to all metaheuristics (NSGA-II, IBEA, SPEA2), selection of individuals was binary tournament, we used simulated binary crossover (SBX) (Deb & Agrawal, 1995) with probability 0.9 and distribution index 20, and polynomial mutation with probability $1/n$ (n = number of parameters) and distribution index 20. Note that such parameters were also used in the hyper-heuristics where the multi-objective evolutionary algorithms are the LLHs.

We executed all approaches for each problem instance for 100,000 evaluations, population size was fixed as 100, and each experiment was repeated for 30 trials. Table 5 summarises the values of the main parameters used for all algorithms. For more details about the meaning of such

[9]This probability, ϵ, is not related to the quality indicator ϵ.

parameters, see the research articles where the main hyper-heuristics were proposed, i.e. (Santiago Júnior et al., 2020; Maashi et al., 2014; Li et al., 2019).

Table 5: Experimental evaluation: values of parameters

Common Parameters	
Number of Evaluations	100,000
Population Size	100
Trials	30
Specific Parameters	
HRISE_M, HRISE_R, HRMA	
Iterations/LLH	250 or 100
Maximum Iterations	1,000
Maximum Decision Points	7
α	1.0 (HRISE_M) / 0.1 (HRISE_R)
η_a	2 (HRISE_M) / 3 (HRISE_R)
η_r	2 (HRISE_M) / 1 (HRISE_R)
γ	0.000075
HH-RILA	
Iterations/LLH	10
τ	0.9
m	3.0
K	3
Δ_v	0.0075
HH-CF	
Maximum Iterations	1,000
Maximum Decision Points	25
α	100
HH-ALL	
Maximum Iterations	1,000
Maximum Decision Points	25
NSGA-II, IBEA, SPEA2	
Maximum Iterations	1,000
Maximum Decision Points	1

4.6. Quality Indicators and Types of Assessments

There are several quality indicators (Jiang et al., 2014; Tan et al., 2002) proposed in the literature to measure the adequacy of a population generated by an algorithm to solve a given optimisation problem. In our evaluations, we considered the following five quality indicators: hypervolume, IGD, IGD+, ϵ indicator, and generalised spread (Δ^*).

We took into account at least the front-normalised values of the indicators. For instance, the front-normalised hypervolume is obtained via the normalisation of the "raw" hypervolume based on the minimum and maximum values of the objective functions for a problem instance. From this point onward, we will denote the front-normalised hypervolume simply as hypervolume, h, and as higher this value, the better the algorithm. Likewise we will identify I, $I+$, ϵ, and Δ^* as the (front-normalised) IGD, IGD+, ϵ indicator, and generalised spread, respectively, for which

lower values indicate a better algorithm.

Furthermore, since we deal with real-world problems, in order to obtain the reference points and the *"True" Pareto Front* for the calculation of the quality indicators, we created the so-called *True Known Pareto Front* where, for each problem instance, we joined all final populations of all algorithms after the 30 trials, obtained the nondominated solutions, and removed the repeated ones.

We carried out three types of analyses. Firstly, we performed a cross-domain analysis with the goal of observing the generalisation strength of the algorithms across all problem instances, and not performing a case-by-case (single problem instance) evaluation. Generalisation is one of the desirable characteristics of any optimisation approach. Moreover, we also defined a second level of normalisation of the indicators. For instance, the normalised (front-normalised) hypervolume, h_N, is defined below (Santiago Júnior et al., 2020; Li et al., 2019):

$$h_N = \frac{h^{max}_{(\forall a,p)} - \overline{h_{(a_i,p)}}}{h^{max}_{(\forall a,p)} - h^{min}_{(\forall a,p)}} \tag{9}$$

where $h^{max}_{(\forall a,p)}$ and $h^{min}_{(\forall a,p)}$ are the maximum and minimum values, respectively, of the hypervolume, h, due to all algorithms a for a problem instance p, and $\overline{h_{(a_i,p)}}$ is the average value of the hypervolume due to algorithm a_i for p. However, note that the formulation of h_N is like a maximisation problem (maximise hypervolume) is turned into a minimisation problem. Hence, the lower the value of h_N, the better.

However, for the other four remaining indicators, the normalised value is calculated in a standard manner. For instance, the normalised (front-normalised) IGD, I_N, is obtained as follows:

$$I_N = \frac{\overline{I_{(a_i,p)}} - I^{min}_{(\forall a,p)}}{I^{max}_{(\forall a,p)} - I^{min}_{(\forall a,p)}} \tag{10}$$

where $I^{max}_{(\forall a,p)}$ and $I^{min}_{(\forall a,p)}$ are the maximum and minimum values, respectively, of the IGD, I, due to all algorithms a for a problem instance p, and $\overline{I_{(a_i,p)}}$ is the average value of the IGD due to algorithm a_i for p. The lower I_N, the better. Thus, the normalised (front-normalised) IGD+ ($I+_N$), ϵ indicator (ϵ_N), and generalised spread (Δ^*_N) are calculated as in equation 10, and also the lower their values, the better. From this point onward, we will denote a normalised (front-normalised) indicator simply as normalised indicator (e.g. normalised hypervolume, h_N).

Hence, our cross-domain analysis is carried out by calculating the averages of the normalised quality indicators considering all problem instances of all problems: $\overline{h^{APR}_N}$, $\overline{\epsilon^{APR}_N}$, $\overline{I^{APR}_N}$, $\overline{\Delta^{*APR}_N}$, $\overline{I+^{APR}_N}$. Again, the algorithm which gets the lower average of such normalised indicators is the best overall.

Secondly, we performed statistical analysis of the results obtained by the algorithms. Here,

we decided not to rely on the normalised indicators, considering indeed h, ϵ, I, Δ^*, and $I+$ in this situation. We believe that using the second-degree of normalisation may mask the results of the statistical test and hence the choice for the first-degree of normalisation. We applied a two-tailed permutation test (conditional inference procedure) (Hothorn et al., 2008) for multi-group comparison with significance level equal to 0.05. Moreover, we now performed a case-by-case analysis, i.e. we checked for each problem instance if an algorithm a_i was significantly better (“>”) than an algorithm a_j, if it was worse (“<”), or else if there was no significant difference (“∼”).

Analysing the results of the performances of all algorithms considering multiple quality indicators is really important. But, eventually, a decision-maker would prefer having a single unified result that indicates which is the most suitable optimisation algorithm for all problems, or for a particular problem, rather than looking at each quality indicator in its own, i.e. in isolation. We then propose a *Multi-Metric Indicator* (MMI) where we take the results of all five quality indicators together and provide a single outcome. This is our third type of evaluation.

The calculation of the MMI is as follows:

1. Firstly, we get the average values of all normalised quality indicators of all algorithms for all problem instances. This is the final outcome of the cross-domain analysis. Specifically, we get $\overline{h_N^{APR}}$, $\overline{\epsilon_N^{APR}}$, $\overline{I_N^{APR}}$, $\overline{\Delta_N^{*APR}}$, $\overline{I+_N^{APR}}$ which are such average values of hypervolume, ϵ indicator, IGD, generalised spread, and IGD+, respectively;

2. Then, we perform a normalisation by identifying the minimum and maximum values of each quality indicator in isolation as presented below. Roughly speaking, this can be seen as a third level of normalisation. In Equation 11, q_{NN} is the new normalised value of a quality indicator of an algorithm, q_N is the original average value of a normalised quality indicator ($\overline{h_N^{APR}}$, $\overline{\epsilon_N^{APR}}$, $\overline{I_N^{APR}}$, $\overline{\Delta_N^{*APR}}$, $\overline{I+_N^{APR}}$) of an algorithm, q_N^{min} and q_N^{max} are the minimum and maximum values considering a certain normalised indicator:

$$q_{NN} = \frac{q_N - q_N^{min}}{q_N^{max} - q_N^{min}} \quad ; \tag{11}$$

3. Hence, we calculate the Euclidean distance considering the performance of each algorithm under all quality indicators and an ideal point ($\widehat{0}$). This is the MMI, $\mathcal{M}$, of an algorithm:

$$\mathcal{M} = ||\langle 0,0,0,0,0\rangle - \langle h_{NN}, \epsilon_{NN}, I_{NN}, \Delta^*_{NN}, I+_{NN}\rangle|| \tag{12}$$

4. The best algorithm is the one with the lowest $\mathcal{M}$.

4.7. Simple Circuits Configurations and Constraints

As we have mentioned earlier in Section 4.1, we would like to know about the sensitiveness of the algorithms regarding different configurations of decision variables/maximum number of simple circuits. We then defined two simple circuits configurations and obtained the results according to them. In the configuration **less**, a solution (test suite) has five decision variables (set DV; recall that each decision variable is an integer which identifies a simple circuit) and we set a maximum of $500 \times |DV| = 2,500$ possible test cases (simple circuits). On the other hand, in the configuration **more**, there are 10 decision variables and $1,000 \times |DV| = 10,000$ possible simple circuits.

Moreover, we also defined two constraints in the number of vertices a test case (simple circuit) may have. In the constraint **narrow**, the length of a test case (the number of vertices it may have), $|tc_i|$, is:

$$\text{narrow} \Longrightarrow (0.75 \times |V|) \leq |tc_i| \leq (1.25 \times |V|) \tag{13}$$

where $|V|$ is the number of vertices of the EFG that represents the problem instance. On the other hand, the constraint **wide** is as follows:

$$\text{wide} \Longrightarrow (0.5 \times |V|) \leq |tc_i| \leq (1.5 \times |V|). \tag{14}$$

These constraints serve to avoid generating short and meaningless test cases in practical terms. Table 6 shows how we used the configurations and constraints in accordance with each problem. Hence, the constraint narrow was selected only for the code-driven problem instances under the configuration less. In the other cases, the constraint wide was the option.

Table 6: Simple circuits configurations and constraints

Problem	**Configuration less**	**Configuration more**
code-driven	narrow	wide
use case-driven	wide	wide

4.8. Validity

Our results are associated with version 5.6 of the jMetal framework where the metaheuristics and quality indicators are implemented, and where we also developed the hyper-heuristics on top of it. Moreover, all runs were executed on the same hardware/software platform, an iMac with 2.7 GHz Intel i5 processor, 8 GB of RAM memory, and macOS High Sierra Operating System. This avoids any influence of the computing platform on the results. We believe that replication of this study by other researchers will produce similar results, and our study has a high conclusion validity since the measures are reliable.

The problem instances were GUIs of a software product and use cases of three other applications, and thus we neither had any human/nature/social factor nor unanticipated events to

interrupt the collection of the measures once started to pose an internal validity. Hence, our controlled experiment has a high internal validity.

However, we have a threat to external validity related to the population. The 32 problem instances are interesting but we need to consider more code-driven and use case-driven problem instances in order to generalise the results. However, the results of this controlled experiment are valuable as shown in the next section.

5. Experimental Results

In this section, we present the results of our experiment whose features have been explained in Section 4. All final populations (`VAR` files) of all algorithms, the respective values of objective functions (`FUN` files), all True Known Pareto Fronts (`.pf` files), and all EFGs (`.dot` files) of both configurations (less and more) are available online[10]. Furthermore, two tools[11] were developed to fully support our approach and they are publicly available too.

5.1. Configuration Less

As for the configuration less, Table 7 presents the results of the cross-domain performance analysis using as metric the normalised hypervolume, h_N, for the code-driven problem instances, while Table 8 presents the results regarding the use case-driven problem instances and all problem instances. In these tables, $\overline{h_N^{COD}}$ is the average value of h_N for the code-driven problem instances, $\overline{h_N^{USE}}$ is the average value of h_N for the use case-driven problem instances, and the average of all 32 problem instances is $\overline{h_N^{APR}}$. The intermediate means of the normalised hypervolume, $\overline{h_N^{COD}}$ and $\overline{h_N^{USE}}$, are placed here only for a better understanding of the calculation required by the cross-domain analysis. But, the value that really matters is the average value that represents all problem instances of all problems, i.e. $\overline{h_N^{APR}}$. In grey background we show the top performance approaches where in bold it is the best algorithm, normal font is the second best, and in italics it is the third best.

In terms of the code-driven problem, we see that NSGA-II was the best, followed by HH-CF and SPEA2 and, in the use case-driven problem, NSGA-II was again the best, the second best was SPEA2, and HH-CF was the third. However, overall concerning h_N, NSGA-II was the top algorithm followed by HH-CF and SPEA2. The first important remark is about HH-CF where we have already pointed out, in previous study (Santiago Júnior et al., 2020), the good performance of this hyper-heuristic for some problem instances contradicting previous results where HH-CF presented low performance with respect to the hypervolume (Li et al., 2019).

[10] https://github.com/vsantjr/HRISE-DiscreteReal/tree/master/Experiment%20Data. Access on: May 10, 2022.
[11] https://github.com/vsantjr/HRISE-DiscreteReal; https://github.com/BaleraJuliana/CppEFGTranslator. Access on: May 10, 2022.

Table 7: Cross-domain analysis for the code-driven problem instances, configuration less: h_N

	HRISE_M	HRISE_R	HRMA	IBEA	NSGA-II	SPEA2	HH-ALL	HH-CF	HH-RILA
GUI1	0.07731101	0.10376872	0.09981437	0.19257312	8.507214e-03	8.302682e-02	0.37004090	4.601590e-02	0.088419194
GUI2	0.05305004	0.05217340	0.05024710	0.04520481	2.273510e-02	2.461050e-02	0.26667729	2.485243e-02	0.027364994
GUI3	0.05698504	0.05284079	0.06199761	0.03918174	2.476143e-02	4.712535e-02	0.26872555	3.745455e-02	0.027420058
GUI4	0.01811220	0.02949370	0.02437403	0.09322467	2.350970e-03	1.755631e-02	0.30193143	1.090126e-02	0.045106024
GUI5	0.03775412	0.03073172	0.03393428	0.04339389	5.819247e-03	2.068130e-02	0.15857896	1.569080e-02	0.018167393
GUI6	0.03567925	0.02989464	0.03238993	0.04160067	4.632950e-03	2.277342e-02	0.24894592	1.386653e-02	0.018272304
GUI7	0.03116970	0.01892297	0.02801013	0.08657145	1.768353e-03	1.428678e-02	0.20525971	1.075080e-02	0.043842436
GUI8	0.02775397	0.02380624	0.03205334	0.04417819	5.165630e-03	2.185202e-02	0.16856431	1.317028e-02	0.019217918
GUI9	0.02757319	0.02818022	0.02338844	0.02113247	3.288233e-03	8.225268e-03	0.12781406	7.244350e-03	0.007239956
GUI10	0.04011842	0.03043820	0.03876372	0.02891881	1.727867e-02	1.847686e-02	0.24401614	1.875035e-02	0.014540840
GUI11	0.03623134	0.02617025	0.03300869	0.03028949	7.456845e-03	1.272656e-02	0.20937379	1.319648e-02	0.015001033
GUI12	0.02290002	0.02285275	0.01817092	0.02326197	2.763300e-03	8.235529e-03	0.22359319	6.160941e-03	0.007269120
GUI13	0.03303853	0.03232894	0.03993729	0.03010183	9.961669e-03	1.401829e-02	0.27236855	1.173639e-02	0.015591719
GUI14	0.05568418	0.03303459	0.03916624	0.04263298	6.629002e-03	3.065693e-02	0.29476877	1.971773e-02	0.016530251
GUI15	0.03979737	0.04527738	0.04351366	0.03506728	1.362966e-02	2.970274e-02	0.28944857	2.021234e-02	0.015303418
GUI16	0.03779578	0.03278176	0.03863614	0.09231048	2.395470e-03	2.764327e-02	0.27653066	1.771002e-02	0.036082291
GUI17	0.02296564	0.01764168	0.01697319	0.09755103	1.676597e-03	2.231343e-02	0.26295806	9.543465e-03	0.041944880
GUI18	0.02137769	0.03138886	0.02942952	0.02562717	8.676485e-03	9.468055e-03	0.19532787	1.099339e-02	0.010217284
GUI19	0.02474530	0.03021454	0.02250526	0.02251918	7.596704e-03	9.246451e-03	0.14749955	1.065876e-02	0.011197137
GUI20	0.02811372	0.03299651	0.03836481	0.04814709	6.270267e-03	2.304916e-02	0.22161304	1.475409e-02	0.021677620
GUI21	0.03156988	0.02195924	0.02472046	0.02287379	8.235703e-03	1.019709e-02	0.24797116	9.308812e-03	0.009836836
GUI22	0.02198338	0.03511707	0.01976491	0.02160570	9.388181e-03	1.263762e-02	0.23309540	1.096335e-02	0.011932755
GUI23	0.03270318	0.02696969	0.03560591	0.04819086	7.887075e-03	2.054642e-02	0.20546995	1.880379e-02	0.017348957
GUI24	0.03245355	0.01758312	0.02570746	0.11340472	2.135109e-03	2.101911e-02	0.24606107	7.733958e-03	0.051840276
$\overline{h_N^{COD}}$	0.03528611	0.03360696	0.03543656	0.05373181	**7.958744e-03**	*2.208647e-02*	0.23694308	1.584128e-02	0.024640196

Table 8: Cross-domain analysis for the use case-driven problem instances and for all problem instances, configuration less: h_N

	HRISE_M	HRISE_R	HRMA	IBEA	NSGA-II	SPEA2	HH-ALL	HH-CF	HH-RILA
LOF1	0.03708273	0.04354926	0.04117137	0.03829460	1.520373e-02	2.587959e-02	0.19264207	2.788872e-02	0.023570036
LOF2	0.02952615	0.03561611	0.03058975	0.01874405	1.393088e-02	1.595956e-02	0.26225579	1.654268e-02	0.010232772
LOF3	0.04489326	0.06941929	0.06406599	0.02424095	8.143101e-03	3.122515e-02	0.26947929	1.591564e-02	0.011159522
LOF4	0.02216853	0.02209580	0.02409868	0.05653257	2.216317e-02	1.020245e-02	0.09466297	1.755157e-02	0.012584815
WUP1	0.01750758	0.01401272	0.02277137	0.05251687	3.437201e-16	1.565614e-05	0.19652645	3.437201e-16	0.006875571
WUP2	0.09956173	0.07447048	0.08551887	0.22409304	3.968530e-02	4.055866e-02	0.25458713	4.874564e-02	0.061238516
WUP3	0.14552016	0.13479433	0.16050473	0.40092270	6.332109e-02	7.628299e-02	0.32701587	9.273448e-02	0.112767827
YOU1	0.04395980	0.04918372	0.05904825	0.08725825	2.000832e-02	1.325181e-02	0.21373337	2.132042e-02	0.029665972
$\overline{h_N^{USE}}$	0.05502749	0.05539271	0.06097113	0.11282538	**2.280695e-02**	2.667198e-02	0.22636287	*3.008739e-02*	0.033511879
$\overline{h_N^{APR}}$	0.04022145	0.03905340	0.04182020	0.06850520	**1.167080e-02**	*2.323285e-02*	0.23429803	1.940281e-02	0.026858116

Table 9 presents a summarised version of the results of the four remaining normalised quality indicators showing only the average values of all problem instances: ϵ indicator ($\overline{\epsilon_N^{APR}}$), IGD ($\overline{I_N^{APR}}$), generalised spread ($\overline{\Delta_N^{*APR}}$), and IGD+ ($\overline{I+_N^{APR}}$). In two normalised quality indicators, ϵ_N and I_N, we see the very same previous order of best performance algorithms: NSGA-II, HH-CF, and SPEA2. As for $I+_N$, NSGA-II was still the best approach where SPEA2 and HH-CF switched positions, i.e. SPEA2 was the second and HH-CF was the third best algorithm. But, surprisingly, HH-ALL was the best approach regarding Δ_N^* where IBEA was the second and SPEA2 was the third one.

With respect to the statistical evaluation, in Table 10 "$>$" means the leftmost algorithm was significantly better than the rightmost one, "$<$" means the leftmost algorithm was significantly worse than the rightmost one, and "$\sim$" means no significant difference. Moreover, the number that appears in each cell of the table means in how many times, out of the 32 instances, the leftmost algorithm was significantly better than the rightmost one, in how many problem instances there was no statistical difference, and also in how many times the leftmost was the worse. For example, regarding the hypervolume, in the third line of the table we see that HRISE_M was significantly better than IBEA in 14 problem instances, in 15 instances there was no statistical difference, and in three problem instances IBEA was significantly better than HRISE_M.

We realise that the statistical outcomes are basically in line with the previous results based on the cross-domain analysis considering the normalised values of the quality indicators. For instance, in terms of h, ϵ and I, NSGA-II won all the pairwise comparisons (eight wins), HH-CF obtained seven wins while SPEA2 got six wins[12]. These are the top contenders just as in the cross-domain analysis. In terms of Δ^*, HH-ALL, IBEA, and NSGA-II obtained eight, seven, and five wins, respectively, and these are the three best approaches. Hence, the only difference is that SPEA2 and not NSGA-II was the third according to the cross-domain analysis. As for $I+$, there is an agreement that NSGA-II was the best (eight wins) but now HH-CF (seven wins) and SPEA2 (six wins) switched positions.

If we compare the recently proposed selection hyper-heuristics, the HRISE family and HH-RILA, there is basically an agreement. The cross-domain analysis states that HRISE_R and HRISE_M were each better than HH-RILA in terms of ϵ_N, I_N, and Δ_N^* while HH-RILA was better with respect to h_N and $I+_N$. We can clearly confirm these performances by looking at Table 10. The only disagreement regarding this previous conclusion is related to the comparison HRMA $\times$ HH-RILA in terms of the generalised spread because the cross-domain analysis says that HH-RILA was slightly better than HRMA, while the statistical evaluation states that HRMA

[12]Each algorithm can get at maximum eight wins since we have nine algorithms in total. Hence, a "win" means that an algorithm a_i was significantly better more times than other algorithm a_j.

Table 9: Cross-domain analysis for all problem instances, configuration less: ϵ_N, I_N, Δ^*_N, $I+_N$

	HRISE_M	HRISE_R	HRMA	IBEA	NSGA-II	SPEA2	HH-ALL	HH-CF	HH-RILA
$\overline{\epsilon_N^{APR}}$	0.17081706	0.16925104	0.17384102	0.4014268	**8.367461e-02**	*0.122384607*	0.3264633	1.066287e-01	0.20598567
$\overline{I_N^{APR}}$	0.08149988	0.07794398	0.08298372	0.29318411	**0.035856669**	*0.052284543*	0.20352293	0.047060663	0.09947329
$\overline{\Delta_N^{*APR}}$	0.6966376	0.6976178	0.6989199	0.5910151	0.6773317	*0.6634811*	**0.5393301**	0.6797998	0.6980882
$\overline{I+_N^{APR}}$	0.032347365	0.030829413	0.033230255	0.066960170	**1.003470e-02**	1.319687e-02	0.14867535	*1.432011e-02*	0.024450671

Table 10: Statistical evaluation, configuration less. Caption: Comp = Comparison

Comp	$>$	$\sim$	$<$	$>$	$\sim$	$<$	$>$	$\sim$	$<$	$>$	$\sim$	$<$	$>$	$\sim$	$<$
	h			ϵ			I			Δ^*			$I+$		
HRISE_M × HRISE_R	2	28	2	1	31	0	1	31	0	0	31	1	1	31	0
HRISE_M × HRMA	1	31	0	0	32	0	0	32	0	1	31	0	1	31	0
HRISE_M × IBEA	14	15	3	31	1	0	31	0	1	1	5	26	28	2	2
HRISE_M × NSGA-II	0	1	31	0	2	30	0	2	30	5	14	13	1	0	31
HRISE_M × SPEA2	0	12	20	0	12	20	0	9	23	9	11	12	0	4	28
HRISE_M × HH-ALL	32	0	0	25	7	0	26	6	0	0	4	28	25	7	0
HRISE_M × HH-CF	0	6	26	0	5	27	0	6	26	1	30	1	0	3	29
HRISE_M × HH-RILA	2	9	21	7	24	1	8	23	1	7	20	5	4	17	11
HRISE_R × HRMA	0	31	1	1	31	0	0	32	0	0	32	0	0	31	1
HRISE_R × IBEA	16	12	4	31	1	0	31	0	1	1	5	26	27	3	2
HRISE_R × NSGA-II	0	1	31	0	2	30	0	3	29	3	16	13	0	1	31
HRISE_R × SPEA2	0	13	19	0	9	23	0	9	23	10	8	14	0	6	26
HRISE_R × HH-ALL	32	0	0	27	5	0	26	6	0	0	7	25	28	4	0
HRISE_R × HH-CF	0	4	28	0	4	28	0	5	27	0	31	1	0	3	29
HRISE_R × HH-RILA	3	9	20	11	19	2	8	23	1	8	18	6	5	17	10
HRMA × IBEA	11	18	3	31	0	1	31	0	1	1	6	25	27	3	2
HRMA × NSGA-II	0	1	31	0	0	32	0	1	31	4	16	12	0	1	31
HRMA × SPEA2	0	9	23	0	10	22	0	10	22	8	13	11	0	3	29
HRMA × HH-ALL	31	1	0	27	5	0	25	7	0	0	6	26	25	7	0
HRMA × HH-CF	0	3	29	0	4	28	0	5	27	0	30	2	0	2	30
HRMA × HH-RILA	3	4	25	11	18	3	9	21	2	7	22	3	4	17	11
IBEA × NSGA-II	0	1	31	0	0	32	0	0	32	27	4	1	0	2	30
IBEA × SPEA2	0	4	28	0	1	31	0	1	31	17	8	7	1	0	31
IBEA × HH-ALL	29	3	0	3	12	17	2	18	12	3	19	10	12	20	0
IBEA × HH-CF	0	3	29	0	1	31	0	1	31	21	10	1	0	1	31
IBEA × HH-RILA	0	0	32	0	1	31	0	1	31	21	8	3	0	1	31
NSGA-II × SPEA2	19	11	2	19	13	0	24	7	1	14	4	14	24	4	4
NSGA-II × HH-ALL	32	0	0	32	0	0	31	1	0	0	14	18	30	2	0
NSGA-II × HH-CF	20	11	1	15	17	0	14	17	1	12	17	3	25	6	1
NSGA-II × HH-RILA	22	9	1	31	1	0	31	0	1	13	10	9	28	0	4
SPEA2 × HH-ALL	32	0	0	32	0	0	31	1	0	1	15	16	30	2	0
SPEA2 × HH-CF	2	24	6	0	27	5	1	26	5	6	16	10	4	19	9
SPEA2 × HH-RILA	8	20	4	27	4	1	30	2	0	6	25	1	28	3	1
HH-ALL × HH-CF	0	0	32	0	0	32	0	1	31	23	9	0	0	2	30
HH-ALL × HH-RILA	0	0	32	0	14	18	0	14	18	26	6	0	0	7	25
HH-CF × HH-RILA	8	22	2	29	3	0	28	4	0	8	23	1	24	4	4

was indeed better than HH-RILA.

Table 11 shows the results of the third evaluation we have proposed based on the MMI indicator. Here, we realise that there is a difference in the ranking if we compare to the previous results (cross-domain analysis, statistical evaluation). Now, SPEA2 was the best approach followed by NSGA-II and HH-CF.

This difference might be explained precisely by the combination of all metrics and the three-level normalisation process. In other words, even if NSGA-II was better than SPEA2, looking at the results of the cross-domain analysis, in terms of h_N, ϵ_N, I_N, and $I+_N$, SPEA2 got superior

performance regarding Δ_N^*. Eventually this better performance under Δ_N^* and consistent results considering the other four quality indicators are enough, and SPEA2 turned out to be the most standing approach according to the MMI. Since MMI is based on the outcomes of the cross-domain analysis, where we do not adopt a case-by-case evaluation but rather we consider the averages of the normalised quality indicators across all problem instances of all problems, the consistent performance of an algorithm taking into account all problem instances influences its MMI. NSGA-II and HH-CF obtained the second and third places, respectively, and HRISE_M, HRISE_R, HRMA all surpassed HH-RILA.

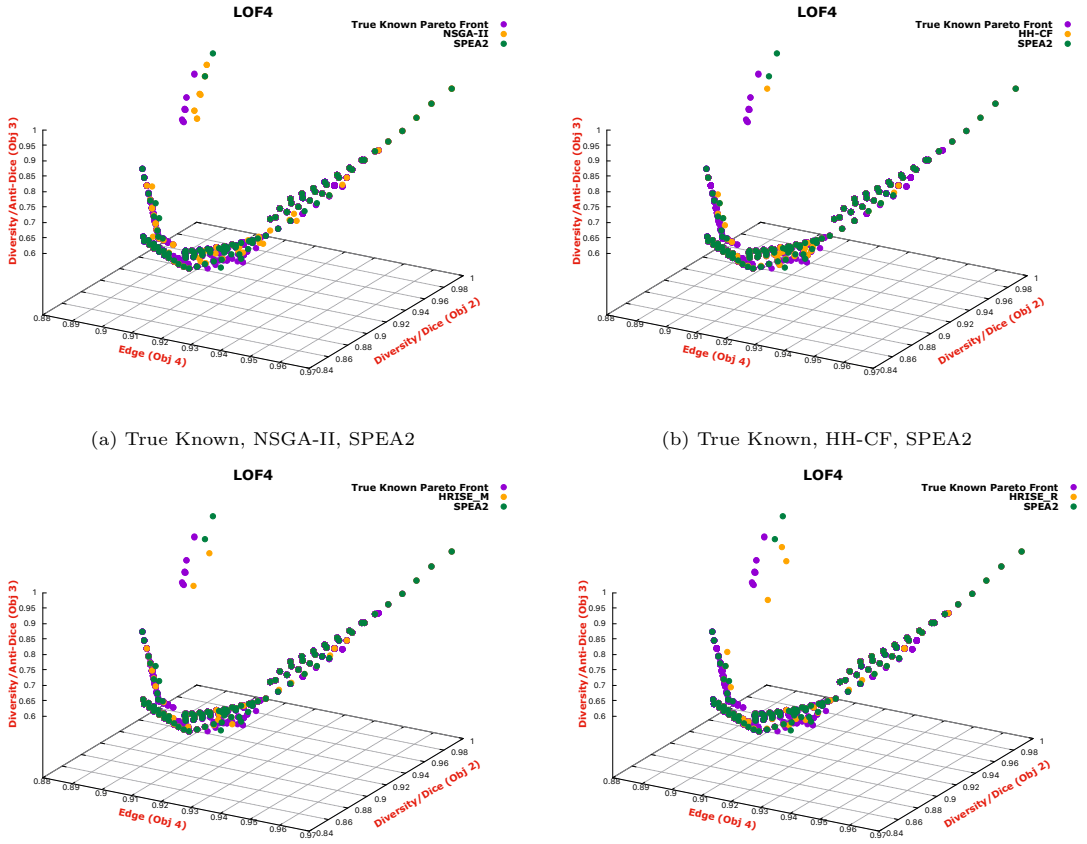
Table 11: MMI, configuration less

Ranking	**Algorithm**	**Euclidean Distance**
1	SPEA2	0.792036643075591
2	NSGA-II	0.864726943701915
3	HH-CF	0.885445758314289
4	HRISE_M	1.05859485301485
5	HRISE_R	1.05861066340369
6	HRMA	1.07721572577917
7	HH-RILA	1.10197819263326
8	IBEA	1.52926425105214
9	HH-ALL	1.73446292695956

Regardless the assessment considered, one point is clear according to the results and this helps answering RQ_1. As for the configuration less, the metaheuristics, particularly SPEA2 and NSGA-II, obtained in general better performances than the hyper-heuristics. This conclusion is in disagreement with the outcomes recently published in the literature where hyper-heuristics got superior achievements (Li et al., 2019; Santiago Júnior et al., 2020; Carvalho et al., 2020). Moreover, the hyper-heuristic which stood out among all was HH-CF which is not also in line with some recent studies (Li et al., 2019).

We now show some Solution Fronts[13] for the configuration less in Figure 7 where we see one front of an algorithm related to the LOF4 problem instance. In each subfigure we show the True Known Pareto Front, the Solution Fronts of SPEA2 (the best algorithm in accordance with the MMI), and one of the other algorithms (from the second to the fifth best approaches according to the MMI). Moreover, we show three out of the four objective functions: edge coverage (x axis; Obj 4), test case diversity via Gower-Legendre (dice) similarity measure (y axis; Obj 2), and test case diversity via Sokal-Sneath (anti-dice) similarity measure (z axis; Obj 3). Note the very good performance of SPEA2, where the elements of its Solution Front overlap many elements of the True Known Pareto Front.

[13]The final population obtained by the algorithms is also known as the Solution Set. The Solution Front lies in the objective space, and it is composed of the values of the objective functions of the elements (solutions) in the Solution Set.



(a) True Known, NSGA-II, SPEA2

(b) True Known, HH-CF, SPEA2

(c) True Known, HRISE_M, SPEA2

(d) True Known, HRISE_R, SPEA2

Figure 7: True Known Pareto and Solution Fronts, configuration less, best algorithms: LOF4

5.2. Configuration More

Table 12 presents the results of the cross-domain analysis for the configuration more where we see only the average values of all problem instances and considering all normalised quality indicators: $\overline{h_N^{APR}}$, $\overline{\epsilon_N^{APR}}$, $\overline{I_N^{APR}}$, $\overline{\Delta_N^{*APR}}$, and $\overline{I{+}_N^{APR}}$. HH-RILA, NSGA-II, and IBEA were the first three regarding h_N and $I{+}_N$. NSGA-II, SPEA2, and HH-CF, in this order, were the best regarding ϵ_N and I_N while IBEA, HH-ALL, and NSGA-II were the top three algorithms regarding Δ_N^*.

Table 13 presents the results according to the statistical evaluation. There is a total agreement between the statistical results and the cross-domain analysis in terms of h, ϵ, I, and Δ^*. In other words, regarding these four quality indicators, the classification of the three best approaches in the statistical evaluation is the same as presented in the cross-domain analysis. In terms of $I+$, there is a tie between HH-RILA and NSGA-II (each obtained seven wins) while the cross-domain evaluation states that HH-RILA was better than NSGA-II. IBEA was the third in this case in total agreement with the cross-domain analysis. Overall, we may say that both evaluations produced

Table 12: Cross-domain analysis for all problem instances, configuration more: h_N, ϵ_N, I_N, Δ^*_N, $I+_N$

	HRISE_M	HRISE_R	HRMA	IBEA	NSGA-II	SPEA2	HH-ALL	HH-CF	HH-RILA
$\overline{h_N^{APR}}$	0.20759950	0.21055286	0.21042155	*0.10067897*	0.0952755207	0.1290253992	0.39939285	0.1304313727	**0.086522291**
$\overline{\epsilon_N^{APR}}$	0.2092056	0.2083920	0.2092787	0.27205945	**0.09545690**	0.12448931	0.3973271	*0.12635823*	0.16223275
$\overline{I_N^{APR}}$	0.11184677	0.11216365	0.11441431	0.20604696	**0.045701966**	0.05701937	0.27312140	*0.06034831*	0.09080320
$\overline{\Delta_N^{*APR}}$	0.5894822	0.5863466	0.5890051	**0.3985050**	*0.5306087*	0.5713666	0.4769654	0.5332393	0.5729960
$\overline{I+_N^{APR}}$	0.10525401	0.10720828	0.10735785	*0.056038525*	0.0492990658	0.059771106	0.24269769	0.0631368209	**0.044471428**

very similar results.

Table 13: Statistical evaluation, configuration more. Caption: Comp = Comparison

Comp	$>$	$\sim$	$<$	$>$	$\sim$	$<$	$>$	$\sim$	$<$	$>$	$\sim$	$<$	$>$	$\sim$	$<$
	h			ϵ			I			Δ^*			$I+$		
HRISE_M × HRISE_R	0	31	1	0	32	0	0	32	0	0	32	0	0	31	1
HRISE_M × HRMA	0	32	0	0	32	0	0	32	0	1	31	0	1	31	0
HRISE_M × IBEA	2	0	30	19	11	2	28	4	0	0	3	29	2	0	30
HRISE_M × NSGA-II	0	3	29	0	2	30	0	2	30	1	15	16	0	2	30
HRISE_M × SPEA2	0	6	26	0	3	29	0	4	28	1	28	3	0	4	28
HRISE_M × HH-ALL	27	5	0	27	4	1	28	3	1	0	12	20	25	7	0
HRISE_M × HH-CF	0	6	26	0	6	26	0	4	28	0	23	9	0	6	26
HRISE_M × HH-RILA	0	1	31	1	15	16	2	17	13	0	27	5	0	1	31
HRISE_R × HRMA	1	31	0	1	31	0	0	32	0	1	31	0	0	32	0
HRISE_R × IBEA	2	0	30	20	8	4	28	4	0	0	3	29	2	0	30
HRISE_R × NSGA-II	0	2	30	0	1	31	0	2	30	1	18	13	0	2	30
HRISE_R × SPEA2	0	4	28	0	0	32	0	2	30	2	27	3	0	1	31
HRISE_R × HH-ALL	27	5	0	27	4	1	28	3	1	0	14	18	25	7	0
HRISE_R × HH-CF	0	2	30	0	3	29	0	4	28	0	21	11	0	4	28
HRISE_R × HH-RILA	0	0	32	1	14	17	1	20	11	1	28	3	0	0	32
HRMA × IBEA	2	0	30	19	9	4	27	5	0	0	3	29	2	0	30
HRMA × NSGA-II	0	1	31	0	1	31	0	1	31	1	16	15	0	1	31
HRMA × SPEA2	0	5	27	0	2	30	0	3	29	2	27	3	0	3	29
HRMA × HH-ALL	27	5	0	27	4	1	26	5	1	0	14	18	26	6	0
HRMA × HH-CF	0	2	30	0	4	28	0	2	30	0	21	11	0	3	29
HRMA × HH-RILA	0	0	32	2	13	17	2	17	13	0	27	5	0	0	32
IBEA × NSGA-II	4	19	9	1	0	31	0	0	32	28	0	4	4	18	10
IBEA × SPEA2	22	7	3	1	1	30	0	0	32	26	3	3	14	16	2
IBEA × HH-ALL	30	1	1	23	7	2	18	12	2	9	22	1	30	1	1
IBEA × HH-CF	19	11	2	1	2	29	0	0	32	24	4	4	13	17	2
IBEA × HH-RILA	0	26	6	1	3	28	0	0	32	28	3	1	2	24	6
NSGA-II × SPEA2	24	6	2	16	14	2	15	15	2	7	23	2	22	8	2
NSGA-II × HH-ALL	32	0	0	32	0	0	31	1	0	2	23	7	30	2	0
NSGA-II × HH-CF	23	9	0	13	19	0	18	13	1	3	28	1	22	9	1
NSGA-II × HH-RILA	2	27	3	23	6	3	29	1	2	9	23	0	3	26	3
SPEA2 × HH-ALL	32	0	0	32	0	0	32	0	0	3	18	11	32	0	0
SPEA2 × HH-CF	3	28	1	2	29	1	3	28	1	1	27	4	2	29	1
SPEA2 × HH-RILA	0	11	21	14	17	1	18	14	0	4	26	2	2	13	17
HH-ALL × HH-CF	0	0	32	0	0	32	0	0	32	7	22	3	0	0	32
HH-ALL × HH-RILA	0	1	31	0	1	31	0	1	31	15	17	0	0	1	31
HH-CF × HH-RILA	0	7	25	17	14	1	17	13	2	5	27	0	2	10	20

Results of the MMI for the configuration more are in Table 14. As in the previous section, the MMI outcomes present differences comparing to the previous evaluations. NSGA-II was the best according to this metric followed by HH-CF and IBEA. Furthermore, even if HH-RILA obtained two top positions (h_N, $I+_N$) in the cross-domain analysis, not only IBEA but also SPEA2 were here overall better than HH-RILA. Particularly, note that IBEA was the best considering Δ^*_N and got two third positions (h_N, $I+_N$). On the other hand, SPEA2 obtained two second best performances (ϵ_N, I_N). The same explanations given in the previous section help to explain

these differences: combination of all quality indicators, consistent performance of the algorithms, three-level normalisation, not case-by-case analysis. However, here HH-RILA was superior than HRISE_R, HRISE_M, and HRMA.

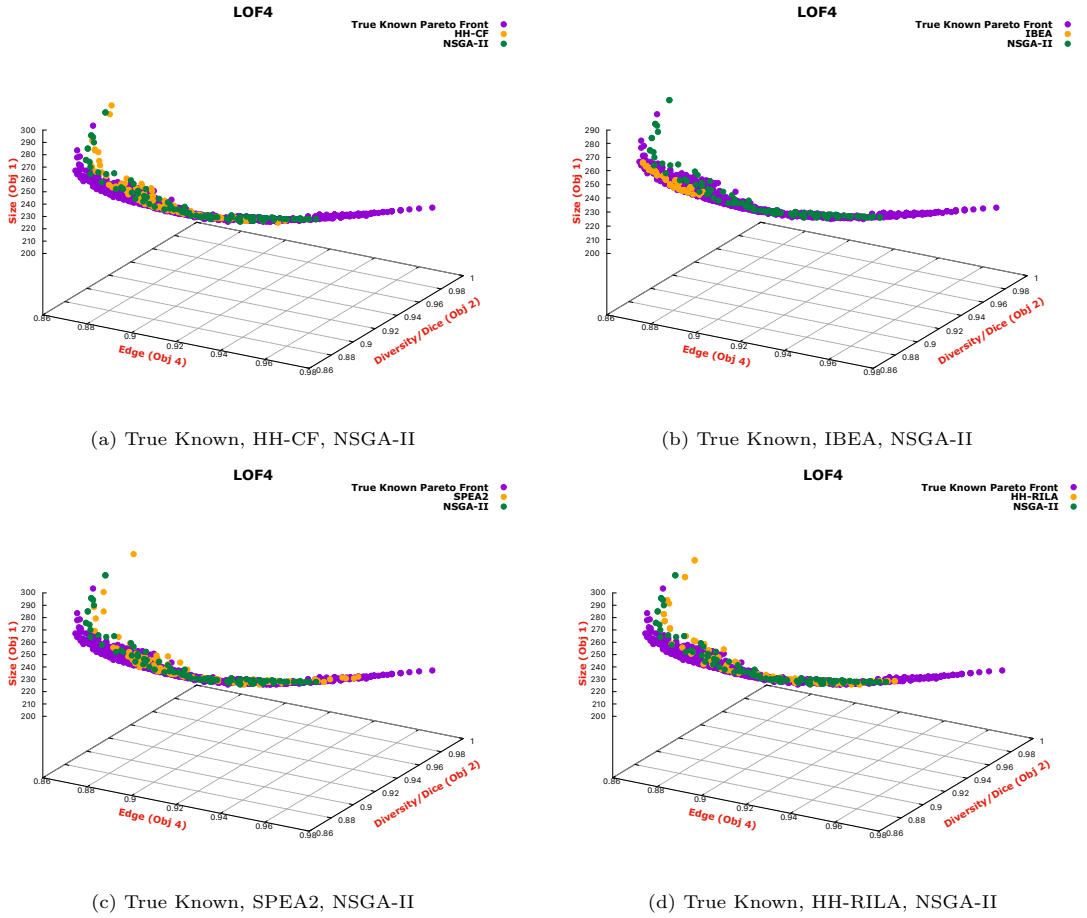
Table 14: MMI, configuration more

Ranking	Algorithm	Euclidean Distance
1	NSGA-II	0.692718771610298
2	HH-CF	0.735471598187579
3	IBEA	0.919143040464169
4	SPEA2	0.924889941707152
5	HH-RILA	0.960761946867152
6	HRISE_R	1.20421090445104
7	HRISE_M	1.21258602154972
8	HRMA	1.21898973079064
9	HH-ALL	2.04176066439771

Answering RQ_2 which relates to the configuration more, we may say that the metaheuristics, particularly NSGA-II, obtained in general better performances than the hyper-heuristics, although HH-CF was now the second best according to the MMI. HH-RILA demonstrated a better performance than in the configuration less but even so this was not enough to beat IBEA and SPEA2.

We show again some Solution Fronts generated for the configuration more in Figure 8. Once more, we show one front of an algorithm related to the LOF4 problem instance. As earlier, in each subfigure we show the True Known Pareto Front, the Solution Fronts of NSGA-II (the best algorithm in accordance with the MMI), and one of the other algorithms (from the second to the fifth best approaches according to the MMI). We show three out of the four objective functions: edge coverage (x axis; Obj 4), test case diversity via Gower-Legendre (dice) similarity measure (y axis; Obj 2), and size of the test suites (z axis; Obj 1). Again, observe the very good performance of NSGA-II, the best algorithm.

As for RQ_3, we can not clearly say that there is a single algorithm that is highly superior than the others considering both configurations. Overall, even if the metaheuristics obtained better performances, in particular NSGA-II, they are not clearly superior. To state this claim, an algorithm must be better than the others in most of the quality indicators in the context of the cross-domain analysis, as well as it must obtain more wins in most of the indicators regarding the statistical evaluation, and it must be the best in accordance with the MMI indicator. In addition, this should consistently happen for both configurations. We have no such algorithm according to the results. This response just corroborates the fact that generalisation is really difficult to achieve regarding optimisation problems. Note that this conclusion, based on empirical evidences, seems to agree with the no free lunch theorem since "old" metaheuristics which performed worse than hyper-heuristics solving continuous optimisation problems, now were better addressing discrete



(a) True Known, HH-CF, NSGA-II

(b) True Known, IBEA, NSGA-II

(c) True Known, SPEA2, NSGA-II

(d) True Known, HH-RILA, NSGA-II

Figure 8: True Known Pareto and Solution Fronts, configuration more, best algorithms: LOF4

optimisation ones.

6. Discussion

In this section, we discuss the results we obtained in more detail and provide general recommendations to practitioners when facing the testing of GUI applications. Firstly, we can accept the hypothesis we have formulated in Section 1 where we have stated that it is possible to combine SBST with other traditional type of testing adopted for GUI applications, so that we can generate test cases according to the many-objective perspective. We believe that the case studies (problem instances) we considered are interesting where in one type of problem we derive test cases based on the source of a non-trivial software product, and the other problem is a completely different perspective, via use cases. Additional remarks and recommendations are mentioned below.

6.1. State Space Explosion

State space explosion is a well-known problem related to any model-based test case generation strategy. In other words, if the model has many states (vertices in a graph) and/or transitions

(edges in a graph), test case criteria may derive not only extremely lengthier test cases but also test suites with an enormous amount of test cases. We have addressed this issue by limiting the number of test cases a test suite may have, by adding constraints which automatically delimit the length of the test cases based on the number of vertices of the EFG and hence avoid creating meaningless test cases in practical terms, and also by defining configurations (less, more) which limit the maximum number of simple circuits to consider. All these decisions favour the generation of test cases in a timely manner, addressing the state space explosion issue.

Another remark refers to the selected test criterion to generate test cases: simple circuits. Note that this is equivalent to the *all-simple-paths* test criterion for Statechart-based testing (Santiago Júnior, 2011; Santiago Júnior & Vijaykumar, 2012; Souza, 2000), which requires that all simple paths (simple circuits) are traversed at least once by a test suite. As we have just explained above, the decisions we have made relax this condition that all simple circuits must be traversed to address the state space explosion problem. However, we may suggest other test criteria which are in a higher-level position in the inclusion (hierarchy) relation among Statechart test criteria which can be adapted to EFG models. For instance, practitioners may consider *all-paths-k-C0-configuration* and *all-paths-k-configurations* as test criteria and perceive the benefits in their practical settings.

6.2. Model, Test Suite and Test Case

We decided to rely on EFGs to generate test cases. Note that EFG is not suitable for all types of GUI applications. However, the way we defined the test suite and test case allowed us to generate test cases based on the source code of GUI desktop applications (code-driven problem) and also via use cases (use case-driven problem). Since each decision variable of a solution is an integer which identifies a test case which in its turn is a simple circuit of the graph, we do not have problems of inconsistent test cases, since all of them are already consistent.

However, this does not prevent that other more sophisticated models, such as ESIG (Yuan & Memon, 2010a), from being evaluated, in combination with the test suite and test case decisions we have adopted. Recall that our definitions of test suite and test case provide more flexibility to the professional to select a test suite according to his/her preference, and also having longer but not extremely lengthier test cases.

6.3. Choice of Algorithms and Assessments

It is naturally important to choose the most adequate algorithm to generate GUI test cases. The whole point of different types of assessments is giving several perspectives so that a testing professional can decide which is the most appropriate approach. We just need to emphasise that traditional, or eventually considered "outdated", algorithms are still worth being investigated. As we have mentioned in the previous section, there is no single algorithm that is highly superior than

all the others but, considering all the evaluations we performed, NSGA-II, a metaheuristic proposed quite some time ago, would be a natural preference for the problem instances we considered.

Observing the results for both configurations less and more, the three types of analyses (cross-domain, statistical, MMI) do not totally agree in their rankings. As we have pointed out, cross-domain and MMI are assessments that do not take into account a case-by-case analysis but rather they are performed across all problem instances of all problems. On the other hand, the statistical evaluation is indeed a case-by-case assessment. Hence, our recommendation is that if a practitioner is solving a single and specific problem instance, the statistical evaluation is a better tool to rely on. However, if the goal is to address several optimisation problem instances from many real-world problems, cross-domain and the MMI are the approaches to go for. Particularly, the MMI indicator we have proposed in this work is a valuable metric and it can be easily extended to n quality indicators.

Depending on the selected quality indicators, the conclusions could be different. Let us recall the configuration less where even if NSGA-II was better than SPEA2 with respect to four out of the five quality indicators in the cross-domain analysis (h_N, ϵ_N, I_N, and $I+_N$), SPEA2 got superior performance regarding Δ^*_N. This better performance under Δ^*_N and consistent results considering the other four quality indicators were sufficient, and SPEA2 overcame NSGA-II regarding the MMI. If we did not consider Δ^*_N, there is no doubt that NSGA-II would get a better MMI value than SPEA2. We decided to rely on five quality indicators to have a broader view where these metrics cover three categories: convergence-diversity, convergence, and diversity.

6.4. Many-Objective Test Case Generation including other Non-Functional Requirements

We believe that the many-objective perspective is really important to generate test cases for non-trivial applications. The motivation is to create test cases that are, by design, already suitable according to different test objectives.

We considered two non-functional properties, effectiveness and cost, of the created test suites. Thus, it is truly relevant to consider other non-functional requirements related to the GUI applications themselves such as performance, security, safety, in addition to functional requirements and non-functional properties that we have already addressed. This is a valuable unified direction where we try to create test cases aiming at detecting different classes of defects.

7. Conclusions

This study showed how to combine search-based with model-based testing to generate test cases for GUI applications taking into account the many-objective perspective. However, this combination allows the generation of test cases for any software system whose structure and/or behaviour can be represented by a sort of graph (e.g. EFG).

Our main motivation was the fact that, in spite of having several previous studies supported by tools in the literature for GUI test case generation, the majority of these approaches generate test cases based on or addressing functional properties only. Hence, we found the need to propose a strategy where the many-objective perspective is satisfied. In our evaluation, in addition to considering five quality indicators, we also proposed a new multi-metric measure, MMI, in which all these quality indicators contribute to present a unified response.

Several previous studies in the literature show superior achievements of selection hyper-heuristics over metaheuristics (Santiago Júnior et al., 2020; Santiago Júnior & Özcan, 2019; Maashi et al., 2014; Li et al., 2019; McClymont et al., 2012; Li et al., 2017; Carvalho et al., 2020; Guizzo et al., 2017; Ferreira et al., 2017). However, based on three different assessments, i.e. cross-domain analysis, statistical evaluation, and MMI, our findings show that the metaheuristics, particularly NSGA-II which may be considered the best overall, produced better results for GUI test case generation than the hyper-heuristics. However, the HRISE family, HH-RILA, and HH-CF were all originally designed and applied to continuous optimisation problems. In this article, we considered discrete optimisation problems, which might have caused the performance differences deviating from what was reported previously.

Another surprisingly conclusion is that among all hyper-heuristics, HH-CF was the best overall surpassing the recently proposed algorithms of the HRISE family (Santiago Júnior et al., 2020; Santiago Júnior & Özcan, 2019) and HH-RILA (Li et al., 2019). HH-CF was even the second best approach in the configuration more according to the MMI. While some studies (Santiago Júnior et al., 2020) have already mentioned the good performance of HH-CF for some continuous problems (e.g. Deb-Thiele-Laumanns-Zitzler (DTLZ) (Deb et al., 2005)) others do not agree with this conclusion and HH-CF was not that good (Li et al., 2019).

Note that HRISE_M, HRISE_R, and HH-RILA are all reinforcement learning-based algorithms. It is possible that the learning rate/parameters of such approaches were inadequate considering the landscapes of the discrete problems and this can be another explanation for their relative low performances. In all algorithms to obtain the final population, we considered only the non-dominated solutions. We then noticed that the final populations of HRISE_M, HRISE_R, HRMA were, in general, considerably smaller than the other algorithms for basically all problem instances (recall that the initial population was randomly created with 100 solutions for all algorithms). A smaller population can lead to poor performances with respect to the metrics. It is not completely clear why such proportionally small populations were devised by the HRISE family. Probably, the two-level move acceptance mechanism is too demanding for these problems but we need further investigations to clarify this point.

The first future direction is to design and execute a controlled experiment where we can evaluate the algorithms under the point of view of the testing community. Hence, we need to execute the

created test suites and realise which of the algorithms is the best based on their ability of detection of defects. Therefore, we may empirically corroborate our claims regarding the many-objective perspective. We also intend to include other non-functional requirements as objective functions in order to propose a unified test case generation approach for GUI applications. More GUI case studies will be considered in order to have an even wider range of problem instances. Thus, we will have more experimental ground to conclude better about generalisation. In terms of the HRISE family, we can relax the two-level (hierarchical) move acceptance mechanism and perceive the influence of such modification in the final results. We also intend to embed many-objective evolutionary algorithms as they are known, such as the Reference Vector Guided Evolutionary Algorithm (RVEA) (Cheng et al., 2016), into the selection hyper-heuristics and thus we can perform experimental evaluations considering such new versions of the hyper-heuristics.

Acknowledgements

This research was supported by *Fundação de Amparo à Pesquisa do Estado de São Paulo* (FAPESP), Brazil, process number: 2018/08372-8.

References

Amalfitano, D., Riccio, V., Amatucci, N., Simone, V. D., & Fasolino, A. R. (2019). Combining automated gui exploration of android apps with capture and replay through machine learning. *Information and Software Technology*, *105*, 95 – 116. URL: http://www.sciencedirect.com/science/article/pii/S0950584918301708. doi:https://doi.org/10.1016/j.infsof.2018.08.007.

Ariss, O. E., Xu, D., Dandey, S., Vender, B., McClean, P., & Slator, B. (2010). A systematic capture and replay strategy for testing complex gui based java applications. In *2010 Seventh International Conference on Information Technology: New Generations* (pp. 1038–1043). doi:10.1109/ITNG.2010.216.

Arlt, S., Bertolini, C., & Schäf, M. (2011). Behind the scenes: An approach to incorporate context in gui test case generation. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops* (pp. 222–231). doi:10.1109/ICSTW.2011.70.

Balera, J. M., & Santiago Júnior, V. A. (2019). A systematic mapping addressing hyper-heuristics within search-based software testing. *Information and Software Technology*, *114*, 176 – 189. URL: http://www.sciencedirect.com/science/article/pii/S0950584919301430. doi:https://doi.org/10.1016/j.infsof.2019.06.012.

Banerjee, I., Nguyen, B., Garousi, V., & Memon, A. (2013). Graphical user interface (gui) testing: Systematic mapping and repository. *Inf. Softw. Technol.*, *55*, 1679–1694. URL: http://dx.doi.org/10.1016/j.infsof.2013.03.004. doi:10.1016/j.infsof.2013.03.004.

Bauersfeld, S., Wappler, S., & Wegener, J. (2011a). A Metaheuristic Approach to Test Sequence Generation for Applications with a GUI. In M. B. Cohen, & M. O. Cinnéide (Eds.), *Search Based Software Engineering* (pp. 173–187). Berlin, Heidelberg: Springer Berlin Heidelberg.

Bauersfeld, S., Wappler, S., & Wegener, J. (2011b). An Approach to Automatic Input Sequence Generation for GUI Testing Using Ant Colony Optimization. In *13th Annual Conference Companion on Genetic and Evolutionary Computation* GECCO '11 (pp. 251–252). New York, NY, USA: ACM. URL: http://doi.acm.org/10.1145/2001858.2001999. doi:10.1145/2001858.2001999.

Belli, F., Beyazit, M., Budnik, C. J., & Tuglular, T. (2017). Chapter five - advances in model-based testing of graphical user interfaces. In A. M. Memon (Ed.), *Advances in Computers* (pp. 219 – 280). Elsevier volume 107 of *Advances in Computers*. URL: http://www.sciencedirect.com/science/article/pii/S006524581730030X. doi:https://doi.org/10.1016/bs.adcom.2017.06.004.

Bures, M., Macik, M., Ahmed, B. S., Rechtberger, V., & Slavik, P. (2020). Testing the usability and accessibility of smart tv applications using an automated model-based approach. *IEEE Transactions on Consumer Electronics*, *66*, 134–143. doi:10.1109/TCE.2020.2986049.

Burke, E. K., Hyde, M. R., Kendall, G., Ochoa, G., Özcan, E., & Woodward, J. R. (2019). A classification of hyper-heuristic approaches: Revisited. In M. Gendreau, & J.-Y. Potvin (Eds.), *Handbook of Metaheuristics* (pp. 453–477). Cham: Springer International Publishing. URL: https://doi.org/10.1007/978-3-319-91086-4_14. doi:10.1007/978-3-319-91086-4_14.

Carvalho, V. R., Larson, K., Brandão, A. A. F., & Sichman, J. S. (2020). Applying social choice theory to solve engineering multi-objective optimization problems. *Journal of Control, Automation and Electrical Systems*, *31*, 119–128. doi:10.1007/s40313-019-00526-2.

Cheng, R., Jin, Y., Olhofer, M., & Sendhoff, B. (2016). A reference vector guided evolutionary algorithm for many-objective optimization. *IEEE Transactions on Evolutionary Computation*, *20*, 773–791. doi:10.1109/TEVC.2016.2519378.

Chinnapongse, V., Lee, I., Sokolsky, O., Wang, S., & Jones, P. L. (2009). Model-based testing of gui-driven applications. In S. Lee, & P. Narasimhan (Eds.), *Software Technologies for Embedded and Ubiquitous Systems* (pp. 203–214). Berlin, Heidelberg: Springer Berlin Heidelberg.

Corne, D. W., Jerram, N. R., Knowles, J. D., & Oates, M. J. (2001). PESA-II: Region-based selection in evolutionary multiobjective optimization. In *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation* GECCO'01 (pp. 283–290). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Deb, K., & Agrawal, R. B. (1995). Simulated binary crossover for continuous search space. *Complex Systems*, *9*, 115–148.

Deb, K., Pratap, A., Agarwal, S., & Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, *6*, 182–197. doi:10.1109/4235.996017.

Deb, K., Thiele, L., Laumanns, M., & Zitzler, E. (2005). Scalable test problems for evolutionary multiobjective optimization. In A. Abraham, L. Jain, & R. Goldberg (Eds.), *Evolutionary Multiobjective Optimization: Theoretical Advances and Applications* (pp. 105–145). London: Springer London. URL: https://doi.org/10.1007/1-84628-137-7_6. doi:10.1007/1-84628-137-7_6.

Dokeroglu, T., Sevinc, E., Kucukyilmaz, T., & Cosar, A. (2019). A survey on new generation metaheuristic algorithms. *Computers & Industrial Engineering*, *137*, 106040. URL: http://www.sciencedirect.com/science/article/pii/S0360835219304991. doi:https://doi.org/10.1016/j.cie.2019.106040.

Drake, J. H., Kheiri, A., Özcan, E., & Burke, E. K. (2019). Recent advances in selection hyper-heuristics. *European Journal of Operational Research*, . URL: http://www.sciencedirect.com/science/article/pii/S0377221719306526. doi:https://doi.org/10.1016/j.ejor.2019.07.073. . In Press (Available Online: Aug 7, 2019).

Durillo, J. J., & Nebro, A. J. (2011). jMetal: A java framework for multi-objective optimization. *Advances in Engineering Software*, *42*, 760 – 771. URL: http://www.sciencedirect.com/science/article/pii/S0965997811001219. doi:https://doi.org/10.1016/j.advengsoft.2011.05.014.

Eras, E. R., Santiago Júnior, V. A., & Santos, L. B. R. (2019). Singularity: A methodology for automatic unit test data generation for C++ applications based on model checking counterexamples. In *Proceedings of the IV Brazilian Symposium on Systematic and Automated Software Testing* SAST 2019 (pp. 72–79). New York, NY, USA: Association for Computing Machinery. URL: https://doi.org/10.1145/3356317.3356319. doi:10.1145/3356317.3356319.

Farto, G. C., & Endo, A. T. (2017). Reuse of model-based tests in mobile apps. In *Proceedings of the 31st Brazilian Symposium on Software Engineering* SBES'17 (pp. 184–193). New York,

NY, USA: Association for Computing Machinery. URL: https://doi.org/10.1145/3131151.3131160. doi:10.1145/3131151.3131160.

Ferreira, J. C., Fonseca, C. M., & Gaspar-Cunha, A. (2007). Methodology to select solutions from the pareto-optimal set: A comparative study. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation* GECCO '07 (pp. 789–796). New York, NY, USA: Association for Computing Machinery. URL: https://doi.org/10.1145/1276958.1277117. doi:10.1145/1276958.1277117.

Ferreira, T. N., Lima, J. A. P., Strickler, A., Kuk, J. N., Vergilio, S. R., & Pozo, A. (2017). Hyper-heuristic based product selection for software product line testing. *IEEE Computational Intelligence Magazine*, *12*, 34–45. doi:10.1109/MCI.2017.2670461.

Fonseca, C. M., & Fleming, P. J. (1998). Multiobjective optimization and multiple constraint handling with evolutionary algorithms. i. a unified formulation. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, *28*, 26–37. doi:10.1109/3468.650319.

Gómez, R. H., & Coello, C. A. C. (2015). Improved metaheuristic based on the r2 indicator for many-objective optimization. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation* GECCO '15 (pp. 679–686). New York, NY, USA: ACM. URL: http://doi.acm.org/10.1145/2739480.2754776. doi:10.1145/2739480.2754776.

Gove, R., & Faytong, J. (2011). Identifying infeasible gui test cases using support vector machines and induced grammars. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops* (pp. 202–211). doi:10.1109/ICSTW.2011.73.

Guizzo, G., Vergilio, S. R., Pozo, A. T. R., & Fritsche, G. M. (2017). A multi-objective and evolutionary hyper-heuristic applied to the integration and test order problem. *Applied Soft Computing*, *56*, 331 – 344. URL: http://www.sciencedirect.com/science/article/pii/S1568494617301357. doi:https://doi.org/10.1016/j.asoc.2017.03.012.

Harman, M., Jia, Y., & Zhang, Y. (2015). Achievements, open problems and challenges for search based software testing. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)* (pp. 1–12). doi:10.1109/ICST.2015.7102580.

Hawick, K. A., & James, H. A. (2008). Enumerating circuits and loops in graphs with self-arcs and multiple-arcs. In *Proc. 2008 Int. Conf. on Foundations of Computer Science (FCS'08)* (pp. 14–20). Las Vegas, USA: CSREA.

Hemmati, H., Arcuri, A., & Briand, L. (2013). Achieving scalable model-based testing through test case diversity. *ACM Trans. Softw. Eng. Methodol.*, *22*, 6:1–6:42. URL: http://doi.acm.org/10.1145/2430536.2430540. doi:10.1145/2430536.2430540.

Herbold, S., Bünting, U., Grabowski, J., & Waack, S. (2012). Deployable capture/replay supported by internal messages. In A. Memon (Ed.), *Advances in Computers* (pp. 327 – 367). Elsevier volume 85 of *Advances in Computers*. URL: http://www.sciencedirect.com/science/article/pii/B9780123965264000072. doi:https://doi.org/10.1016/B978-0-12-396526-4.00007-2.

Herbold, S., Grabowski, J., Waack, S., & Bünting, U. (2011). Improved bug reporting and reproduction through non-intrusive gui usage monitoring and automated replaying. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops* (pp. 232–241). doi:10.1109/ICSTW.2011.66.

Hothorn, T., Hornik, K., van de Wiel, M., & Zeileis, A. (2008). Implementing a class of permutation tests: The coin package. *Journal of Statistical Software*, *28*, 1–23. URL: https://www.jstatsoft.org/v028/i08. doi:10.18637/jss.v028.i08.

Huang, S., Cohen, M. B., & Memon, A. M. (2010). Repairing gui test suites using a genetic algorithm. In *2010 Third International Conference on Software Testing, Verification and Validation* (pp. 245–254). doi:10.1109/ICST.2010.39.

Ishibuchi, H., Masuda, H., & Nojima, Y. (2015). A study on performance evaluation ability of a modified inverted generational distance indicator. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation* GECCO '15 (pp. 695–702). New York, NY, USA: ACM. URL: http://doi.acm.org/10.1145/2739480.2754792. doi:10.1145/2739480.2754792.

Jiang, S., Ong, Y., Zhang, J., & Feng, L. (2014). Consistencies and contradictions of performance metrics in multiobjective optimization. *IEEE Transactions on Cybernetics*, *44*, 2391–2404. doi:10.1109/TCYB.2014.2307319.

Khari, M., & Kumar, P. (2019). An extensive evaluation of search-based software testing: a review. *Soft Computing*, *23*, 1933–1946. URL: https://doi.org/10.1007/s00500-017-2906-y. doi:10.1007/s00500-017-2906-y.

Latiu, G., Cret, O., & Vacariu, L. (2013a). Graphical User Interface Testing Optimization for Water Monitoring Applications. In *2013 19th International Conference on Control Systems and Computer Science* (pp. 640–645). doi:10.1109/CSCS.2013.32.

Latiu, G. I., Cret, O., & Vacariu, L. (2013b). Graphical user interface testing using evolutionary algorithms. In *2013 8th Iberian Conference on Information Systems and Technologies (CISTI)* (pp. 1–6).

Li, H., & Zhang, Q. (2009). Multiobjective optimization problems with complicated pareto sets, moea/d and nsga-ii. *IEEE Transactions on Evolutionary Computation*, *13*, 284–302. doi:10.1109/TEVC.2008.925798.

Li, W., Özcan, E., & John, R. (2017). Multi-objective evolutionary algorithms and hyper-heuristics for wind farm layout optimisation. *Renewable Energy*, *105*, 473 – 482. URL: http://www.sciencedirect.com/science/article/pii/S0960148116310709. doi:https://doi.org/10.1016/j.renene.2016.12.022.

Li, W., Özcan, E., & John, R. (2019). A learning automata-based multiobjective hyper-heuristic. *IEEE Transactions on Evolutionary Computation*, *23*, 59–73. doi:10.1109/TEVC.2017.2785346.

Maashi, M., Özcan, E., & Kendall, G. (2014). A multi-objective hyper-heuristic based on choice function. *Expert Systems with Applications*, *41*, 4475 – 4493. URL: http://www.sciencedirect.com/science/article/pii/S095741741400013X. doi:https://doi.org/10.1016/j.eswa.2013.12.050.

Mahmood, R., Mirzaei, N., & Malek, S. (2014). EvoDroid: Segmented Evolutionary Testing of Android Apps. In *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* FSE 2014 (pp. 599–609). New York, NY, USA: ACM. URL: http://doi.acm.org/10.1145/2635868.2635896. doi:10.1145/2635868.2635896.

McClymont, K., Keedwell, E., Savić, D., & Randall-Smith, M. (2012). A general multi-objective hyper-heuristic for water distribution network design with discolouration risk. *Journal of Hydroinformatics*, *15*, 700–716. URL: https://doi.org/10.2166/hydro.2012.022. doi:10.2166/hydro.2012.022.

McMinn, P. (2004). Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, *14*, 105–156.

Memon, A. M., & Nguyen, B. N. (2010). Advances in automated model-based system testing of software applications with a gui front-end. In M. V. Zelkowitz (Ed.), *Advances in Computers* (pp. 121–162). Elsevier.

Menninghaus, M., Wilke, F., Schleutker, J.-P., & Pulvermüller, E. (2017). Search based GUI Test Generation in Java - Comparing Code-based and EFG-based Optimization Goals. In *12th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2017)* (pp. 179–186).

Mohanta, B. K., Jena, D., Satapathy, U., & Patnaik, S. (2020). Survey on iot security: Challenges and solution using machine learning, artificial intelligence and blockchain technology.

Internet of Things, *11*, 100227. URL: http://www.sciencedirect.com/science/article/pii/S2542660520300603. doi:https://doi.org/10.1016/j.iot.2020.100227.

Nguyen, B. N., Robbins, B., Banerjee, I., & Memon, A. (2014). Guitar: an innovative tool for automated testing of gui-driven software. *Automated Software Engineering*, *21*, 65–105. URL: https://doi.org/10.1007/s10515-013-0128-9. doi:10.1007/s10515-013-0128-9.

Rauf, A., Ejaz, N., Abbas, Q., Rehman, S. U., & Shahid, A. A. (2010). PSO based test coverage analysis for event driven software. In *The 2nd International Conference on Software Engineering and Data Mining* (pp. 219–224).

Rauf, A., & Ramzan, M. (2018). Parallel testing and coverage analysis for context-free applications. *Cluster Computing*, *21*, 729–739. URL: https://doi.org/10.1007/s10586-017-1000-7. doi:10.1007/s10586-017-1000-7.

Robinson, B., & Brooks, P. (2009). An initial study of customer-reported gui defects. In *2009 International Conference on Software Testing, Verification, and Validation Workshops* (pp. 267–274). doi:10.1109/ICSTW.2009.22.

Saeed, A., Hamid, S. H. A., & Mustafa, M. B. (2016). The experimental applications of search-based techniques for model-based testing: Taxonomy and systematic literature review. *Applied Soft Computing*, *49*, 1094 – 1117. URL: http://www.sciencedirect.com/science/article/pii/S1568494616304240. doi:https://doi.org/10.1016/j.asoc.2016.08.030.

Sahar, H., Bangash, A. A., & Beg, M. O. (2019). Towards energy aware object-oriented development of android applications. *Sustainable Computing: Informatics and Systems*, *21*, 28 – 46. URL: http://www.sciencedirect.com/science/article/pii/S2210537918302014. doi:https://doi.org/10.1016/j.suscom.2018.10.005.

Santiago, V., Vijaykumar, N. L., Guimarães, D., Amaral, A. S., & Ferreira, É. (2008). An environment for automated test case generation from Statechart-based and Finite State Machine-based behavioral models. In *2008 IEEE International Conference on Software Testing Verification and Validation Workshop* (pp. 63–72).

Santiago Júnior, V. A. (2011). *SOLIMVA: A methodology for generating model-based test cases from natural language requirements and detecting incompleteness in software specifications*. Ph.D. thesis Instituto Nacional de Pesquisas Espaciais (INPE). Available from: https://bit.ly/33uIJpg. Access in: Sept 14, 2020.

Santiago Júnior, V. A., & Özcan, E. (2019). HRMA: Hyper-heuristic based on the random choice of move acceptance methods. In *Proceedings of the Annual Conference of The Operational Research Society (OR61)* (pp. 37–38).

Santiago Júnior, V. A., Özcan, E., & Carvalho, V. R. (2020). Hyper-heuristics based on reinforcement learning, balanced heuristic selection and group decision acceptance. *Applied Soft Computing*, *97*, 1–23. URL: http://www.sciencedirect.com/science/article/pii/S1568494620306980. doi:https://doi.org/10.1016/j.asoc.2020.106760. Available online: Oct. 1, 2020.

Santiago Júnior, V. A., & Vijaykumar, N. L. (2012). Generating model-based test cases from natural language requirements for space application software. *Software Quality Journal*, *20*, 77–143. DOI: 10.1007/s11219-011-9155-6.

Seo, J., Choi, B., & Yang, S. (2012). Lightweight embedded software performance analysis method by kernel hack and its industrial field study. *Journal of Systems and Software*, *85*, 28 – 42. URL: http://www.sciencedirect.com/science/article/pii/S0164121211000781. doi:https://doi.org/10.1016/j.jss.2011.03.049. Dynamic Analysis and Testing of Embedded Software.

Silva, R. A., de Souza, S. R. S., & de Souza, P. S. L. (2017). A systematic review on search based mutation testing. *Information and Software Technology*, *81*, 19 – 35. URL: http://www.sciencedirect.com/science/article/pii/S0950584916300167. doi:https://doi.org/10.1016/j.infsof.2016.01.017.

Souza, S. R. S. (2000). *Validação de especificações de sistemas reativos: definição e análise de critérios de teste*. Ph.D. thesis Universidade de São Paulo. Available from: https://bit.ly/3mkulZt. Access in: Sept 14, 2020.

Tan, K., Lee, T., & Khor, E. (2002). Evolutionary algorithms for multi-objective optimization: Performance assessments and comparisons. *Artificial Intelligence Review*, *17*, 251–290. URL: https://doi.org/10.1023/A:1015516501242. doi:10.1023/A:1015516501242.

Wolpert, D., & Macready, W. (2005). Coevolutionary free lunches. *IEEE Transactions on Evolutionary Computation*, *9*, 721–735. doi:10.1109/TEVC.2005.856205.

Xie, Q., & Memon, A. M. (2008). Using a pilot study to derive a gui model for automated testing. *ACM Trans. Softw. Eng. Methodol.*, *18*. URL: https://doi.org/10.1145/1416563.1416567. doi:10.1145/1416563.1416567.

Yang, W., Chen, Z., Gao, Z., Zou, Y., & Xu, X. (2014). Gui testing assisted by human knowledge: Random vs. functional. *Journal of Systems and Software*, *89*, 76 – 86. URL: http://www.sciencedirect.com/science/article/pii/S0164121213002392. doi:https://doi.org/10.1016/j.jss.2013.09.043.

Yuan, X., & Memon, A. M. (2010a). Generating event sequence-based test cases using gui runtime state feedback. *IEEE Transactions on Software Engineering*, *36*, 81–95. doi:10.1109/TSE.2009.68.

Yuan, X., & Memon, A. M. (2010b). Iterative execution-feedback model-directed gui testing. *Information and Software Technology*, *52*, 559 – 575. URL: http://www.sciencedirect.com/science/article/pii/S0950584909002092. doi:https://doi.org/10.1016/j.infsof.2009.11.009. TAIC-PART 2008.

Zhou, A., Jin, Y., Zhang, Q., Sendhoff, B., & Tsang, E. (2006). Combining model-based and genetics-based offspring generation for multi-objective optimization using a convergence criterion. In *2006 IEEE International Conference on Evolutionary Computation* (pp. 892–899). doi:10.1109/CEC.2006.1688406.

Zitzler, E., & Künzli, S. (2004). Indicator-based selection in multiobjective search. In X. Yao, E. K. Burke, J. A. Lozano, J. Smith, J. J. Merelo-Guervós, J. A. Bullinaria, J. E. Rowe, P. Tiňo, A. Kabán, & H.-P. Schwefel (Eds.), *Parallel Problem Solving from Nature - PPSN VIII* (pp. 832–842). Berlin, Heidelberg: Springer Berlin Heidelberg.

Zitzler, E., Laumanns, M., & Thiele, L. (2001). *SPEA2: Improving the Performance of the Strength Pareto Evolutionary Algorithm*. Technical Report Computer Engineering and Communication Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich.

Zitzler, E., & Thiele, L. (1999). Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *IEEE Transactions on Evolutionary Computation*, *3*, 257–271. doi:10.1109/4235.797969.

Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C. M., & da Fonseca, V. G. (2003). Performance assessment of multiobjective optimizers: an analysis and review. *IEEE Transactions on Evolutionary Computation*, *7*, 117–132. doi:10.1109/TEVC.2003.810758.