Czech Technical University in Prague
Faculty of Electrical Engineering

# Model-based Software Test Automation

## Habilitation Thesis

Miroslav Bures

Department of Computer Science
Karlovo nam. 13, 121 35 Praha 2
Czech Republic

# Preface

This thesis presents my selected work for the period from 2013 to 2017 and is part of my application for the title of Associate Professor at Czech Technical University in Prague. My research has focused on automated software testing, namely, automating the creation of test scenarios and their execution. The motivation for selecting this area arises from the situation in the software industry. The increasing complexity of contemporary software information systems, together with the demand to shorten the time-to-market and to decrease the development prices of information systems create numerous challenges for the quality assurance of the software development process. The latest trends in this field imply demand for more effective and reliable software testing methods, employing automation and model-based approaches. In the thesis, I present three research tracks, namely path-based test case generation strategies, combinatorial and constrained interaction testing and test automation frameworks for sub-optimally structured software development projects. In these tracks, I identify six research challenges; then, I introduce the projects I am leading or have participated in that address these challenges. This work is documented by six papers and articles, that are included in the appendix of this thesis.

# Acknowledgments

I would like to thank my wife Caroline, my little son Kubik and my family for their patience and support during finishing this thesis. I would also like to thank all of my colleagues who have contributed to the work which I present in this thesis and to my colleagues from the Department of Computer Science for their support and inspiration during my work.

# Contents

# 1 Introduction

Current information systems are evolving and have begun to support many parts of people's everyday routines. This evolution has been accompanied by several characteristic trends: information systems are growing in complexity and becoming more distributed and integrated, and users' demand for service availability is and the dependency on various information systems are increasing. Because the IT industry enables various business areas, competition also creates a demand to shorten the time-to-market and to decrease the development prices of information systems. These effects increase the demand for more effective and efficient software testing and verification methods. In addition to supporting sustainable development in the software industry, the demand for more effective testing methods also has a strong economic justification, given that testing and repairing defects represents a significant proportion of typical software project costs (estimates of 50% [35] and ranging from 40% to 80% [75] have been reported).

Model-based Testing (MBT) [52] is widely recognized as a systematic way of ensuring the accurate and effective testing of a software system. In MBT, a model of the system under test (SUT) is created and used to automatically generate test scenarios with a suitable strategy. Despite the challenges (such as the assumption that a sufficiently good SUT model exists), I personally consider MBT to be the best available approach to structured system testing, and the majority of my research work relates to this discipline. Over the period cited above, which is the subject of this thesis, I focused on three principal research tracks:

1. **Path-based test case generation strategies** focus on generating test cases that reflect the priorities of the SUT model. Software test scenarios are composed of sequences of actions to be exercised in the SUT. Selection of particular sequences results in a higher probability of defect detection, while other sequences may contain duplicates and cannot effectively detect defects. This knowledge has motivated us to develop and improve strategies for generating these sequences in an automated way from the SUT model. To aid in this process, various metadata can be used, especially priorities of the individual functions of the SUT (Section 2).

2. **Combinatorial and constrained interaction testing**. Exercising all possible combinations of the input data during the testing of a software system is technically impossible, because the number of combinations grows enormously for an SUT of non-trivial complexity. The same problem also occurs for various SUT configurations (e.g., the platform variants and module versions of various devices to test) in integrated software systems. This problem requires further investigation; strategies should be developed to select the relatively small set of test cases that provide a high probability of detecting the defects in the SUT (Section 3).

3. **Test automation frameworks** to increase the efficiency of testing in sub-optimally structured software development projects. Effective test automation assumes a certain level of structure in a project; for instance, the presence of SUT design documentation, which can be used to design the test cases or presence of a test automation architecture which can decrease the maintenance costs of the automated testware. However, many projects lack such a structure. Utilizing test automation and MBT concepts for such projects is not impossible, but alternative techniques should be proposed and evaluated (Section 4).

In each of these tracks, I identified areas that had not been satisfactorily examined in previous research and development work. Building on these gaps in knowledge, I proposed alternative and innovative approaches or participated in projects that aimed to do so.

The structure of this thesis is as follows. Each of three tracks outlined above is discussed in its own section. Each section begins with a *Motivation and state of the art* subsection in which I summarize the related work and identify the issues (marked as "*challenges*" in the text) that inspire the development of an innovative approach to solving the issue. Next, in the *Contribution* subsection, subsection, I introduce the projects I am leading or have participated in that address these issues. I refer to my published papers (or papers recently accepted for publication) that are included in the appendix of this thesis. For this thesis, I have selected six papers that document the projects I discuss here. Five of these papers have been published (or accepted for publication) in impact factor journals and conference proceeding and one paper is currently under review in a Q1 impact factor journal. I also present this final manuscript to provide a better overview of my current projects.

This thesis discusses six research and development challenges (all of which have numerous synergies). The variety in topics is intentional, reflecting my attempts to gain deeper insight into various important aspects of software testing automation and MBT research and development. This breadth will help me guide and supervise Ph.D. students in a wider number of areas in the software testing discipline. Another reason for this broad focus is that it will improve my ability to participate in grants and industrial cooperation projects. These are important to the funding of the Software Testing IntelLigent Lab (STILL) [1] research group, which I personally consider to be the most stimulating project in my professional track.

# 2 Path-based Test Case Generation Strategies

During the creation of the test cases for a software system, a test analyst must define the sequence of actions in the function calls of the SUT. Testers then apply these sequences in the SUT to detect defects. This technique is frequently used in many current projects and can be applied to various levels of testing and SUT abstraction (e.g., integration testing or functional end-to-end testing). To construct consistent and effective test cases, which are optimal from an economic viewpoint, a systematic approach is needed. The MBT discipline provides ways to generate these test cases from an abstracted model of SUT processes that is based on a directed graph.

## 2.1 Motivation and State of the Art

Most commonly, an SUT model is defined as a directed graph $G = (N, E)$, where $N$ is a set of nodes, $N \neq \emptyset$, $E$ is a set of edges and $E$ is a subset of $N \times N$. We define one start node $n_s \in N$. A set $N_e \subseteq N$ contains the end nodes of the graph, where $N_e \neq \emptyset$ [58]. A test case $t$ is a sequence of nodes $n_1, n_2, .., n_n$, with a sequence of edges $e_1, e_2, .., e_{n-1}$, where $e_i = (n_{i,} n_{i+1})$, $e_i \in E$, $n_i, \in N$, and $n_{i+1} \in N$. The test case $t$ begins with the start node $n_s$ ($n_1 = n_s$) and ends with the $G$ end node ($n_n \in N_e$) . The test case $t$ can be denoted as a sequence of nodes $n_1, n_2, .., n_n$, or by a sequence of edges $e_1, e_2, .., e_{n-1}$. The test set $T$ is a set of test cases. Further, a set of test requirements $R$ is defined to determine the required test coverage [58, 51]. A test requirement is a path in $G$, which must be present as a sub-path of at least one test case $t \in T$ . The test requirements can be used either for (1) definition of the general intensity of the test cases or (2) expression, in which parts of the SUT model $G$ are considered a priority that should be covered by test cases.

Various test coverage criteria are defined and recognized as a de-facto industry standard [58]. *All Edge Coverage* (or *Edge Coverage*) requires each edge $e \in E$ to be present in the test set $T$ at least once. Alternatively, *All Node Coverage* requires each node $n \in N$ to be present in $T$ at least once. To satisfy the *Edge-Pair Coverage* criterion, $T$ must contain each possible pair of adjacent edges in $G$ [58]. *All Node* and *All Edge Coverage* is suitable for low-intensity tests, whereas *Edge-Pair Coverage* provides a higher probability of defect detection (which is achieved by a greater number of test cases). For high-intensity testing, *Prime Path Coverage* is used [58]. To satisfy this criterion, each reachable prime path in $G$ must be a sub-path of the test case $t \in T$. A path $p$ from $e_1$ to $e_2$is prime if (1) $p$ is simple, and (2) $p$ is not a sub-path of any other simple path in $G$. A path $p$ is simple when no node $n \in N$ is present more than once in $p$ (i.e., $p$ does not contain any loops); the only exceptions are $e_1$ and $e_2$, which can be identical ($p$ itself can be a loop) [58].

Alternatively, the Test Depth Level ($TDL$) [82] coverage criterion can be used. $TDL = 1$ when $\forall e \in E$ the edge $e$ appears at least once in at least one test case $t \in T$. $TDL = x$ when the test set $T$ satisfies the following conditions: for each node $n \in N$, $P_n$ is a set of all possible paths in $G$ starting with an edge incoming to the decision point $n$, followed by a sequence of $x - 1$ edges outgoing from the node $n$. Then, $\forall n \in N$, the test cases in test set $T$ contain all paths from $P_n$.

[1] http://still.felk.cvut.cz

The problem of generating a $T$ that satisfies the defined test requirements has been solved by a number of algorithms [5, 54, 85, 22, 12, 51, 73]. In addition to genetic algorithms [12, 76], a number of nature-inspired algorithms have also been applied, for instance, the ant colony [22, 61] and firefly [62] algorithms or the algorithms inspired by the field of microbiology [85].

The path-based technique is universal and can be applied in various types of tests. In addition to the composition of functional end-to-end test scenarios **[A.1, A.9]**, the technique can be used to create scenarios for integration tests or testing techniques based on the SUT source code [43, 55]. On the source code level, the technique naturally overlaps with data-flow techniques [24, 84, 49, 4], that primarily aim to verify data consistency.

To assess the optimality of $T$, various criteria can be used. Commonly used test set optimality criteria are based on the number of nodes or edges in the test cases, the number and lengths of the test cases, or the coverage of $R$ by the individual test cases [54, 58, 55].

In a practical test design process, prioritizing the individual parts of the SUT model is essential to optimize the test set. As an example, consider a workflow in an information system that must be covered by path-based test scenarios. Only certain parts of the workflow represent the main part of the business process, which should run without blocking defects to ensure the business operation of the company. Other parts of the workflow have lower priority. From a practical testing viewpoint, it is important to address the prioritized parts of the workflow with intense test cases, but less stringent testing could be sufficient for the low-priority parts; this would reduce testing costs. Such prioritization adds more complexity to the problem of generating $T$ from $G$ and $R$.

As many individual reports show, previously published algorithms differ in their ability to produce a $T$ that satisfies the various test coverage criteria [51]. Moreover, they differ in their ability to produce a $T$ that will be optimal, considering the various optimality criteria. In addition, for various problem instances $G$, different algorithms will provide optimal solutions **[A.1]**. Formulation of a universal algorithm that will produce an optimal test set $T$ for various test coverage and test set optimality criteria seems to be unrealistic. This difficulty suggests the following question: could the available algorithms be combined to produce an optimal test set according to selected optimality criteria? (**Challenge 1.1**)

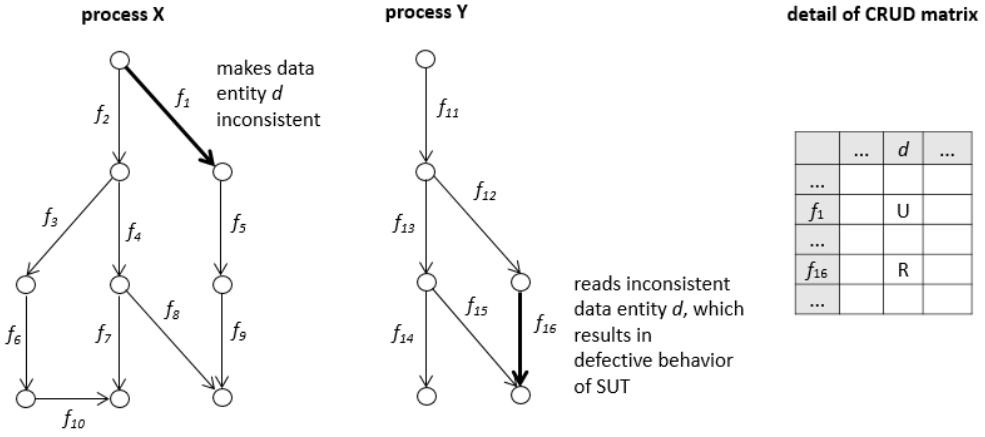
Apart from the Challenge 1.1, it must be noted that some types of defects cannot be detected using path-based techniques. One example is a data consistency defect. To detect defects of this type, we must consider not only the SUT functions but also the principal data entities processed by them. Such data entities define the data objects that are generally stored in a persistent data layer of the SUT. For test design purposes, these entities are commonly identified at the conceptual level of the SUT design, and they model the principal data objects processed by the SUT. During the SUT run, several data objects are initiated as instances of a particular data entity. These data objects are created, read, changed or deleted by SUT functions.

A function is an SUT feature that performs any of the Create, Update, Read and Delete (C, R, U, and D, respectively) operations on a data entity. $F = \{f_1, .., f_n\}$ is a set of all the SUT functions, and $D = \{d_1, .., d_p\}$ is a set of all the data entities that are considered in testing. A data entity $d \in D$ is defined with a set of attributes $A = \{a_1, .., a_n\}$. The data object $o$ is an instance of the data entity $d$. In the initial state, the values of attributes of data object $o$ are set to $V = \{v_1, .., v_n\}$. If a data entity $d$ is inconsistent, we consider this to be a consequence of an SUT defect in function $f_1$, when the following scenario is fulfilled.

Function $f_1$ modifies the attribute values of data object $o$. Based on the SUT specifications, after this change, $o$ should have its attribute values set to $V' = \{v'_1, .., v'_n\}$. Function $f_1$ contains a defect, which which causes that the attribute values of $o$ are set to $V'' = \{v''_1, .., v''_n\}$. If $V' \neq V''$, we consider the data object $o$ to be inconsistent. Such an inconsistency can trigger defective behavior in the SUT when $o$ is handled by another SUT function from $F_X \subset F$.

When $f_1$ and $F_X$ are part of the same SUT process model $G$ (representing nodes or edges of the model), such data consistency defects can certainly be detected using path-based test scenarios with All Paths Coverage strength. However, such an approach would be very impractical, as All Paths Coverage produces extensive test cases, which are resource intensive, even in the case of test automation.

Moreover, the limits of path-based techniques become clear when $f_1$and $F_X$ belong to differ-



|  | ... | *d* | ... |
|---|---|---|---|
| ... |  |  |  |
| $f_1$ |  | U |  |
| ... |  |  |  |
| $f_{16}$ |  | R |  |
| ... |  |  |  |

Figure 1: An example of a data consistency defect [A.2].

ent, disjunct SUT processes (e.g., if different user roles are used or the processes of the integrated software systems are implemented in a different subsystem but share the same data). This situation is depicted in Fig. 1.

To detect such a data consistency defect, the tester must switch from process X to process Y. The path-based technique can accomplish this task. However, the situation would have to be modeled using a directed graph based model, which would become impractical (given that it would be necessary to cover all possible transitions between process X and process Y).

These situations ($f_1$ and $F_X$ being part of the same SUT process and $f_1$ and $F_X$ belonging to disjunct SUT processes) are not rare. During our cooperation with industry, we discussed this issue several times with test analysts and test managers from more than five software industry projects. In projects that involve complex integrated information systems from the telecommunications and finance domains, such data consistency defects are estimated to be the root cause of 15-25% of the defects reported during the development lifecycle. Moreover, these defects are by nature difficult to detect and reproduce (replicating the defective behavior to analyze the defect), and give the impression of "random" behavior in the SUT. This increases the overhead in the software testing and defect removal process.

To detect data consistency defects systematically and efficiently, a specialized data consistency testing technique should be used. Regarding the SUT source code level, the problem has been addressed by numerous studies in the area of data flow analysis [24, 84, 49, 4, 23].

However, for the design of functional end-to-end tests (such as those typically required in user acceptance testing), a higher-level SUT model is needed. The Data Cycle Test (DCyT) technique [82, 31] employs the CRUD matrix, which is defined as $\boldsymbol{M} = (m_{i,j})_{n,p}$, $n =\mid F \mid$, $p =\mid D \mid$, $m_{i,j} = \{o \mid o \in \{C, R, U, D\} \Longleftrightarrow$function $f_i$ performs the respective Create, Read, Update or Delete operations on the data entity $d_j \in D\}$.

A common guidelines for the DCyT include a high-level method, how to create a CRUD matrix, a set of remarks about the possibility of static testing using a CRUD matrix as an SUT model and a method to create dynamic test cases that can detect data consistency defects. The principle of the method is to create special test cases for each of the data entities and to compose these test cases using a suitable flow of Create, Update, Read and Delete operations that are exercised on a particular data entity. Regarding the static testing, the TMap Next description of the DCyT [82] is based on verification of the completeness of the C, R, U, and D operations for each data entity $d \in D$. This approach is valid; however, it can be extended to gain more efficient testing method.

Previous literature has not intensely addressed the static testing that employs classical CRUD matrices. A number of data flow analysis studies explore the possibilities of detecting workflow design errors [78, 34, 57], including automated correction measures [3]. However, these studies used different SUT models; for instance, UML statechart diagrams [34], Petri nets [3, 57] (applicable in cases where Business Process Model and Notation (BPMN) diagrams are available as the

SUT model) or Web Services Business Process Execution Language (WS-BPEL) for verification of web services design [79].

For static testing of the database design, the Formal Concept Analysis (FCA) approach can be used [40]. From a conceptual viewpoint, this approach is similar to a static testing technique based on a CRUD matrix. However, there are several differences in the model and in the process of static testing. The formal context used to model the problem captures data objects and their attributes (which can also be used to model the events relevant to these objects). No C, R, U, or D operations are used in this model. Analysis based on the defined relationships and dependencies between the data objects is used to detect anomalies in the model; this step involves more extensive processing than CRUD matrix based static testing. The technique primarily focuses on verification of the SUT model. The FCA method also has broader applications, such as the detection of faulty data in a system [17] or construction of a system model [28].

Alternatively, the Data-Flow Matrix technique has been used for static testing of business processes [89] to detect design errors (missing, redundant or conflicting data in the process). The Data-Flow Matrix is similar to the CRUD matrix, but it includes only the Read and Write operations. The Data-Flow Matrix is then integrated with a SUT workflow model to detect design anomalies.

Despite the advantages of these approaches, a lightweight static testing technique that assumes the existence of only a CRUD matrix and does not require further analysis or combination with other SUT design information should be developed for test engineers (**Challenge 1.2**).

## 2.2 Contribution

We address **Challenge 1.1** by providing a comprehensive solution that enables determination of an optimal $T$ for various test coverage criteria, various problem instances and different test set optimality criteria. We combine three previously published algorithms currently used to solve the problem, the Brute Force Solution (BF) [54], Set-Covering Based Solution (SC) [54, 36, 37] and Matching-Based Prefix Graph Solution (PG) [54, 36, 37], as well as our work in the area (Process Cycle Test **[A.10]**, inspired by its general definition in [82], Prioritized Process Test (PPT) **[A.9]** and Set-Covering Based Solution with Test Set Reduction (RSC) **[A.1]**).

The test requirements $R$, which have been extensively used in previous work, can either be used to define the prioritized parts of the SUT model that should be examined in the test cases, or to define the general intensity of the test set $T$ (Prime Path coverage for instance [36]). Unfortunately, when the test requirements are used to specify the general test coverage criteria, they cannot also be used to determine which parts of the SUT should be considered a priority to be covered by the tests, along with the general test coverage criterion. This limits a number of published algorithms that use the test requirement concept, such as the SC or the PG [54].

To propose an alternative approach that addresses this issue, we modeled an SUT process as a weighted multigraph $\mathcal{G} = (N, E, s, t)$, where $N$ is a set of nodes, $N \neq \emptyset$, and $E$ is a set of edges. $s : E \to N$ assigns each edge to its source node and $t : E \to N$ assigns each edge to its target node. One start node $n_s \in N$ is defined in the model. A set $N_e \subseteq N$ contains the end nodes of $\mathcal{G}$ and $N_e \neq \emptyset$. For each edge $e \in E$ (resp. node $n \in N$), $priority(e)$ (resp. $priority(n)$) is defined. In addition, $priority(e) \in \{high, medium, low\}$ and $priority(n) \in \{high, medium, low\}$. When a priority is not defined, it is considered *low*. $E_h$, $E_m$ and $E_l$ are a set of high, medium and low-priority edges, respectively; $E_h \cap E_m \cap E_l = \emptyset$.

In contrast to algorithms based on $G$ and $R$, the PPT, is based on $\mathcal{G}$ as the SUT model. We used two parallel test coverage criteria: $TDL$ and its reduction by *Priority Level (PL)*, which means that it includes only the priority parts of the SUT model for the created test cases **[A.1]**. $PL \in \{high, medium\}$; $PL = high$ when $\forall e \in E_h$ the edge $e$ is present at least once in at least one test case $t \in T$. Further, $PL = medium$ when $\forall e \in E_h \cup E_m$ the edge $e$ is present at least once in at least one test case $t \in T$. When a test set $T$ satisfies the *All Edge Coverage*, it also satisfies $PL$. The $TDL$ includes *Edge Coverage* ($TDL = 1$) and *Edge-Pair Coverage* ($TDL = 2$). The model $\mathcal{G}$ and $PL$ are convertible to $G$ and a set of Test Requirements $R$ when no parallel edges are present in $\mathcal{G}$.

The strategy works as follows. The test analyst creates an SUT model $\mathcal{G}$ and defines the

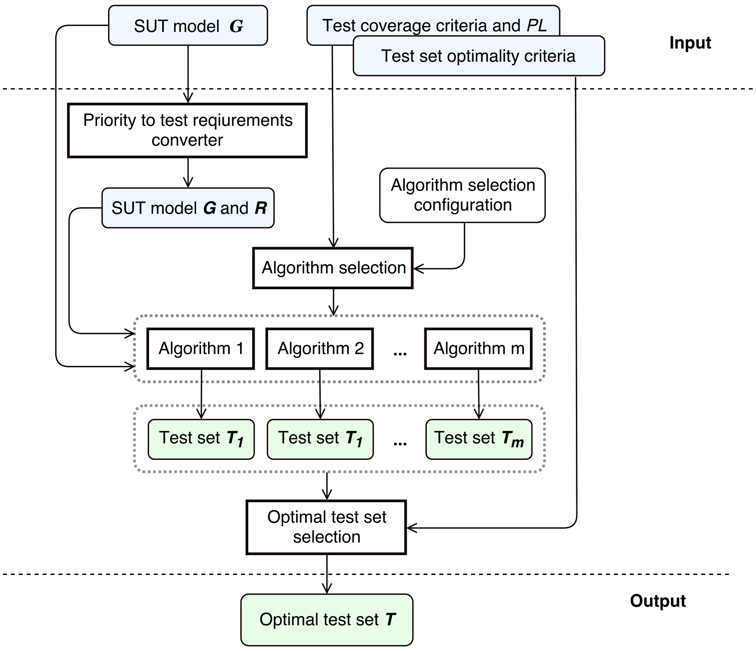


Figure 2: High-level test set selection strategy schema [A.2].

prioritized parts of the model. This process is accomplished in the Oxygen[2] platform user interface. To generate the test cases, the user selects the test coverage criteria, $PL$ and test set optimality criteria from a set of options [A.2]. The employed algorithms then create the test sets based on the test coverage criteria and $PL$. Subsequently, the test set that best fits the test set optimality criteria is provided to the test analyst. An overview of the test case selection strategy is depicted in Fig. 2.

The process begins with the conversion of $\mathcal{G}$ to $G$ and $R$ for the BF, SC, PG and RSC algorithms. Next, a set of algorithms are selected to generate the $T$ for $\mathcal{G}$ for the particular test coverage criteria. The algorithms are executed with $\mathcal{G}$ (or with $G$ and $R$, corresponding to $\mathcal{G}$), which results in test sets $T_1..T_m$. Values of the test set optimality criteria for $T_1..T_m$ are computed, and based on these criteria, the optimal $T$ of $T_1..T_m$ is selected (a more complex optimality consideration is also available; employing the optimality function or a sequence selection approach, which is composed of simpler optimality criteria).

This strategy was verified using four combinations of *Edge Coverage* and *Edge-Pair Coverage* with two $PL$s on 50 various problem instances and 16 alternative test set optimality criteria (or test set selection strategies). The results indicate the strength of the proposed approach. Depending on the individual problem instances and the test set optimality criteria, different algorithms provided the optimal test set; this effect was observed in four combinations of the *Edge Coverage* and *Edge-Pair Coverage* criteria with two $PL$s.

As already mentioned above, PPT provided the best results for *Edge Coverage*. However, in particular problem instances and given certain test set optimality criteria, SC [54, 37, 36], PG [54, 37, 36] and RSC [A.1] provided better results than PPT. Surprisingly, in isolated cases, BF [54] outperformed the other algorithms. For *Edge-Pair Coverage*, RSC outperformed the PPT for the majority of the problem instances and test set optimality criteria. However, it was not the clear winner in all instances and for all optimality criteria.

The strategy implemented in the development branch of the Oxygen (formerly PCTgen) experimental MBT platform [A.1, A.9, A.10]. This platform was developed by the STILL research group; in addition to generation of the test cases, it allows benchmarking and comparison of individual algorithms.

The proposed strategy cannot replace the development of new and more efficient algorithms for generating path-based test cases. Because it utilizes existing algorithms, the quality of the

[2] http://still.felk.cvut.cz/oxygen/

results depends on the quality of the algorithms. The added value of the strategy is that it allows the optimal test set to be selected based on defined optimality criteria for a particular SUT model. Considering that the algorithm that will generate the optimal test set for a particular problem instance is not known in advance and that comparison between the test sets can be a demanding task, the proposed strategy is a practical option for the daily test design routine in software development projects.

The results are presented in paper **[A.1]**, currently being reviewed in a Q1 impact factor journal. In this subproject, I designed the test set selection strategy, I designed the test set selection strategy and the PPT and RSC algorithms. I also designed the experimental method, supervised colleagues who implemented the strategy and the algorithm in the Oxygen platform, supervised the experiments, analyzed the results and wrote the article that presented the strategy.

To address **Challenge 1.2**, we proposed an extended method of static testing using CRUD matrices to fully utilize its potential. In general, the use of DCyT [82, 31] does not consider the preparation of the CRUD matrix. Implicitly, the CRUD matrix created by a system analyst or architect during the design phase of the software project is expected to be present as the test basis. However, alternative methods of CRUD matrix preparation are available, and techniques based on the comparison of two alternative CRUD matrices can be employed during static testing. In addition, a greater number of static testing rules can be defined for a single CRUD matrix.

In principle, the DCyT static testing using a CRUD matrix has two goals: (1) to detect design defects in the SUT, and (2) to help design more effective and consistent dynamic test cases for detecting the data consistency defects explained in section 2.1, given that inconsistencies and defects in the test basis can lead to the production of inconsistent, wrong or inefficient DCyT dynamic test cases. We define an inconsistency in the test case as occurring when the sequence of the actions to be exercised in the SUT test case cannot be executed in the actual SUT. In other words, the test case instructs the tester to execute an action that is not achievable from the actual position of the tester in the SUT.

We defined four types of CRUD matrices that can be created during a software development project: (a) a matrix constructed directly from SUT code, ideally in a semi-automated way, (b) a matrix created by a technical analyst or data architect as a part of data storage design during the design phase of the project (this matrix is typical of the CRUD matrices assumed to be present in the project by DCyT [82, 31]), (c) a matrix created by a test designer based on the business or technical specifications of the SUT and (d) a matrix created independently by a test designer based on SUT design specifications and the test designer's domain knowledge (in consultation with the other team members).

Based on these types of matrices, we proposed a set of rules that allowed comparison between two alternative CRUD matrices and the detection of design inconsistencies or defects, as well as potential indicators of a suboptimal SUT design. We also suggested these rules for a single CRUD matrix.

In an experiment measuring the efficiency of the suggested technique, we focused on the impact of static testing on the consistency of DCyT test cases, given that this topic had not been investigated in the literature. We provided several groups of participants with design documentation of an experimental SUT (Mantis BT system), in which artificial inconsistencies were present. We let one group create DCyT test cases without support from static testing, while participants in the second group were instructed to use the suggested static testing techniques and then to create the DCyT test cases. With the help of the Tapir framework (see section 4.2), we evaluated the resulting DCyT test cases. To evaluate the consistency of the test cases, we used the SUT model re-engineered using the Tapir framework. To evaluate the efficiency of the test cases with respect to their ability to detect defects, we defined a set of artificial data consistency defects in the SUT model used by the framework.

No significant differences were observed in the total number of test steps, but static testing led to the design of more consistent test cases. Additionally, these test cases had better detection abilities than the test cases prepared using an inconsistent test basis without the proposed static testing technique.

The results are presented in article **[A.2]**, which was recently accepted in the journal *Cluster Computing* (Q2, IF 2.04). In this subproject, I defined the method of extended static testing, and the experimental procedure, supervised colleagues who implemented the experimental setup using the Tapir framework, supervised the experiments, analyzed the results and wrote the article presenting the method.

# 3 Combinatorial and Constrained Interaction Testing

During software testing, various combinations of test data can be used as SUT inputs. Particular data combinations can activate a software defect, while others do not. Additionally, various data combinations can activate a particular defect. Preparing exhaustive data combinations for SUT inputs of non-trivial complexity results in an enormous number of possible combinations; this phenomenon is referred to as combinatorial explosion. Running an SUT with these data combinations is not a realistic task for either manual or automated testing.

Hence, a strategy for reducing this dataset while maintaining the data's potential to activate (detect) defects is needed. Various test coverage criteria have been defined and techniques for reducing and optimizing the dataset have been provided [58]. As part of my research, I focused on Combinatorial Interaction Testing (CIT) [15] and Constrained Interaction Testing [38] (we use the acronym ConsIT to avoid confusion with Combinatorial Interaction Testing). Both approaches are effective means of reducing the set of testing data while maintaining its defect detection potential. They also allow for flexible scaling of the test intensity. These features make the Combinatorial and Constrained Interaction Testing techniques applicable in a variety of situations in industrial software or system testing projects. In addition to the preparation of testing data combinations, these techniques can also be used to help design an efficient test bed (that is, selecting suitable platform variants or versions of the devices or software modules to test) and for solving various other engineering problems, as we explain in section 3.1.

## 3.1 Motivation and State of the Art

Most test design techniques consider SUT input parameters individually. However, based on deeper analysis of the problem and recent evidence, interactions between the input parameters can be a potentially large source of defects in an SUT [53, 68]. As an example, consider an algorithm for determining the shipment price in a logistics information system. Three inputs are used: client type, distance, and order volume. The defect is present for a condition in which the client type and the order volume are combined. Considering all three parameters separately does not guarantee that the defect will be detected. To ensure detection, the interaction between client type and order volume must be considered.

The CIT (also referred to as $t-way$ or $t-wise$ testing, where $t$ stands for the interaction strength) can provide a solution. The technique considers the interaction of $t$ inputs in the generated test set [15]. Typically, not every input parameter in the SUT will lead to detection of a defect; most faults are identified by including interactions between a small number of input parameters. The CIT minimizes the size of the test set and prevents exhaustive testing, which is not realistic from the viewpoint of project resources. The most common variant of the CIT is pairwise ($2-way$) testing, which is also the most frequently applied method in software development projects [6]. However, evidence shows that to detect some software defects, interactions between a larger number of input parameters must be considered [15]. This need represents the motivation for the $t-way$ CIT where $t>2$, which is suitable for intense testing of critical software systems. Some strategies have tried to generate the test sets for values up to $t=12$ [93].

To model the problem, a Combinatorial Covering Array (CA) can be used. CA is a practical alternative to the older mathematical object Orthogonal Array (OA). An $OA_\lambda(N;t,k,v)$ is an $N \times k$ array, where for every $N \times t$ sub-array, each $t-tuple$ occurs exactly $\lambda$ times, where $\lambda = N/v^t$; $t$ is the combination strength; $k$ is the number of input functions ($k \geq t$); and $v$ is the number of levels associated with each input parameter [29]. However, OAs reach their limit for medium and large SUTs, because it is difficult to generate suitable OAs for them. Additionally, problems arise with modeling the real testing problem using an OA. The CA addresses these limitations. A $CA_\lambda(N;t,k,v)$ is an $N \times k$ array over $(0,...,v-1)$ such that every $B=\{b_0,...,b_{t-1}\} \in$ is $\lambda$-covered and every $N \times t$ sub-array contains all ordered subsets from $v$ values of size $t$ at least $\lambda$

times, where the set of column $B = \{b_0, ..., b_{t-1}\} \supseteq \{0, ..., k-1\}$ [68]. In this case, each $t - tuple$ must appear at least once in a CA.

In cases when the number of component values varies, a Mixed Covering Array (MCA) can be used. An $MCA(N; t, k, (v_1, v_2, \ldots v_k))$, is an $N \times k$ array on $v$ values, where the rows of each $N \times t$ sub-array cover and all $t - tuples$ of values from the $t$ columns occur at least once. Alternatively, the array can be denoted as $MCA(N; t, v_1^{k_1} v_2^{k_2} .. v_k^{k})$.

Various approaches have been used to generate the CIT test cases or to solve other problems using CA. Recently, metaheuristic search techniques have been applied; these have a significant impact on the construction of optimal $t - way$ and variable strength test sets [1, 94, 91, 2, 92]. Commonly, metaheuristic strategies start with a population of random solutions. Next, one or more search operators are iteratively applied to improve the overall fitness of the solution. The main difference between individual metaheuristic strategies is based on their search operators as well as on how the exploration and exploitation are manipulated.

Numerous strategies have been applied. For instance, to support construction of the uniform and variable strength $t-way$ test set, approaches have been employed that include the simulated annealing-based strategy [11], colony optimization [87], the genetic algorithm [83], and even the harmony search algorithm, which mimics the behavior of musicians trying to compose music [1].

CAs have extensive applications [72]. For instance, they have been used for performance evaluation of communication systems [77], optimization of dynamic voltage scaling in high-performance processors [64], servomotor controllers [7], and tuning of functional order proportional–integral–derivative (PID) controllers [65].

Many applications have been identified in software testing on various levels [30, 10, 39, 70, 68, 66] and software product lines [80]. In addition to production of the test cases, CA is also suitable in the prioritization of a regression test set [14]. In these techniques, fault detection data are employed to analyze the efficiency of the test set [88].

In software testing, the applicability of a CA grows with the complexity of the SUT. In addition to generating effective input test data combinations, the technique can be used to reduce the possible SUT configurations to test (all possible combinations of particular module versions, deployment platforms or device variants). The importance of CIT has grown with the rise of Internet of Things (IoT) technology, for which the growing number of system configurations in operation has been described as an important issue [46].

Considering the applicability of the CIT technique and the number of software testing problems that can be solved using this technique, development of more effective CIT algorithms represents an important research topic. This topic includes obtaining deeper insight into the practical effectiveness of the test cases produced using the CIT technique; as previous reports on similar topics have indicated that this area merits further empirical investigation [47, 81] (**Challenge 2.1**).

To improve the effectiveness and applicability of CIT, the constraints on input parameters must be managed appropriately [67]. In the various SUTs, input parameters are subject to various constraints (e.g., if a particular shipment method is selected that is available only for certain regions or if orders exceeding a certain sum exclude certain payment methods). If these constraints are not respected during test generation, invalid test cases could be produced and could result in false defect reports. In addition, these conditions must be tested more intensely, because they represent a potential source of defects in the SUT code. The same problem is present in the combination of configurations; constraints are often present when determining the combination of platforms, versions of the individual modules, or versions of the devices to test (e.g., an Android web browser is not available for an iOS mobile phone).

Only a few of the current interaction testing strategies satisfy the constraints in the generated test set [15, 86, 69]. Optimization of the test set to satisfy the constraints also adds complexity to the problem. This demand has lead researchers to explore alternative strategies for generating the constrained interaction test sets. These efforts are supported by practical applications in the software development industry [38]. Moreover, we can expect significant growth in this research area, especially given the IoT trend. The complexity of IoT solutions, the increasing importance of integration testing, and the possible combinatorial explosion of configurations to test will all increase the demand for more effective test design techniques. Examining the constraints can help optimize the test set (or configuration set).

What are the prospective research directions in constrained integration testing? Some previous studies have discussed CIT and ConsIT, such as Nie and Lueng [15]. This study reviewed CIT and its applications, summarizing the basic concepts, methods of test set construction and applications. The study also reviewed ConsIT methods. Another study by Kuliamin and Petukhov [86] provided an overview of the various methods for constructing CIT test sets. Ahmed and Zamli [69] provided an overview of the applications of interaction testing. However, no comprehensive review paper has specifically focused on ConsIT and its research directions (**Challenge 2.2**).

## 3.2 Contribution

To obtain more insight into the effectiveness of $t - way$ combinatorial testing, adressing **Challenge 2.1**, we measured the effectiveness of $2 - way$ and $3 - way$ combinatorial test sets for a selected business scenario: reporting a problem in the issue-tracking system JTrac [3]. In the experiment, we examined entering the issue and subsequent processing of the entered data. We used the configurable data fields of the issue object to design an object consisting of nine fields; we then extended the SUT code by additional verifications of the issue data entered into the system. For each of the fields, we identified equivalence classes of the data that were entered into the SUT. We adopted the setup for this configuration from a recent industrial project on which we consulted. Several combinations of issue object values were not allowed during the issue-tracking process. We considered this version of code as a baseline SUT and the correct version. We used the baseline SUT to implement the test oracle, which determined the expected results of the individual tests.

To simulate defects in the SUT, we prepared ten instances of the SUT, each with a different set of defects, which were injected by a standard code mutation technique. We focused on mutating conditions in the source code, and different types of mutants were used to simulate the defects likely to be made by a developer (e.g., operator change, variable change or value change). To detect the defects caused by the code mutations in the SUT, we implemented Selenium WebDriver [4] automated tests, which entered the issues into the SUT and tested the subsequent processing of the issue, including the data validation mechanism for the allowed values.

We generated the test data to be entered into the SUT using the recently developed PSTG strategy [71, 67]. The $2 - way$ test set for the defined problem was composed of 48 test cases and the $3 - way$ test set was composed of 322 test cases. Next, using the Selenium test scripts and the automated test oracle, we examined each test combination in the SUT for each of the ten SUT instances (8020 tests in total). The data were automatically collected to allow further processing and evaluation.

The results provide insight into the efficiency of the $2 - way$ and $3 - way$ test sets as well as the types of defects that could be detected by the individual tests. The potential to detect a defect caused by a code mutation was not uniformly distributed among the test cases. Each test case had different detection capabilities. Three artificial defects were not detected by the $2 - way$ test set, but all the defects were detected by the $3 - way$ test set. Surprisingly, the ratio of tests that did not detect any defect in the SUT was relatively high; it was similar for the $2 - way$ (39,6%) and $3 - way$ (40,4%) test sets. The findings can help in the development of prioritization method for the regression test set based on the efficiency of the test cases examined in the SUT.

The results are presented in paper **[A.3]**, which was published in the 2017 IEEE International Workshop on Combinatorial Testing and its applications (part of the IEEE QRS 2017 conference proceedings). For this subproject, I designed the experiment, supervised colleagues who implemented the experiments, supervised the experiment, analyzed the results, wrote part of the paper presenting the experiment and reviewed the final text.

To address **Challenge 2.2**, I participated in a comprehensive systematic literature review to summarize and categorize work related to Constrained Interaction Testing over the last decade.

[3] JTrac Official Web Page, http://jtrac.info/

[4] http://www.seleniumhq.org/projects/webdriver/

Although previous reviews have examined CIT [15, 69, 86], no aggregation of the available research in Constrained Interaction Testing had been performed. Thus, the paper is a useful complement to existing reviews, because it presents a more recent and complete review of the field. The goal of this study was to identify relevant papers and evaluate the methods used for ConsIT and the results that had been presented, thus allowing discussion of future research directions and opportunities. We used an established systematic method for conducting mapping studies in the software engineering area [45].

The study aimed to address eight research questions, as follows: (1) the evolution of published studies in the field, (2) an overview of researchers, organizations, and countries active in the field, (3) the topics and subjects that had been addressed in the research and their distribution, (4) the strategies and techniques used to support the generation of constrained interaction test sets, (5) the benchmarks used to evaluate Constrained Interaction Testing techniques, (6) the current applications of Constrained Interaction Testing, (7) the challenges and limitations of the field, and (8) possible future research directions.

By using a three-phase search and refinement process, we narrowed the original 2668 papers found by the search string in five databases (IEEE Explore, ScienceDirect, ACM Digital Library, SpringerLink, and Scopus) to a final set of 103 relevant high-quality studies, which were analyzed further. The studies were grouped into four categories: constrained test generation studies, application studies, generation and application studies and model validation studies.

A number of findings from the study can be used to discuss prospective future research areas. To solve the constraints, it was most common to exclude constraints or to use a constraint solver. The most common areas of ConsIT applications were software product lines, fault detection and characterization, test selection, and security and GUI testing.

Further details and data can be found in article **[A.4]**, which was published in the journal *IEEE Access* (Q1, IF 3.24). In this subproject, I provided feedback and comments regarding the method of the survey, participated in data collection and processing and reviewed the text of the article during its creation and finalization.

# 4 Test Automation for Sub-optimally Structured Projects

Current test automation frameworks and test automation approaches are relatively efficient when the following conditions are satisfied: (1) tests are well defined and (2) the SUT structure does not change. However, these conditions are not met in many software development projects. There are various reasons for this, such as frequent scope changes, obsolete, inconsistent, or even missing design documentation resulting from a lack of resources or an inadequately structured project management approach. Based on many reports on this topic (summarized in [87] and from our own industry observations, this occurs relatively frequently.

The test automation concept can still be utilized effectively in these projects. However, doing so requires the development of alternative strategies.

## 4.1 Motivation and State of the Art

Automated tests that simulate a user's actions in the front-end (FE) user interface of the SUT (we use the term FE automated tests) generally overperform manual testing in several respects. The tests can be repeatedly executed on various platform combinations with minimal cost, guaranteeing a defined sequence of test steps relative to manual tests; the tests can also be executed in non-productive time slots in the software development cycle. However, automated testing has its tradeoffs. Implementation of automated testing usually costs more than defining manual test scenarios. Exactly defined sequences of test steps tend to be brittle when the SUT FE user interface changes. Additionally, automated tests, unlike human testers, cannot overcome inconsistencies between their definition and the actual state of the SUT FE. This frequently leads to interruption of the test flow and can result in a false defect report. Consequently, the maintenance costs associated with sub-optimally structured automated tests grow rapidly in the case of frequent SUT changes. High maintenance costs are the major challenge associated with this technology, based on both our own experience **[A.8]** and other reports in the literature [50, 32].

To create FE automated tests, two major strategies are currently used: (1) Record and Replay, in which the test automation tool saves a sequence of test steps performed in the SUT and replays them later as a test, and (2) Descriptive Programming, in which the automated test is programmed by a test developer using a test automation application programming interface (API).

These two strategies can be combined. Record and Replay is generally valuable for rapid test creation. However, it often results in high maintenance costs for the created tests [21]. Descriptive Programming can reduce brittleness and thus the maintenance costs of the tests by techniques such as employing reusable objects, adding more intelligence to SUT FE element locators and using an extra configurable layer for element identification. However, the initial effort required to create the test scripts is greater.

Employing a reusable object in the test automation code has been a basic technique for almost two decades [18], because code-fragment duplication is potentially problematic and can lead to increased maintenance overhead and script brittleness. A number of frameworks addressing the reusability issue have been presented [27, 13, 26]. Additionally, the Object Character Recognition (OCR) approach has been proposed to reduce maintenance costs [20].

Although each of these platforms provides an individual level of support for reusable objects in the test scripts, none have implemented a direct solution to reduce duplication in previously recorded test scripts. Few studies have explored this area. In the BlackHorse project [74], tests are recorded and their brittleness is subsequently reduced during post-processing, which analyzes potentially reusable parts.

Regarding support for developers of the automated tests, the current Integrated Development Environments (IDEs) are limited in their ability to detect code fragments, meaning the same action in the test. The programming languages currently used for test automation allow a number of variants for implementing identical test steps. An example in Java and the Selenium WebDriver is provided below; the same test step, clicking on an element with ID “login”, is represented by two alternative notations:

```
driver.findElement(By.id("login")).click();
Element e = driver.findElement(By.id("login")); e.click();
```

The current possibilities of code static analysis being employed in IDEs have reached their limit at this point. An analysis specifically designed for automated tests should be provided (**Challenge 3.1**). Such an analysis would also enable using the Record and Replay approach and combining it with descriptive programming more efficiently.

To define a scenario for FE-based automated tests, manual test scenarios or design documentation are usually used. However, in numerous software projects, this documentation is missing, is incomplete or obsolete. These projects still require an efficient testing method. The Exploratory Testing (ET) method is appropriate in these cases.

The efficiency of ET strongly depends on the methods used, the documentation of the tests performed and the parts of the SUT that have been explored, and the distribution of tasks among individual testers of the team. The key factor is systematic documentation of the explored path and examined test cases [59, 19]. Such documentation prevents the duplication of performed tests, leads to the exploration of larger parts of the SUT, improves the accuracy of reporting and gives better insights into the test coverage and the actual state of the SUT. However, poor structuring of the ET process can lead to issues with test prioritization, selection of a suitable test scope and repetition of tests [19, 8]. The particular SUT exploration strategy is also an important factor [48], as is the positive role of teamwork in the overall efficiency of the ET process [59]. However, these factors have only been assessed for a manual version of the ET process.

To conduct an SUT exploration strategy efficiently, tests must be manually recorded and documented, which can generate additional overhead for the testing team. This overhead can outweigh the benefits gained by a more efficient SUT exploration strategy. Thus, automating the tasks related to test documentation would be helpful (**Challenge 3.2**).

Practically, constructing such a machine support means combining three fields: (1) engineering of the SUT model, (2) generation of tests from the SUT model and (3) the ET technique. The first two areas are closely related to the MBT approach. However, a significant difference is related to the fact that machine support of the ET requires reengineering the SUT model via a continuous process that runs in parallel with the exploration of the SUT by the testers.

Several studies have explored reengineering the SUT model based on its actual state. From a web applications viewpoint, the Guitar [56] project serves as an example. The Guitar crawls a web application and creates a state machine model based on possible transitions in the SUT user interface. An alternative for mobile applications, the MobiGuitar, has also been developed [16]. Additionally, rather than the state machine diagram, a Page Flow Graph can be used [41].

Numerous methods can be used to generate the tests from the SUT model. UML, as an extensively used modeling language, is also suitable for generation of the test cases [51]. Typically, behavioral specifications diagrams (e.g., sequence, collaboration, and statechart and activity diagrams [44]) are employed. Alternative modeling languages such as SysML [25] or IFML can be used. Finally, mainstream programming languages, finite machine notations, and mathematical formalisms such as coq [90] have been utilized.

Test cases that can efficiently guide the tester during the ET process will differ from the standard test cases produced by an MBT technique in several ways: (1) the Exploratory Tester should have a certain level of freedom to use their experience to explore the SUT efficiently (even if this is recorded and guided), (2) previous exploration history and entered test data (including from other team members) can help the tester decide on next steps, and (3) to make a test case more effective, particular elements of the user interface must be shown explicitly to the tester.

Previous literature has provided only limited support for the ET using an automated framework. Initial experiments that aid the ET using the MBT approach have been performed by Schaefer [42]. In this approach, an initial SUT model is assumed to be present. Based on this model, automated test cases are derived to exercise the SUT, but testers are given flexibility to divert from the defined tests cases to exercise more variants or transitions.

## 4.2 Contribution

In the TestOptimizer project addressing **Challenge 3.1**, we provide a framework that enables utilization of the advantages of both recording and descriptive programming for the creation of FE automated tests. The relatively inexpensive creation of automated tests by recording can be combined with refactoring the repetitive parts of these scripts to decrease their brittleness and maintenance costs. We assist this refactoring process with an automated method based on post-processing the recorded test scripts and identifying the potentially reusable parts. This method can also be used in refactoring descriptively programmed test scripts in which few or no reusable objects are identified **[A.5]**.

The TestOptimizer loads a set of automated test scripts and identifies repeating code fragments. Next, it suggests potentially reusable subroutines to the test script developer, including their positions in the source code. The final decision concerning whether a suggested subroutine is a suitable candidate for a reusable object is the responsibility of the developer. In this analysis, we do not solely consider identical fragments of the source code. Such an approach would only allow the detection of trivial cases of redundancy. The TestOptimizer analyzes the automated test source code to find fragments that have the same semantics regarding the actions that the automated tests perform in the SUT user interface. The source code is converted into an abstract layer that reflects the semantics of the test steps it contains. A test step signature concept is used to build this abstraction of the automated test. The analyzed code can be written in a common programming language with a test automation API (e.g., Java and Selenium WebDriver), or the code can be an internal representation of test cases in a particular test automation tool (e.g., Selenese).

In the current project phase, the TestOptimizer does not automatically refactor the automated tests; the test developer maintains control of the source code throughout the process. This feature, together with the flexibility of the proposed approach, differentiate the TestOptimizer from BlackHorse [74], the previous project in this area that addresses a similar problem. The BlackHorse framework employs a proprietary recording tool, which captures the test traces.

These test traces are converted to Java test script code, which ensures greater script stability and minimizes the effects of change in the SUT FE.

In contrast, the TestOptimizer analyzes already created scripts and identifies common subroutines, which are suggested to the test developer. The framework is more platform independent. By developing a particular test script converter module, the TestOptimizer can be used to analyze automated test scripts in other languages or test automation APIs. The proposed approach does not imply any dependency of the created test automation code on an external library or framework.

The experimental results show that applying the TestOptimizer framework can decrease test creation and optimization costs. Recording the tests decreases test creation costs, while automated identification of code refactoring opportunities decreases the subsequent test set optimization costs. Moreover, optimization of the tests results in lower maintenance costs of the test set. The greatest potential of the framework lies in reducing the test maintenance costs for larger automated test sets that are recorded or sub-optimally structured.

A novelty of the proposed approach involves its post-processing method, which is based on an abstraction of the analyzed tests that enables test engineers to identify truly relevant reusable subroutines with greater accuracy than the common tools used for code refactoring. At this stage, our semantic analysis is still approximate and can detect fewer refactoring opportunities than a human. Nevertheless, the approach is already more efficient than simply comparing source code fragments, as shown by experiments in which the PMD tool was used as a baseline.

The results are presented in paper **[A.5]**, which was published in the *International Journal of Software Engineering and Knowledge Engineering* (Q4, IF 0.3). This work formed part of the Ph.D. project of Martin Filipsky, for whom I served as the supervisor-specialist. I led Martin in formulating the topic of his Ph.D. project, participated in designing the TestOptimizer framework and the experiments, participated in the analysis and evaluation of the data and wrote part of the paper presenting the proposed concept and experimental results.

In the Test Analysis SUT Process Information Reengineering (Tapir) project, which addresses **Challenge 3.2**, we created a framework for automated support of the ET process. The framework aims to improve the efficiency of ET by automating activities related to (1) recording tests previously exercised in the SUT, (2) decision support regarding parts of the SUT that will be explored in later tests, and (3) organization of work for a team of exploratory testers **[A.6]**.

Using a special web browser extension, the framework tracks testers' activity in the browser and incrementally builds the SUT model, which is based on the SUT user interface. The model can also be extended by various metadata entered by the testers, such as a prioritization of SUT parts or equivalence classes for particular data inputs. Based on the actual position of the tester in the SUT and the SUT model, the Tapir framework generates navigational test cases and provides them to the testers. The parts that should be explored in the next step are also highlighted in the SUT user interface in the web browser. The navigational test cases help the testers explore the SUT more systematically and efficiently, reducing possible duplication of tests or test data. This method is particularly effective in cases with a more extensive team of testers.

The framework and SUT model are designed for web-based SUTs. Regarding teamwork organization, the framework supports two principal roles: tester and test lead.

The tester is guided by the framework and explores the parts of the SUT that have not been previously tested. For each tester, a navigational and test data strategy can be set by the test lead. The navigational strategy determines the sequence of SUT functions to be explored during the tests, which is suggested to the tester by the navigational test cases. The test data strategy determines the test data to be entered in the SUT inputs (e.g., forms). The test data are also suggested to the testers by the navigational test cases.

The test lead prioritizes the elements of the SUT front end, which are captured by the SUT model. Priority can be added to pages, links and action elements of the SUT pages. A priority can be set during the initial exploration of the SUT and changed later during the ET process. The priorities are reflected by navigational strategies. The test lead also defines equivalence classes for the individual SUT input elements. When the equivalence classes are defined for a particular input form on an SUT page, the test lead can let the Tapir framework generate suitable test data combinations to be explored and remember these in the SUT model. In this process, a CIT module is used.

The Tapir framework's navigational strategies are primarily focused on exploring new SUT parts that have not been tested previously. Additionally, a navigational strategy focused on regression testing and retesting defect fixes is available. In these strategies, the framework considers the defined priorities, previous visits to the page or element, the complexity of the following pages and other factors. Team variants of these strategies are also available, and they help organize and automate the work of the individual testers.

The test data strategies available in the framework cover both exploration of new SUT parts and support of regression testing and retesting of defect fixes. To suggest a suitable test data combination to the tester, the test data strategies use the following: testing data previously used in the SUT inputs, defined equivalence classes and test data combinations computed by the CIT module.

To evaluate the efficiency of the Tapir framework, we conducted five different case studies. During these studies, the framework was connected to the Mantis BT, Moodle, OFBiz and JTrac open-source web systems and Pluto, a healthcare information system developed by a recent software project. For Mantis BT, we injected artificial defects accompanied by a logging mechanism. For the Pluto system, real software defects were present in the code, and we knew the location of these defects from the project history. Using Mantis BT and Pluto, we conducted a set of experiments with a group of exploratory testers. In these experiments, we compared the efficiency of manual ET with the ET supported by the Tapir framework. The ET supported by the Tapir framework enabled the testers to explore more SUT functions and to explore components of the SUT that were previously unreached. This effect was confirmed by the number of SUT pages explored per time unit. These results were relevant for rapid lightweight ET, in which a tester's goal is to efficiently explore new SUT functions and as many previously unexplored functions as possible. With the support of the Tapir framework, testers performing exploratory tests were able to reach and activate more of the unique inserted artificial defects; additionally, the time needed to activate one unique defect decreased. All the improvements were observed for both Mantis BT and the Pluto system.

The initial results achieved using the first version of the Tapir framework prototype are presented in article **[A.7]**, published in the journal *Cluster Computing* (Q2, IF 2.04). In this thesis, I present an article describing the more recent version of the framework, including actual experiments **[A.6]**, which has been accepted to publication in journal *IEEE Transactions on Reliability* (Q1, IF 2.79) in January 2018.

This work has been conducted as part of the Ph.D. project of Karel Frajtak, for whom I served as the supervisor-specialist. I led Karel in formulating the topic of the Ph.D. project, participated in the design of the Tapir framework and the experiments and participated in the analysis and evaluation of the data. Further, I supervised the writing of the initial paper **[A.7]** and wrote the part of the paper **[A.6]** that presents the proposed concept and experimental results. Karel's Ph.D. thesis is now awaiting defense and received positive reviews from all three appointed reviewers.

# 5 Summary and the Future Work

In this thesis, I present my research work in MBT and software test automation for the period from 2013-2017. I focus on three principal tracks, for which I defined six challenges arising from recent state in the field. I demonstrate how I have addressed these challenges by proposing an algorithm, strategy or framework that aims to solve them. In the text, I also describe my personal contribution to each of these proposals.

In the area of **path-based test case generation strategies** (Section 2), I proposed an alternative definition of the SUT model, providing greater flexibility in modeling and formulating new algorithms that reflect the priorities of individual model components. Based on this model, I formulated the PPT algorithm, which allows the combination of the $TDL$ coverage criterion with the $PL$ criterion, determining which priorities in the SUT model will be covered by the generated test cases. For *Edge Coverage* and some problem instances for the *Edge-Pair Coverage* criterion, the PPT outperforms several previously published algorithms with respect to test set optimality when prioritization of SUT model parts is considered. Next, I proposed a test set selection strategy that uses a set of algorithms to generate the test cases for a particular problem instance and then selects the optimal test set based on the defined optimality criteria. This strategy proved to be a valid approach; as with the experimental data for *Edge* and *Edge-Pair Coverage*, different algorithms provided optimal results for different problem instances.

In the area of **Combinatorial and Constrained Interaction Testing** (Section 3), I lead empirical experiments investigating the efficiency of the produced test cases in a real software system. These data provide a valuable source of information that is applicable in the development of new algorithms. Therefore, I conducted an experiment analyzing the effectiveness of $2-way$ versus $3-way$ combinatorial test sets in a defined business scenario for a set of ten SUT instances with different injected defects, involving 8020 test runs in total. The recent trend to optimize the combinatorial test sets requires examining the interaction between individual input parameters of the SUT; currently, we have two experimental projects addressing this topic in the STILL group. Although some initial reports have been published, there has not been a comprehensive review paper focusing specifically on Constrained Interaction Testing and its research directions; therefore, I participated in writing this review paper.

Regarding the area of **Test automation frameworks** that can assist in the test automation of sub-optimally structured software development projects (Section 4), I supervised two Ph.D. projects (in the role of supervisor-specialist) that aimed to create such frameworks. The TestOptimizer aims to help optimize user interface-based automated test scripts, leading to decreased maintenance of these tests, which is a major issue for this technology. The framework searches for code refactoring opportunities (common subroutines) in a set of test scripts. In contrast to classical static code analyzers, TestOptimizer also detects subroutines that differ in syntax but perform the same actions when the test is executed. Compared to manual identification of potential common subroutines, the automated analysis was less accurate, but significantly faster. In the Tapir project, combining MBT, Model Reengineering, and Exploratory Testing, we created a framework that assists ET with the automated generation of navigational test cases by guiding a team of exploratory testers through the SUT. A specific model of the SUT is created in real time during the testing process, and the navigational test cases are generated dynamically based on the actual state of testing and the selected navigational and test data strategy. Compared to a manual ET, the framework is more effective in several aspects, such as the extent of explored functions or the average efficiency.

Numerous synergies are present between the research tracks discussed above. Test data combinations identified by the Combinatorial and Constrained Interaction Testing techniques can be entered as SUT inputs for decision points in the test cases generated by the path-based test case generation strategies. The generated path-based test scenarios and the test data combinations can be entered into a test automation architecture as parametrized tests, minimizing the need to define the automated tests manually (for instance, the CIT module is employed in the Tapir framework to determine the test data combinations to be entered into the SUT). I am currently participating in another experimental project in which Selenium tests are automatically composed from a library of reusable test objects based on path-based test cases generated by the Oxygen platform. The SUT configuration variants to test can be reduced using the ConsIT technique, considering the particular constraints of test bed setup and test automation framework limitations during generation of the platform combinations. The empirical results regarding the efficiency of particular test cases gathered using a reporting mechanism in a test automation framework can be used to further optimize a path-based or combinatorial regression test. Thus, these investigations overlap in numerous ways. Exploring their synergies can lead to more effective software testing methods.

## 5.1 Future Research Directions

For this reason, I consider all three tracks discussed in this thesis to be worthy of further investigation, especially in the context of testing IoT solutions, an area that has grown significantly in importance in recent years. A number of challenges must be addressed by researchers in this domain. IoT solutions are characterized by increasing demands on the testing of security, integration, a growing number of configurations or the IoT solutions under a limited or unstable network connection [9, 33, 63].

However, based on a literature survey we recently conducted (part of a systematic mapping study on IoT quality assurance methods that involved analysis of over 500 papers), the current literature coverage of these issues is not sufficient. In addition to the security and privacy issues and security testing [60] that have been the focus of numerous recent studies, many other quality assurance related areas lack research on methods appropriate to the growing importance of the IoT technology.

With colleagues, I reacted to this demand by preparing the Quality Assurance System project for the Internet of Things Technology (funded by the Technology Agency of the Czech Republic government in 2017), and by focusing on the development of IoT-specific testing techniques. In the area of paths-based testing, I am currently attempting to develop a technique to generate test cases that specifically focus on testing of an IoT solution under a limited or unstable connection. The Combinatorial and Constrained Interaction testing strategies we focus on in our research group are also applicable to solving the problem of testing a large number of configurations. In the area of test automation frameworks, together with an industrial partner, we are developing an innovative IoT test automation framework that combines unit integration and end-to-end integration testing with simulation of part of the tested infrastructure. Despite the technological and conceptual challenges and risks that such a hybrid approach involves, we strongly believe that integrating these tests will enable more efficient testing relative to maintaining the individual tests in separate testbeds or test set-ups, as has been reported by a number of industrial partners interviewed during the analytical phase of the project.

Considering not only the IoT technology but also other areas of information technology development, work processes in numerous domains that depend on ICT systems must be backed up by proper quality assurance techniques and methods. The principal concepts and techniques used in software testing are valid for three decades. However, to function effectively, particular techniques should be tailored to the contemporary state of the technology and software development styles. The present demand for more effective and automated software quality assurance techniques renders the MBT and related disciplines worthy of further research and development.

# 6 List of selected author's publications

[A.1] Miroslav Bures, and Bestoun S. Ahmed. Employment of Multiple Algorithms for Optimal Path-based Test Selection Strategy. Currently under review in Q1 impact factor journal. Manuscript published at arXiv.org, 1802.08005, https://arxiv.org/abs/1802.08005, 22 Feb 2018.

[A.2] Miroslav Bures, Tomas Cerny, Karel Frajtak, and Bestoun S. Ahmed. Testing the consistency of business data objects using extended static testing of CRUD matrices. *Cluster Computing*, online, pages 1-14. 2017. (Q2, IF 2.04)

[A.3] Bures, Miroslav, and Bestoun S. Ahmed. On the effectiveness of combinatorial interaction testing: A case study. In *2017 IEEE International Conference on Software Quality, Reliability and Security* (IEEE International Workshop on Combinatorial Testing and its Applications, Proceedings companion volume), pages 69-76. IEEE, 2017.

[A.4] Bestoun S. Ahmed, Kamal Z. Zamli, Wasif Afzal, and Miroslav Bures. Constrained Interaction Testing: A Systematic Literature Study. *IEEE Access*, 5, pages 25706-25730. 2017. (Q1, IF 3.24)

[A.5] Miroslav Bures, Martin Filipsky, and Ivan Jelinek. Identification of Potential Reusable Subroutines in Recorded Automated Test Scripts. *International Journal of Software Engineering and Knowledge Engineering*, 28(01), pages 3-36. 2018. (Q4, IF 0.3)

[A.6] Miroslav Bures, Karel Frajtak, and Bestoun S. Ahmed. Tapir: Automation Support of Exploratory Testing Using Model Reconstruction of the System Under Test. Accepted in *IEEE Transactions on Reliability*, January 2018. (Q1, IF 2.79). Pre-print published at arXiv.org, 1802.07983, https://arxiv.org/abs/1802.07983, 22 Feb 2018.

[A.7] Karel Frajtak, Miroslav Bures, and Ivan Jelinek. Exploratory testing supported by automated reengineering of model of the system under test. *Cluster Computing*, 20(1), pages 855-865, 2017. (Q2, IF 2.04)

[A.8] Miroslav Bures. Automated testing. In Miroslav Bures, Miroslav Renda, and Michal Dolezel, *Effective software testing*, pages 179-203. Grada, Prague, 2016. (book in Czech)

[A.9] Miroslav Bures, Tomas Cerny, and Matej Klima. Prioritized process test: More efficiency in testing of business processes and workflows. In *International Conference on Information Science and Applications*, Lecture Notes in Electrical Engineering vol. 424, pages 585-593. Springer, 2017.

[A.10] Miroslav Bures. PCTgen: automated generation of test cases for application workflows. In *New Contributions in Information Systems and Technologies*, Advances in Intelligent Systems and Computing, vol. 353, pages 789-794. Springer, 2015.

**Papers [A.1] to [A.6] are included as appendices of this thesis.**

# References

[1] Alsewari A. R. A. and Zamli K. Z. Design and implementation of a harmony-search-based variable-strength t-way testing strategy with constraints support. *Information and Software Technology*, 54(6):553–568, 2012.

[2] Alsewari A. R. A. and Zamli K. Z. A harmony search based pairwise sampling strategy for combinatorial testing. *International Journal of Physical Sciences*, 7(7):1062–1072, 2012.

[3] Awad A., Decker G., and Lohmann N. Diagnosing and repairing data anomalies in process models. In *International Conference on Business Process Management*, pages 5–16. Springer, 2009.

[4] Chandra A. and Singhal A. Study of unit and data flow testing in object-oriented and aspect-oriented programming. In *Innovation and Challenges in Cyber Security (ICICCS-INBUSH), 2016 International Conference on*, pages 245–250. IEEE, 2016.

[5] Dwarakanath A. and Jankiti A. Minimum number of test paths for prime path and other structural coverage criteria. In *IFIP International Conference on Testing Software and Systems*, pages 63–79. Springer, 2014.

[6] Hervieu A., Marijan D., Gotlieb A., and Baudry B. Practical minimization of pairwise-covering test configurations using constraint programming. *Information and Software Technology*, 71:129–146, 2016.

[7] Sahib M. A., Ahmed B. S., and Potrus M. Y. Application of combinatorial interaction design for dc servomotor pid controller tuning. *Journal of Control Science and Engineering*, 2014:7, 2014.

[8] Shah S. M. A., Gencel C., Alvi U. S., and Petersen K. Towards a hybrid testing process unifying exploratory testing and scripted testing. *Journal of Software: Evolution and Process*, 26(2):220–250, 2014.

[9] Whitmore A., Agarwal A., and Da Xu L. The internet of thingsâa survey of topics and trends. *Information Systems Frontiers*, 17(2):261–274, 2015.

[10] Cohen M. B., Dwyer M. B., and Shi J. Constructing interaction test suites for highly-configurable systems in the presence of constraints: a greedy approach. *IEEE Transactions on Software Engineering*, 34(5):633–650, 2008.

[11] Cohen M. B., Gibbons P. B., Mugridge W. B., Colbourn C. J., and Collofello J. S. A variable strength interaction testing of components. In *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International*, pages 413–418. IEEE, 2003.

[12] Hoseini B. and Jalili S. Automatic test path generation from sequence diagram using genetic algorithm. In *Telecommunications (IST), 2014 7th International Symposium on*, pages 106–111. IEEE, 2014.

[13] Mu B., Zhan M., and Hu L. Design and implementation of gui automated testing framework based on xml. In *Software Engineering, 2009. WCSE'09. WRI World Congress on*, volume 4, pages 194–199. IEEE, 2009.

[14] Bryce R. C. and Colbourn C. J. Test prioritization for pairwise interaction coverage. In *Proceedings of the 1st International Workshop on Advances in Model-based Testing*, A-MOST '05, pages 1–7, New York, NY, USA, 2005. ACM.

[15] Nie C. and Leung H. A survey of combinatorial testing. *ACM Computing Surveys*, 43(2):1–29, 2011.

[16] Amalfitano D., Fasolino A. R., Tramontana P., Ta B. D., and Memon A. M. Mobiguitar: Automated model-based testing of mobile apps. *Software, IEEE*, 32(5):53–59, 2015.

[17] Borchmann D. Exploring faulty data. In *International Conference on Formal Concept Analysis*, pages 219–235. Springer, 2015.

[18] Lonngren D. D. Reducing the cost of test through reuse. In *AUTOTESTCON'98. IEEE Systems Readiness Technology Conference., 1998 IEEE*, pages 48–53. IEEE, 1998.

[19] Pfahl D., Yin H., Mäntylä M. V., and Münch J. How is exploratory testing used? a state-of-the-practice survey. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 5. ACM, 2014.

[20] Alegroth E., Nass M., and Olsson H. H. Jautomate: A tool for system-and acceptance-test automation. In *Software testing, verification and validation (icst), 2013 ieee sixth international conference on*, pages 439–446. IEEE, 2013.

[21] Alégroth E. and Feldt R. Industrial application of visual gui testing: Lessons learned. In *Continuous Software Engineering*, pages 127–140. Springer, 2014.

[22] Sayyari F. and Emadi S. Automated generation of software testing path based on ant colony. In *Technology, Communication and Knowledge (ICTCK), 2015 International Congress on*, pages 435–440. IEEE, 2015.

[23] Wedyan F., Ghosh S., and Vijayasarathy L. R. An approach and tool for measurement of state variable based data-flow test coverage for aspect-oriented programs. *Information and Software Technology*, 59:233–254, 2015.

[24] Denaro G., Margara A., Pezze M., and Vivanti M. Dynamic data flow testing of object oriented systems. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 947–958. IEEE Press, 2015.

[25] Chang C. H., Lu C. W., Yang W. P., Chu W. C. C., Yang C. T., Tsai C. T., and Hsiung P. A. A sysml based requirement modeling automatic transformation approach. In *2014 IEEE 38th International Computer Software and Applications Conference Workshops*, pages 474–479, July 2014.

[26] Kaur H. and Gupta G. Comparative study of automated testing tools: Selenium, quick test professional and testcomplete. *International Journal of Engineering Research and Applications*, 3(5):1739–43, 2013.

[27] Nguyen D. H., Strooper P., and Süß J. G. Automated functionality testing through guis. In *Proceedings of the Thirty-Third Australasian Conferenc on Computer Science-Volume 102*, pages 153–162. Australian Computer Society, Inc., 2010.

[28] Carbonnel J., Huchard M., Miralles A., and Nebut C. Feature model composition assisted by formal concept analysis. In *ENASE: Evaluation of Novel Approaches to Software Engineering*, pages 27–37. SciTePress, 2017.

[29] Colbourn C. J., Kéri G., Soriano P. P. R., and Schlage-Puchta J.-C. Covering and radius-covering arrays: Constructions and classification. *Discrete Applied Mathematics*, 158(11):1158–1180, 2010.

[30] Colbourn C. J. and McClary D. W. Locating and detecting arrays for interaction faults. *Journal of Combinatorial Optimization*, 15(1):17–48, 2008.

[31] De Grood D. J. *Testgoal: Result-driven testing*. Springer Science & Business Media, 2008.

[32] Kasurinen J., Taipale O., and Smolander K. Software test automation in practice: empirical observations. *Advances in Software Engineering*, 2010, 2010.

[33] Kiruthika J. and Khaddaj S. Software quality issues and challenges of internet of things. In *2015 14th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES)*, pages 176–179, Aug 2015.

[34] Küster J., Ryndina K., and Gall H. Generation of business process models for object life cycle compliance. *Business Process Management*, pages 165–181, 2007.

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

[36] Offutt J. and Ammann P. Graph coverage web application, http://cs.gmu.edu:8080/offutt/coverage/graphcoverage, 2017.

[37] Offutt J., Ammann P., and Li N. Source code of coverage web applications, http://cs.gmu.edu/ offutt/softwaretest/coverage-source/, 2015.

[38] Petke J., Cohen M. B., Harman M., and Yoo S. Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection. *IEEE Transactions on Software Engineering*, 41(9):901–924, 2015.

[39] Petke J., Yoo S., Cohen M. B., and Harman M. Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 26–36, New York, NY, USA, 2013. ACM.

[40] Poelmans J., Dedene G., Snoeck M., and Viaene S. Using formal concept analysis for the verification of process-data matrices in conceptual domain models. In *Proceedings of the IASTED International Conference on Software Engineering*, pages 79–86. Acta Press, 2010.

[41] Polpong J. and Kansomkeat S. Syntax-based test case generation for web application. In *2015 International Conference on Computer, Communications, and Control Technology (I4CT)*, pages 389–393, April 2015.

[42] Schaefer C. J. and Do H. Model-based exploratory testing: A controlled experiment. In *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on*, pages 284–293. IEEE, 2014.

[43] Yan J. and Zhang J. An efficient method to generate feasible paths for basis path testing. *Information Processing Letters*, 107(3-4):87–92, 2008.

[44] Jena A. K., Swain S. K., and Mohapatra D. P. A novel approach for test case generation from uml activity diagram. In *2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*, pages 621–629, Feb 2014.

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

[46] Rose K., Eldridge S., and Chapin L. The internet of things: An overview. *The Internet Society (ISOC)*, pages 1–50, 2015.

[47] Inozemtseva L. and Holmes R. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, pages 435–445. ACM, 2014.

[48] Micallef M., Porter C., and Borg A. Do exploratory testers need formal training? an investigation using hci techniques. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 305–314, April 2016.

[49] Prabu M., Narasimhan D., and Raghuram S. An effective tool for optimizing the number of test paths in data flow testing for anomaly detection. In *Computational Intelligence, Cyber Security and Computational Models*, pages 505–518. Springer, 2016.

[50] Rafi D. M., Moses K. R. K., Petersen K., and Mäntylä M. V. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *Proceedings of the 7th International Workshop on Automation of Software Test*, pages 36–42. IEEE Press, 2012.

[51] Shirole M. and Kumar R. Uml behavioral model based test case generation: a survey. *ACM SIGSOFT Software Engineering Notes*, 38(4):1–13, 2013.

[52] Utting M., Pretschner A., and Legeard B. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.

[53] Kacker R. N., Richard K. D., Lei Y., and Lawrence J. F. Combinatorial testing for software: An adaptation of design of experiments. *Measurement*, 46(9):3745–3752, 2013.

[54] Li N., Li F., and Offutt J. Better algorithms to minimize the cost of test paths. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 280–289. IEEE, 2012.

[55] Li N., Praphamontripong U., and Offutt J. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*, pages 220–229. IEEE, 2009.

[56] Nguyen B. N., Robbins B., Banerjee I., and Memon A. Guitar: an innovative tool for automated testing of gui-driven software. *Automated Software Engineering*, 21(1):65–105, Mar 2014.

[57] Trcka N., Van der Aalst W. M. P., and Sidorova N. Data-flow anti-patterns: Discovering data-flow errors in workflows. In *CAiSE*, volume 9, pages 425–439. Springer, 2009.

[58] Ammann P. and Offutt J. *Introduction to software testing.* Cambridge University Press, 2016.

[59] Raappana P., Saukkoriipi S., Tervonen I., and MÃntylÃ M. V. The effect of team exploratory testing – experience report from f-secure. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 295–304, April 2016.

[60] Jing Q., Vasilakos A. V., Wan J., Lu J., and Qiu D. Security of the internet of things: Perspectives and challenges. *Wireless Networks*, 20(8):2481–2501, 2014.

[61] Srivastava P. R., Jose N., Barade S., and Ghosh D. Optimized test sequence generation from usage models using ant colony optimization. *International Journal of Software Engineering & Applications*, 2(2):14–28, 2010.

[62] Srivatsava P. R., Mallikarjun B., and Yang X. S. Optimal test sequence generation using firefly algorithm. *Swarm and Evolutionary Computation*, 8:44–53, 2013.

[63] Stojkoska B. L. R. and Trivodaliev K. V. A review of internet of things for smart home: Challenges and solutions. *Journal of Cleaner Production*, 140:1454–1464, 2017.

[64] Sulaiman D. R. and Ahmed B. S. Using the combinatorial optimization approach for dvs in high performance processors. In *International Conference on Technological Advances in Electrical Electronics and Computer Engineering (TAEECE)*, pages 105–109, 2013.

[65] Ahmed B. S., Sahib M. A., Gambardella L. M., Afzal W., and Zamli K. Z. Optimum design of PI$^{\lambda}$ D$^{\mu}$ controller for an automatic voltage regulator system using combinatorial test design. *PLOS ONE*, 11:1–20, 11 2016.

[66] Ahmed B. S., Sahib M. A., and Potrus M. Y. Generating combinatorial test cases using simplified swarm optimization (sso) algorithm for automated gui functional testing. *Engineering Science and Technology, an International Journal*, 17(4):218–226, 2014.

[67] Ahmed B. S., Gambardella L. M., Afzal W., and Zamli K. Z. Handling constraints in combinatorial interaction testing in the presence of multi objective particle swarm and multithreading. *Information and Software Technology*, 86:20–36, 2017.

[68] Ahmed B. S., Abdulsamad T. S., and Potrus M. Y. Achievement of minimized combinatorial test suite for configuration-aware software functional testing using the cuckoo search algorithm. *Information and Software Technology*, 66(0):13–29, 2015.

[69] Ahmed B. S. and Zamli K. Z. A review of covering arrays and their application to software testing. *Journal of Computer Science*, 7(9):1375–1385, 2011.

[70] Ahmed B. S. and Zamli K. Z. A variable strength interaction test suites generation strategy using particle swarm optimization. *Journal of Systems and Software*, 84(12):2171–2185, 2011.

[71] Ahmed B. S., Zamli K. Z., and Lim C. P. Constructing a t-way interaction test suite using the particle swarm optimization approach. *International Journal of Innovative Computing, Information and Control (IJICIC)*, 8(1):431–452, 2012.

[72] Ali S., Briand L. C., Hemmati H., and Panesar-Walawege R. K. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering*, 36(6):742–762, 2010.

[73] Anand S., Burke E. K., Chen T. Y., Clark J., Cohen M. B., Grieskamp W., Harman M., Harrold M. J., Mcminn P., et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[74] Carino S., Andrews J. H., Goulding S., Arunthavarajah P., and Hertyk J. Blackhorse: creating smart test cases from brittle recorded tests. *Software Quality Journal*, 22(2):293–310, 2014.

[75] Eldh S., Hansson H., Punnekkat S., Pettersson A., and Sundmark D. A framework for comparing efficiency, effectiveness and applicability of software testing techniques. In *Testing: Academic and Industrial Conference-Practice And Research Techniques, 2006. TAIC PART 2006. Proceedings*, pages 159–170. IEEE, 2006.

[76] Ghiduk A. S. Automatic generation of basis test paths using variable length genetic algorithm. *Information Processing Letters*, 114(6):304–316, 2014.

[77] Hoskins D. S., Colbourn C. J., and Montgomery D. C. Software performance testing using covering arrays: Efficient screening designs with categorical factors. In *Proceedings of the 5th International Workshop on Software and Performance*, WOSP '05, pages 131–136, New York, NY, USA, 2005. ACM.

[78] Meda H. S., Sen A. Kumar, and Bagchi A. On detecting data flow errors in workflows. *Journal of Data and Information Quality (JDIQ)*, 2(1):4, 2010.

[79] Moser S., Martens A., Gorlach K., Amme W., and Godlinski A. Advanced verification of distributed ws-bpel business processes incorporating cssa-based data flow analysis. In *Services Computing, 2007. SCC 2007. IEEE International Conference on*, pages 98–105. IEEE, 2007.

[80] Neto P. A. D. M. S., do Carmo Machado I., McGregor J. D., de Almeida E. S., and de Lemos Meira S. R. A systematic mapping study of software product lines testing. *Information and Software Technology*, 53(5):407–423, 2011.

[81] Shamshiri S., Just R., Rojas J. M., Fraser G., McMinn P., and Arcuri A. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 201–211. IEEE, 2015.

[82] Koomen T., Broekman B., van der Aalst L., and Vroon M. *TMap next: for result-driven testing.* Uitgeverij kleine Uil, 2013.

[83] Shiba T., Tsuchiya T., and Kikuno T. Using artificial life techniques to generate test cases for combinatorial testing. In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, pages 72–77. IEEE, 2004.

[84] Su T., Wu K., Miao W., Pu G., He J., Chen Y., and Su Z. A survey on data-flow testing. *ACM Computing Surveys (CSUR)*, 50(1):5, 2017.

[85] Arora V., Bhatia R., and Singh M. Synthesizing test scenarios in uml activity diagram using a bio-inspired approach. *Computer Languages, Systems & Structures*, 2017.

[86] Kuliamin V. V. and Petukhov A. A survey of methods for constructing covering arrays. *Programming and Computer Software*, 37(3):121–146, 2011.

[87] Chen X., Gu Q., Li A., and Chen D. Variable strength interaction testing with an ant colony system approach. In *Software Engineering Conference, 2009. APSEC'09. Asia-Pacific*, pages 160–167. IEEE, 2009.

[88] Qu X., Cohen M. B., and Woolf K. M. Combinatorial interaction regression testing: a study of test case generation and prioritization. In *IEEE International Conference on Software Maintenance, ICSM 2007*, pages 255–264. IEEE Computer Society, 2007.

[89] Sun S. X., Zhao J. L., Nunamaker J. F., and Sheng O. R. L. Formulating the data-flow perspective for business process management. *Information Systems Research*, 17(4):374–391, 2006.

[90] Paraskevopoulou Z., Hritcu C., Denes M., Lampropoulos L., and Pierce B. C. *Foundational Property-Based Testing*, pages 325–343. Springer International Publishing, Cham, 2015.

[91] Zamli K. Z, Din F., Kendall G., and Ahmed B. S. An experimental study of hyper-heuristic selection and acceptance mechanism for combinatorial t-way test suite generation. *Information Sciences*, 399:121–153, 2017.

[92] Zamli K. Z., Din F., Baharom S., and Ahmed B. S. Fuzzy adaptive teaching learning-based optimization strategy for the problem of generating mixed strength t-way test suites. *Engineering Applications of Artificial Intelligence*, 59:35–50, 2017.

[93] Zamli K. Z., Klaib M. F. J., Younis M. I., Isa N. A. M., and Abdullah R. Design and implementation of a t-way test data generation strategy with automated execution tool support. *Information Sciences*, 181(9):1741–1758, 2011.

[94] Zamli K. Z, Alkazemi B. Y, and Kendall G. A tabu search hyper-heuristic strategy for t-way test suite generation. *Applied Soft Computing*, 44:57–74, 2016.

# 7 Appendix A: Employment of Multiple Algorithms for Optimal Path-based Test Selection Strategy

[A.1] Miroslav Bures, and Bestoun S. Ahmed. Employment of Multiple Algorithms for Optimal Path-based Test Selection Strategy. Currently under review in Q1 impact factor journal. Manuscript published at arXiv.org, 1802.08005, https://arxiv.org/abs/1802.08005, 22 Feb 2018.

# Employment of Multiple Algorithms for Optimal Path-based Test Selection Strategy

Miroslav Bures and Bestoun S. Ahmed

***Abstract*—Executing various sequences of system functions in a system under test represents one of the primary techniques in software testing. The natural way to create effective, consistent and efficient test sequences is to model the system under test and employ an algorithm to generate the tests that satisfy a defined test coverage criterion. Several criteria of test set optimality can be defined. In addition, to optimize the test set from an economic viewpoint, the priorities of the various parts of the system model under test must be defined. Using this prioritization, the test cases exercise the high priority parts of the system under test more intensely than those with low priority. Evidence from the literature and our observations confirm that finding a universal algorithm that produces an optimal test set for all test coverage and test set optimality criteria is a challenging task. Moreover, for different individual problem instances, different algorithms provide optimal results. In this paper, we present a path-based strategy to perform optimal test selection. The strategy first employs a set of current algorithms to generate test sets; then, it assesses the optimality of each test set by the selected criteria, and finally, chooses the optimal test set. The experimental results confirm the validity and usefulness of this strategy. For individual instances of 50 system under test models, different algorithms provided optimal results; these results varied by the required test coverage level, the size of the priority parts of the model, and the selected test set optimality criteria.**

***Index Terms*—Model-based Testing; Path-based Test Scenarios; Test Set Optimization; Directed Graph; Edge Coverage; Edge-Pair Coverage**

## I. Introduction

THE natural way to construct a test case is to chain a sequence of specific calls to various functions of the system under test (SUT). Whether designing a method flow, or API calls are used for an integration test, or a test scenario is designed for a manual business end-to-end test, following a systematic approach that generates consistent and effective test sequences is essential. The field of model-based testing provides a solution for this issue through which we first model a particular SUT process or workflow in a suitable notation and then use an appropriate algorithm to generate the flows (i.e., path-based test cases).

To generate the path-based test cases systematically and consistently, a SUT model based on a directed graph is used [1]. Several algorithms have been presented (e.g., [2], [3], [4], [5], [6], [7]) to solve this problem. However, based on both evidence from the literature and our experiments while developing new algorithms to solve this problem, it is challenging task to find a universal algorithm that can generate an optimal test set for all instances. Not only do individual problem instances (particular SUT models) differ but also different test set optimality criteria can be formulated [3], [1], [8]. A significant finding here addresses the possibility for creating a universal algorithm that satisfies multiple optimality criteria. This task is complicated and must consider the different test coverage criteria that have been defined, which span a range from All Node Coverage to All Path Coverage, and individual algorithms differ in their ability to produce test sets that satisfy these different criteria [7].

The complexity of the problem increases when individual parts of the SUT model should be tested at different priority levels. For instance, consider a complex workflow in an information system that must be covered by path-based test scenarios. Only selected parts of the workflow require coverage by high-intensity test scenarios, while for the remaining parts, lightweight tests are sufficient to optimize the test set and reduce the testing costs. The priorities can captured by the test requirements [1], [3] or by defining edge weights in the model [9]. However, to reflect the priorities captured by edge weights when generating test cases, alternative strategies must be defined because the current algorithms provide near optimum results only for non-prioritized SUT models and can be suboptimal when solving this type of problem for prioritized SUT models.

To address the issues described above, in this study, we employ an approach based on combining current algorithms, including both our own work in this area [9] and selected algorithms previously published in the literature [3], [1]. The strategy, which includes these algorithms, relies on input from the tester as follows. The tester first creates a SUT model, defines the priority parts of the model, and specifies the test coverage criteria. Then, the tester selects the test set optimality criteria from a set of options (details are provided in Section III-B3). This strategy uses all the algorithms to generate different test sets based on the SUT model. Then, based on the test set optimality criteria, the best test set is selected and provided to the test analyst. We implemented this test case generation strategy in the latest version of the experimental Oxygen Model-based Testing platform[1] developed by the STILL group. In this paper, we present the details of this strategy and its results for 50 SUT models using Edge Coverage and Edge-Pair Coverage criteria and for 16 different test set optimality criteria (including an optimality function

M. Bures, Software Testing Intelligent Lab (STILL), Department of Computer Science, Faculty of Electrical Engineering Czech Technical University, Karlovo nam. 13, 121 35 Praha 2, Czech Republic, (email: buresm3@fel.cvut.cz)

B. Ahmed, Software Testing Intelligent Lab (STILL), Department of Computer Science, Faculty of Electrical Engineering Czech Technical University, Karlovo nam. 13, 121 35 Praha 2, Czech Republic, (email: albeybes@fel.cvut.cz)

[1] http://still.felk.cvut.cz/oxygen/

and a sequence-selection strategy composed from additional test set optimality indicators). These data can also be used to compare the test sets produced by the algorithms.

The paper is organized as follows. Section II defines the problem; then, it provides an overview of the test coverage criteria used to determine the intensity of the test set, and finally, it discusses possible test set optimality criteria. Section III provides the details of the process for selecting an optimal test set based on the optimality criteria. Section IV presents the experimental method and the acquired data. Section V discusses the results and Section VI analyzes possible threats to validity. Section VII summarizes the relevant related work. Finally, Section VIII concludes this paper.

## II. Problem Definition

As mentioned previously, the strategy presented in this paper takes a SUT model as input. Here, the SUT process is modeled as a directed graph $G = (N, E)$, where $N$ is a set of nodes, $N \neq \emptyset$, and $E$ is a set of edges. $E$ is a subset of $N \times N$. In the model we define one start node $n_s \in N$. The set $N_e \subseteq N$ contains the end nodes of the graph, and $N_e \neq \emptyset$ [1].

The SUT functions and decision points are mapped to $G$ depending on the level of abstraction. In addition, the SUT layer for which we prepare test cases plays an essential role in the modeling. As an example, we can provide data-flow testing at the code level or design an end-to-end (E2E) high-level business-process test set. More information about this topic appears in V.

The test case $t$ is a sequence of nodes $n_1, n_2, .., n_n$, with a sequence of edges $e_1, e_2, .., e_{n-1}$, where $e_i = (n_{i,} n_{i+1})$, $e_i \in E$, $n_i, \in N$, $n_{i+1} \in N$. The test case $t$ starts with the start node $n_s$ ($n_1 = n_s$) and ends with a $G$ end node ($n_n \in N_e$) . We can denote the test case $t$ as a either sequence of nodes $n_1, n_2, .., n_n$, or a sequence of edges $e_1, e_2, .., e_{n-1}$. The test set $T$ is a set of test cases.

To determine the required the test coverage, we define a set of test requirements $R$. Generally, a test requirement is a path in $G$ that must be a sub-path of at least one test case $t \in T$. The test requirements can be used either to (1) define the general intensity of the test cases or (2) to express which parts of the SUT model $G$ are considered as priorities to be covered by test cases.

The fact that the test requirements can be used either to determine the overall intensity of the test set $T$ or to express which parts of the SUT model $G$ should be tested at a higher priority leads us to adopt an alternative definition of the SUT model. This definition supports formulation of algorithms which allow determining testing intensity and expressing priorities in parallel [9]. Moreover, it uses a multigraph instead of a graph as a SUT model, which gives test analysts more flexibility when modeling SUT processes (this issue is discussed further in Section V). In addition, using more priority levels is natural in the software development process [10]; using test requirements for prioritization results in algorithms being able to work with two priority levels only, which could restrict the development of further and possibly more effective algorithms.

Our alternative definition of the SUT model is as follows. We model a SUT process as a weighted multigraph $\mathcal{G} = (N, E, s, t)$, where $N$ is a set of nodes, $N \neq \emptyset$, and $E$ is a set of edges. Here, $s : E \rightarrow N$ assigns each edge to its source node and $t : E \rightarrow N$ assigns each edge to its target node. We define one start node $n_s \in N$. The set $N_e \subseteq N$ contains the end nodes of the multigraph, $N_e \neq \emptyset$. For each edge $e \in E$ (resp. node $n \in N$), a $priority(e)$ (resp. $priority(n)$) is defined, where $priority(e) \in \{high, medium, low\}$ and $priority(n) \in \{high, medium, low\}$. When $p$ is not defined, the default $low$ value is used. $E_h$ is a set of high-priority edges; $E_m$ is a set of medium-priority edges; and $E_l$ is a set of low-priority edges, where $E_h \cup E_m \cup E_l = E$, $E_h \cap E_m = \emptyset$, $E_m \cap E_l = \emptyset$, $E_h \cap E_l = \emptyset$.

Priority $p$ reflects the importance of the edge to be tested. The test analyst determines the priority based on a risk prioritization technique [11] or a technique that combines risk assessment with information regarding the internal complexity of the SUT or the presence of defects in previous SUT versions [10]. To determine the intensity of the test set $T$, test coverage criteria are used.

### A. Test Coverage Criteria

Several different test coverage criteria have been defined for $T$. For instance, *All Edge Coverage* (or *Edge Coverage*) requires each edge $e \in E$ to be present in the test set $T$ minimally once. Alternatively, *All Node Coverage* requires each node $n \in N$ to be present in test set $T$ at least once. To satisfy the *Edge-Pair Coverage* criterion, the test set $T$ must contain each possible pair of adjacent edges in $G$ [1].

The *All Paths Coverage* (or Complete Path Coverage) requires that all possible paths in $G$, starting from $n_s$ and ending at any node of $N_e$, be present in test set $T$. Such a test set can contain considerable redundancy. To reduce this redundancy, the *Prime Path Coverage* criterion is used. To satisfy the *Prime Path Coverage* criterion, each reachable prime path in $G$ must be a sub-path of a test case $t \in T$. A path $p$ from $e_1$ to $e_2$is prime if (1) $p$ is simple, and (2) $p$ is not a sub-path of any other simple path in $G$. A path $p$ is simple when no node $n \in N$ is present more than once in $p$ (i.e., $p$ does not contain any loops); the only exception is $e_1$ and $e_2$, which can be identical (in other words, $p$ itself can be a loop) [1].

Sorted by the intensity of the test cases, *All Node Coverage* is the weakest option, followed by *All Edge Coverage*, *Edge-Pair Coverage* and *Prime Path Coverage*. The *All Paths Coverage* lies at the other end of the spectrum [1], because it implies the most intense test cases. However, due to the high number of test case steps, this option is not practicable in most software development projects. This problem can be also faced by *Prime Path Coverage*: for many routine process-testing tasks, this level of test coverage can be too extensive.

Test coverage criteria can also be specified by the *Test Depth Level (TDL)* [12]. $TDL = 1$ when $\forall e \in E$ the edge $e$ appears at least once in at least one test case $t \in T$. $TDL = x$ when the test set $T$ satisfies the following conditions: For each node $n \in N$, $P_n$ is a set of all possible paths in $G$ starting with an edge incoming to the decision point $n$, followed by a sequence of $x-1$ edges outgoing from the node $n$. Then, $\forall n \in N$ the test cases cases in test set $T$ contain all paths from $P_n$. When $TDL = 1$, it is equivalent to *All Edge Coverage*, and when $TDL = 2$, it is equivalent to *Edge-Pair Coverage*. All the coverage criteria in this section are defined in the same way for $G$ as well as for $\mathcal{G}$.

To determine the testing priority in selected parts of the SUT processes, we define a *Priority Level (PL)* for $\mathcal{G}$. $PL \in \{high, medium\}$. $PL = high$ when $\forall e \in E_h$ the edge $e$ is present at least once in at least one test case $t \in T$. Further, $PL = medium$ when $\forall e \in E_h \cup E_m$ the edge $e$ is present at least once in at least one test case $t \in T$. When a test set $T$ satisfies the *All Edge Coverage*, it also satisfies *PL*.

In the test case generation strategies we evolve, the *PL* can be combined with yet another test coverage criteria such as *TDL* [9] or *Prime Path Coverage* (refer to Section III-B4). In these cases, *PL* reduces the test coverage by *TDL* or *Prime Paths Coverage* to only the $\mathcal{G}$ parts, which are defined as the priority. It allows optimizing the test cases to exercise only the priority parts of the SUT processes or workflows.

### B. Test Set Optimality Criteria

Various optimality criteria for $T$ have been discussed in the literature (e.g., [3], [8]). Table I lists the optimality criteria used in this paper as defined for SUT model $\mathcal{G}$. Parts of these criteria can also be defined for $G$ which is captured in Table I in the column “Applicable to.”

Individual test set optimality criteria can be combined. In this paper, we explore two possible methods: combining the optimality criteria to a formula and evaluating the test set $T$ using a sequence of criteria. The following section provides more detail on the selection of the optimal test set and how these methods can be used successfully.

## III. Selecting an Optimal Test Set

To obtain an optimal test set $T$ for SUT model $\mathcal{G}$ along with test coverage and test set optimality criteria, we conduct a sequence of three main steps. First, we select a set of algorithms and their suitable input parameters to generate the $T$ for $\mathcal{G}$, the test coverage, and the test set optimality criteria. Then, we run these selected algorithms to produce the test sets $T_1..T_m$. Finally, we analyze the test sets $T_1..T_m$ and select the test set $T$ that has the best value of the optimality criteria. The **inputs** to this process are as follows:

1) SUT model $\mathcal{G}$
2) Test coverage criteria from the following options:
   a) Test intensity from the following options: *Edge Coverage*, *Edge-Pair Coverage*, $TDL$ (where $TDL > 2$, because $TDL = 1$ is equivalent to Edge Coverage and $TDL = 2$ is equivalent to Edge-Pair coverage), and *Prime Path Coverage.*
   b) Coverage of the $\mathcal{G}$ priority parts by *Priority Level (PL)* as defined in Section II-A.
3) Test set optimality criterion from the options defined in Section II-B and Table I.

The **output** of the process is an optimal test set $T$, that satisfies the test coverage criterion.

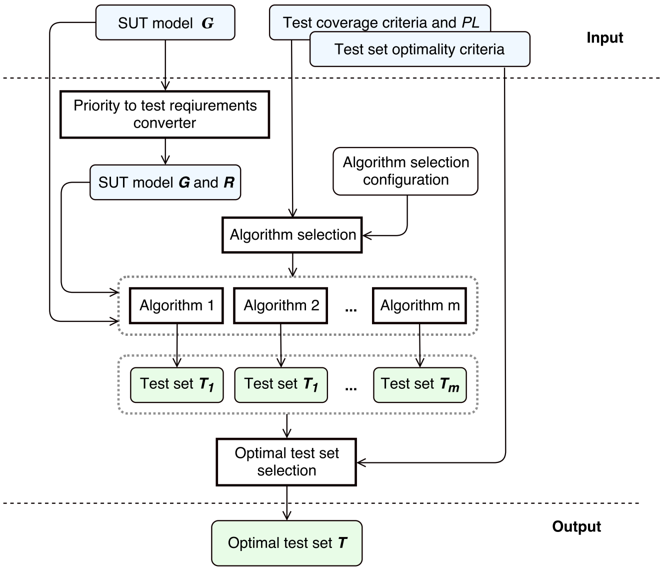

SUT model G
Test coverage criteria and PL
Test set optimality criteria
Input
Priority to test reqiurements converter
SUT model G and R
Algorithm selection configuration
Algorithm selection
Algorithm 1
Algorithm 2
...
Algorithm m
Test set T1
Test set T1
...
Test set Tm
Optimal test set selection
Optimal test set T
Output

Fig. 1. Main steps of the proposed test case generation strategy

### A. Included Algorithms

In the described strategy, we use the algorithms listed in Table II.

We used the Oxygen Model-based Testing experimental platform[2] (formerly PCTgen) [13] to implement the proposed strategy. Our research team implemented process Cycle Test (PCT) and Prioritized Process Test (PPT). We also implemented the Brute Force Solution (BF) algorithm based on the pseudocode published by Li et al. [3]. The implementations of the Set-Covering Based Solution (SC) and Matching-Based Prefix Graph Solution (PG) algorithms is based on the source code by Ammann and Offutt [14]. The Set-Covering Based Solution with Test Case Reduction (RSC) consists of the Set-Covering Based Solution part and our implementation of the test set reduction part (further specified in Section III-B4).

The role of the PCT algorithm is only to provide information on how many test cases and test steps are in a test set $T$ for a particular SUT model $\mathcal{G}$ when the SUT model parts are not prioritized.

### B. Test Set Generation Process

The strategy to determine the optimal test set $T$ by the selected test set optimality criterion consists of five main steps, which are summarized in Algorithm 1.

Figure 1 depicts the overall process. The process inputs are marked in blue while the process outputs are marked in green.

In the Oxygen platform, the $T$ is presented to the user, as well as $T_1..T_m$. For each of $T_1..T_m$ , Oxygen also provides the values of the optimality criteria. In the following subsections, we explain these individual steps in more detail.

*1) Conversion of $\mathcal{G}$ to $G$ and $R$:* For the BF, SC, PG and RSC algorithms, we need to convert graph $\mathcal{G}$ to graph $G$ and a set of test requirements, $R$. The multigraph$\mathcal{G}$ is equivalent to graph $G$, when (1) edge priorities ($priority(e)$) and node

[2] http://still.felk.cvut.cz/oxygen/

TABLE I
TEST SET $T$ OPTIMALITY CRITERIA

| Optimality criterion | Description | Applicable to |
|---|---|---|
| $\mid T \mid$ | Number of test cases in the test set | $\mathcal{G}$ and $G$ |
| $edges(T)$ | Total number of edges in the test cases of a test set $T$, edges can repeat | $\mathcal{G}$ and $G$ |
| $edges_h(T)$ | Total number of edges of priority $high$ in the test cases of a test set $T$, edges can repeat | $\mathcal{G}$ |
| $edges_m(T)$ | Total number of edges of priority $high$ and $medium$ in the test cases of a test set $T$, edges can repeat | $\mathcal{G}$ |
| $uedges(T)$ | Total number of unique edges in the test cases of a test set $T$ | $\mathcal{G}$ and $G$ |
| $uedges_h(T)$ | Total number of unique edges of priority $high$ in the test cases of a test set $T$ | $\mathcal{G}$ |
| $uedges_m(T)$ | Total number of unique edges of priority $high$ and $medium$ in the test cases of a test set $T$ | $\mathcal{G}$ |
| $nodes(T)$ | Total number of nodes in the test cases of a test set $T$, nodes can repeat | $\mathcal{G}$ and $G$ |
| $unodes(T)$ | Total number of unique nodes in the test cases of a test set $T$ | $\mathcal{G}$ |
| $er(T) = \frac{uedges(T)}{\lvert E \rvert}.100\%$ | Ratio of unique edges contained in the test cases of a test set $T$. A lower value of $er(T)$ means more optimal test set, because less unique edges are present in the test cases and these unique edges can represent extra costs for preparation of the detailed test scenarios. | $\mathcal{G}$ and $G$ |
| $e_h(T) = \frac{edges_h(T)}{edges(T)}.100\%$ | Ratio of edges of priority $high$ and all edges in the test cases of a test set $T$. A higher value of $e_h(T)$ means more optimal test set, because less edges which do not have priority $high$ (thus are not necessary to test) are present in the test cases. | $\mathcal{G}$ |
| $e_m(T) = \frac{edges_m(T)}{edges(T)}.100\%$ | Ratio of edges of priority $high$ and $medium$ and all edges in the test cases of a test set $T$. A higher value of $e_h(T)$ means more optimal test set, because less edges which do not have priority $high$ and $medium$ (thus are not necessary to test) are present in the test cases. | $\mathcal{G}$ |
| $ue_h(T) = \frac{uedges_h(T)}{edges(T)}.100\%$ | The same as $e_h(T)$, only unique edges are taken in acount | $\mathcal{G}$ |
| $ue_m(T) = \frac{uedges_m(T)}{edges(T)}.100\%$ | The same as $e_m(T)$, only unique edges are taken in acount | $\mathcal{G}$ |

TABLE II
ALGORITHMS USED TO GENERATE $T$

| Code | Name | SUT model | Reference |
|---|---|---|---|
| PCT | Process Cycle Test | $G$ | Koomen et al.[12], Bures [13] |
| PPT | Prioritized Process Test | $\mathcal{G}$ | Bures et al. [9] |
| BF | Brute Force Solution | $G$, set of test requirements $R$ | Li, Li and Offut [3] |
| SC | Set-Covering Based Solution | $G$, set of test requirements $R$ | Li, Li and Offut [3] |
| PG | Matching-Based Prefix Graph Solution | $G$, set of test requirements $R$ | Li, Li and Offut [3] |
| RSC | Set-Covering Based Solution with Test Set Reduction | $\mathcal{G}$ for the whole algorithm,$G$ and $R$ for its SC part | Specified in Section *III-B4* |

**Algorithm 1:** The main process of test set generation

```
Input: $\mathcal{G}$, test coverage criteria, $PL \in \{high, medium\}$, test set optimality criterion
Output: test set $T$
1 Convert $\mathcal{G}$ to $G$ and $R$ for the BF, SC, PG and RSC algorithms (refer to Section III-B1)
2 Determine the set of algorithms that are suitable for generating the T for $\mathcal{G}$ (refer to Section III-B1)
3 Execute the selected set of algorithms with $\mathcal{G}$ (or with $G$ and $R$, which correspond to $\mathcal{G}$). The output of this step are the
    test sets $T_1 \ldots T_m$
4 Compute the values of single test set optimality criteria for $T_1 \ldots T_m$ (refer to Table I and Section III-B3), which will be
    employed in step 5.
5 Select the optimal $T$ of $T_1 \ldots T_m$ as determined by the test set optimality criterion (refer to Section III-B3)
```

priorities ($priority(n)$) are not considered, and (2) there are no parallel edges in $\mathcal{G}$. This conversion implies that when creating $\mathcal{G}$, no parallel edges can be used, which can restrict the modeling possibilities that using a multigraph as a SUT process abstraction makes possible. However, this restriction can be solved without losing the applicability of the proposed strategy by modeling the parallel edges as graph nodes.

A set of test requirements $R$ is created by a method specified in Table III.

*2) Algorithms Selection:* Table IV specifies the process of selecting algorithms by the specified test coverage criteria (algorithm selection configuration as depicted in Fig. 1).

For the *Edge Coverage* case ($TDL = 1$), the BF, SC and PG algorithms reflect the edge priorities in $\mathcal{G}$ via a set of test requirements, $R$, generated from $G$ (refer to Table III). In both conversion types, the Atomic and the Sequence conversions are used for each of these algorithms. In contrast, PPT and RSC work directly with the edge priorities in $\mathcal{G}$ ([9] and Section III-B4).

For the *Edge-Pair Coverage* case ($TDL = 2$), the PPT and RSC algorithms are comparable candidates when the $PL$ criterion reduces the test set. The RSC satisfies the *Edge-Pair Coverage* criterion, because the test set produced by this algorithm satisfies the *Prime Paths Coverage* criterion [1]. The PPT algorithm is designed to satisfy $1 \leq TDL \leq n$, where

TABLE III
METHOD OF CREATION OF THE TEST REQUIREMENTS $R$ FROM $\mathcal{G}$

| Test coverage: **test intensity** | Method of $R$ creation | $PL = high$ | $PL = medium$ |
|---|---|---|---|
| *Edge Coverage* ($TDL = 1$) | Atomic conversion | $R$ is a set of all $G$ adjacent node pairs $e = (n_i, n_{i+1})$ for each $e \in E_h$ | $R$ is a set of all $G$ adjacent node pairs $e = (n_i, n_{i+1})$ for each $e \in E_h \cup E_m$ |
| *Edge Coverage* ($TDL = 1$) | Sequence conversion | $R$ is a set of paths in $\mathcal{G}$, $priority(e) = high$for each $e \in p \in R$ | $R$ is a set of paths in $\mathcal{G}$, $priority(e) \in \{high, medium\}$for each $e \in p \in R$ |
| *Edge-Pair Coverage* ($TDL = 2$) | A set $E_{pair}$ contains all possible pairs of adjacent edges of $\mathcal{G}$. Then, $R$ is a set of all paths $(n_i, n_{i+1}, n_{i+2})$, such that $e_i = (n_{i,}, n_{i+1})$, $e_{i+1} = (n_{i+1,} n_{i+2})$ for each $(e_i, e_{i+1}) \in E_{pair}$. This process is not influenced by $PL$. | | |
| $TDL = x$, $x > 2$ | A set $E_x$ contains all possible paths of $\mathcal{G}$ consisting of $x$ adjacent edges. Then, $R$ is a set of all paths $(n_i, n_{i+1}, n_{i+2}, .., n_{x+1})$, such that $e_i = (n_{i,}, n_{i+1})$, $e_{i+1} = (n_{i+1,} n_{i+2})$, ... , $e_x = (n_{x,} n_{x+1})$ for each $(e_i, e_{i+1}, .., e_x) \in E_{pair}$. This process is not influenced by $PL$. | | |
| *Prime Path Coverage* | $R$ is a set of all possible prime paths in $\mathcal{G}$ (applies to BF, SC and PG). This process is not influenced by $PL$. | | |

TABLE IV
ALGORITHM SELECTION CONFIGURATION

| | Test set reduction: **coverage of priority parts of** $\mathcal{G}$ | | |
|---|---|---|---|
| Test coverage: **test intensity** | $PL = high$ | $PL = medium$ | not reduced by $PL$ |
| *Edge Coverage* ($TDL = 1$) | PPT<br>RSC<br>BF<br>SC<br>PG | PPT<br>RSC<br>BF<br>SC<br>PG | PCT |
| *Edge-Pair Coverage* ($TDL = 2$) | PPT<br>RSC | PPT<br>RSC | PCT<br>BF<br>SC<br>PG |
| $TDL > 2$ | PPT | PPT | PCT<br>BF<br>SC<br>PG |
| *Prime Path Coverage* | RSC | RSC | BF<br>SC<br>PG |

$n$ is the length of the longest path in $\mathcal{G}$ (excluding the loops) [9]. Thus, it satisfies the *Edge-Pair Coverage* criterion, which is equivalent to $TDL = 2$.

*3) Selection of the Best Test Set:* After the selected algorithms have produced the test sets$T_1..T_m$, our strategy selects the test set $T$, which has the best value of the optimality criteria. The test analyst can select the following options:

1) **Selection by single optimality criterion:** A specific optimality criterion is specified on input. Then, the test set $T$ that has the best value according to the specified optimality criterion is selected. The following options are available: $\mid T \mid$, $edges(T)$, $edges_h(T)$, $edges_m(T)$, $uedges(T)$, $uedges_h(T)$, $uedges_m(T)$, $nodes(T)$, $unodes(T)$, $er(T)$ (the $T$ with the lowest value of these criteria is considered as optimal) and $e_h(T)$, $e_m(T)$, $ue_h(T)$ and $ue_m(T)$ (the $T$ with the highest value of these criteria is considered as optimal). However, a test set $T_1$ could be optimal according to one criterion, for instance $\mid T_1 \mid$, but be strongly sub-optimal according to another criterion, for instance $edges(T_1)$. In such situations, a test set $T_2$ with a slightly higher$\mid T_2 \mid$ value but whose $edges(T_2)$ were closer to the optimum would be a better choice. In these situations, we use the optimality function explained below.
2) **Selection by the optimality function:** The optimality function selects the best test set using several concurrent optimality criteria, and it is defined as $o(T_x) = w_{|T|}(1 - \frac{|T_x|}{\frac{\sum_{T \in T_1..T_m} |T|}{m}}) + w_{edges(T)}(1 - \frac{edges(T_x)}{\frac{\sum_{T \in T_1..T_m} edges(T)}{m}}) + w_{uedges(T)}(1 - \frac{uedges(T_x)}{\frac{\sum_{T \in T_1..T_m} uedges(T)}{m}})$. The constants $w_{|T|}$, $w_{edges(T)}$ and $w_{uedges(T)}$ determine the weight of each specific optimality criterion, $0 \leq w_{|T|} \leq 1$, $0 \leq w_{edges(T)} \leq 1$, $0 \leq w_{uedges(T)} \leq 1$ and $w_{|T|} + w_{edges(T)} + w_{uedges(T)} = 1$.
3) **Sequence selection:** In this approach, a sequence of optimality criteria $c_1..c_n$, is specified on input. When multiple test sets in $T_1..T_m$ have the same best value of $c_1$, the best $T$ selection is then based on the $c_2$ criterion. If multiple test sets still have the same best value of $c_2$, the selection of the final $T$ is based on $c_3$ and so forth.

*4) RSC Algorithm:* The pseudocode for the Set-Covering Based Solution with Test Set Reduction (RSC) is specified in Algorithm 2.

The principle underlying the RSC is to first employ the SC algorithm to generate the test set satisfying the Prime Path Coverage (denoted as $P$). Then, from $P$, the test cases that cover the maximal number of priority edges that must be present in the test cases are utilized to build test set $T$

**Algorithm 2:** Set-Covering Based Solution with Test Set Reduction

```
Input: $\mathcal{G}$, $G$, $PL \in \{high, medium\}$
Output: test set $T$
$P \leftarrow$ test cases satisfying the Prime Path Coverage in $G$ generated by the SC algorithm
$COVER \leftarrow E_h$ for $PL = high$
$COVER \leftarrow E_h \cup E_m$ for $PL = medium$
$T \leftarrow \emptyset$
while $COVER \neq \emptyset$ do
    select $p \in P$ such that $p \cap COVER$ is maximal
    $ADD \leftarrow p \cap COVER$
    $T \leftarrow T + \{p\}$
    $COVER \leftarrow COVER \setminus ADD$
end
// verification of $T$ completeness
if $PL = high$ then
    $V \leftarrow E_h$
end
else if $PL = medium$ then
    $V \leftarrow E_h \cup E_m$
end
foreach $e \in V$ do
    if $e \notin$ any $t \in T$ then
        $T$ is invalid
    end
end
```

incrementally.

## IV. Experiments

In this section, we describe some experiments performed to demonstrate the functionality of the proposed test set selection strategy. The provided data can also be used to compare the test sets produced by the individual algorithms and when using various optimality criteria.

### *A. Experimental Method and Set-up*

In the experiments, we execute the algorithms for the following configurations of the test coverage criteria:

1) ***Edge Coverage:*** reduced by $PL = high$ and $PL = medium$. In this experiment, we compared PPT, RSC, BF, SC and PG. For BF, SC and PG, the priorities in $\mathcal{G}$ were converted to $R$ by both Atomic and Sequence conversion (see Table III).
2) ***Edge-Pair Coverage:*** reduced by $PL = high$ and $PL = medium$. In this experiment, we compared PPT and RSC.

The *Prime Path Coverage* criterion was not involved in the experiments, as its reduction by $PL$ is possible only by the RSC algorithm. Hence, no alternative algorithm was available for comparison. The same situation is applicable for $TDL > 2$, where the PPT algorithm is the only option, which reduces the test cases by $PL$.

Regarding the problem instances, we used 50 SUT models specified by $\mathcal{G}$. To ensure the objective comparability of all algorithms (and the convertibility of $\mathcal{G}$ to $G$ and $R$), the graphs did not contain parallel edges (the SUTs were modeled so that parallel edges were not needed). The models were created in the user interface of the Oxygen platform [13]. The properties of these models are summarized in Table V. $\mid E_l \mid = \mid E \mid - \mid E_h \mid - \mid E_m \mid$. For the atomic conversion of test requirements (see Table III), $R_{ha}$ (resp. $R_{ma}$) denotes a set of test requirements for $PTL = high$ (resp. $PTL = medium$). For the sequence conversion of test requirements, $R_{hs}$ (resp. $R_{ms}$) denotes a set of test requirements for $PTL = high$ (resp. $PTL = medium$). In Table V, we present only$\mid R_{hs} \mid$ and $\mid R_{ms} \mid$, because $\mid R_{ha} \mid = \mid E_h \mid$ and $\mid R_{hm} \mid = \mid E_h \mid + \mid E_m \mid$. The number of loops in $\mathcal{G}$ is denoted by $loops$.

We compared all the options of test set optimality criteria introduced in Section III-B3: a set of single optimality criteria, an optimality function and sequence selection.

The test set selection process described in this paper is implemented as part of the development branch of the Oxygen platform. All the test set optimality criteria discussed in this paper are calculated automatically from the produced test cases and provided in a CSV-formatted report. In the report, the test set $T$ selected by the particular optimality criteria is also presented, including the algorithm that generated this test set.

### *B. Experimental Results*

In this section, we present a performance comparison of the PPT, RSC, BF, SC, and PG algorithms using all the test set optimality criteria discussed in Section II-B. For the comparison that appears in Section IV-B1 we used the data from Step 3 of Algorithm 1. Then, in Section IV-B2, we

TABLE V
PROBLEM INSTANCES USED IN THE EXPERIMENTS

| **ID** | $\mid N \mid$ | $\mid E \mid$ | $\mid E_h \mid$ | $\mid E_m \mid$ | $loops$ | $\mid R_{hs} \mid$ | $\mid R_{ms} \mid$ | **ID** | $\mid N \mid$ | $\mid E \mid$ | $\mid E_h \mid$ | $\mid E_m \mid$ | $loops$ | $\mid R_{hs} \mid$ | $\mid R_{ms} \mid$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 22 | 30 | 8 | 8 | 1 | 6 | 7 | **26** | 51 | 67 | 15 | 7 | 0 | 9 | 12 |
| **2** | 21 | 30 | 8 | 3 | 4 | 5 | 6 | **27** | 28 | 39 | 8 | 4 | 1 | 5 | 8 |
| **3** | 41 | 54 | 10 | 10 | 4 | 9 | 11 | **28** | 21 | 22 | 5 | 2 | 0 | 2 | 3 |
| **4** | 29 | 46 | 11 | 4 | 11 | 9 | 9 | **29** | 29 | 37 | 10 | 6 | 0 | 4 | 7 |
| **5** | 29 | 45 | 17 | 6 | 6 | 15 | 19 | **30** | 9 | 11 | 1 | 4 | 0 | 1 | 2 |
| **6** | 21 | 27 | 9 | 6 | 0 | 8 | 13 | **31** | 10 | 13 | 1 | 4 | 0 | 1 | 2 |
| **7** | 45 | 64 | 18 | 14 | 7 | 15 | 30 | **32** | 25 | 27 | 3 | 5 | 0 | 3 | 5 |
| **8** | 19 | 30 | 10 | 6 | 6 | 9 | 16 | **33** | 11 | 15 | 1 | 2 | 0 | 1 | 3 |
| **9** | 25 | 38 | 11 | 9 | 9 | 8 | 13 | **34** | 13 | 19 | 3 | 4 | 0 | 2 | 6 |
| **10** | 52 | 78 | 9 | 7 | 3 | 6 | 10 | **35** | 10 | 15 | 4 | 2 | 4 | 4 | 4 |
| **11** | 48 | 69 | 10 | 8 | 3 | 4 | 8 | **36** | 8 | 10 | 1 | 3 | 3 | 1 | 4 |
| **12** | 47 | 68 | 9 | 11 | 1 | 5 | 8 | **37** | 8 | 11 | 3 | 2 | 3 | 3 | 3 |
| **13** | 23 | 26 | 9 | 6 | 2 | 5 | 5 | **38** | 7 | 12 | 2 | 3 | 5 | 2 | 3 |
| **14** | 8 | 10 | 1 | 3 | 2 | 1 | 4 | **39** | 8 | 11 | 3 | 2 | 2 | 2 | 4 |
| **15** | 24 | 31 | 10 | 4 | 0 | 2 | 3 | **40** | 7 | 9 | 2 | 2 | 0 | 2 | 3 |
| **16** | 26 | 37 | 10 | 3 | 2 | 3 | 4 | **41** | 9 | 11 | 2 | 3 | 0 | 2 | 4 |
| **17** | 27 | 36 | 6 | 7 | 3 | 6 | 10 | **42** | 11 | 14 | 2 | 2 | 2 | 2 | 4 |
| **18** | 20 | 26 | 1 | 8 | 2 | 1 | 2 | **43** | 22 | 27 | 5 | 7 | 0 | 4 | 9 |
| **19** | 28 | 34 | 1 | 2 | 0 | 1 | 3 | **44** | 26 | 38 | 6 | 3 | 3 | 6 | 7 |
| **20** | 9 | 8 | 3 | 2 | 0 | 3 | 4 | **45** | 29 | 45 | 8 | 4 | 4 | 7 | 11 |
| **21** | 8 | 10 | 2 | 2 | 2 | 2 | 3 | **46** | 35 | 48 | 5 | 9 | 4 | 5 | 11 |
| **22** | 34 | 47 | 13 | 5 | 0 | 7 | 9 | **47** | 40 | 54 | 8 | 6 | 0 | 8 | 11 |
| **23** | 35 | 49 | 8 | 3 | 0 | 7 | 10 | **48** | 50 | 74 | 13 | 6 | 6 | 13 | 17 |
| **24** | 37 | 55 | 16 | 5 | 2 | 9 | 13 | **49** | 21 | 27 | 12 | 6 | 0 | 2 | 3 |
| **25** | 41 | 59 | 15 | 6 | 0 | 10 | 13 | **50** | 22 | 23 | 8 | 2 | 0 | 2 | 3 |

provide the results of each specific algorithm selection, which are the output of Algorithm 1.

*1) Comparison of Individual Algorithms:* In Table VI, we present the averaged values of the optimality criteria of the test sets produced for the individual SUT models used in the experiments (introduced previously in Table V). Table VI presents the numbers for *Edge Coverage* and $PL = high$. The atomic and sequence conversions of test requirements $R$ (see Table III) are denoted by "*atom*" and "*seq*" in the table.

Figures 2 and 3 provide a visual comparison, using these averaged values to compare the individual algorithms.

Table VII lists the averaged values of the optimality criteria of the test sets produced for the individual SUT models for the *Edge Coverage* and $PL = medium$ criterion and $PL = medium$.

To better compare the algorithms using the values presented in VII, Figures 4 and 5 shows a graphical summary.

Table VIII shows a comparison of the PPT and RSC algorithms for the *Edge-Pair Coverage* criterion and $PL \in \{high, medium\}$. Here, the only relevant algorithms are PPT and RSC, because these algorithms allow test set reduction by PL and can satisfy the *Edge-Pair Coverage* criterion before the test set reduction by PL.

A comparison of the individual algorithms for the *Edge-Pair Coverage* criterion is depicted in Figures 6 and 7 for $PL = high$.

Figures 8 and 9 depict this comparison for $PL = medium$.

We analyze the data in Section V.

*2) Algorithm Selection Results:* In this section, we present the results related to the functionality of the proposed test set selection strategy. We start with detailed data that demonstrate the test set selection strategy. Table IX summarizes the execution of the test set selection strategy for the *Edge Coverage* criterion and $PL = high$. For each of the problem instances and optimality criteria, we list the algorithm that produced the optimal $T$ based on the selected criteria. Table IX presents the results of the first twelve selected problem instances as an example.

**OPT** denotes an optimality function (refer to Section III-B3). In all the experiments described in this paper, the optimality function was configured with parameters $w_{|T|} = 0.3$, $w_{edges(T)} = 0.4$ and $w_{uedges(T)} = 0.3$.

**SEQ** denotes sequence selection (refer to Section III-B3). In all of the experiments, the following sequence of optimality criteria was adopted: $\mid T \mid$, $edges(T)$, $uedges(T)$. The selection sequence starts with $\mid T \mid$.

In Table IX we present the name of the algorithm (or algorithms) that produced the optimal test set based on the particular criterion. For simple optimality criteria, the algorithm name is followed by the value of the optimality criterion (in brackets). When multiple algorithms can provide an optimal $T$ for a particular criterion, more algorithms are listed. Cases in which more than three algorithms provided an optimal $T$ for a particular criterion are denoted as $n(M)$, where $n$ denotes the number of algorithms and $M$ denotes the value of the optimality criterion.

Atomic and sequence conversions of test requirements $R$ (refer to Table III) are denoted by "*a*" and "*s*" postfixes, respectively, in italics following the name of the algorithm. ID denotes the ID of the problem instance $\mathcal{G}$.

Figure 10 shows the overall statistics for the test set selection strategies for the *Edge Coverage* criterion and $PL = high$. The x-axis reflects the optimality criteria, and the y-axis presents the individual algorithms. The bubble size represents the number of problem instances for which an algorithm produced an optimal $T$ using a specific criterion. The maximum bubble size is 50 (the number of problem instances). For the case of $uedges_h(T)$, all the algorithms achieved the maximum value, as $uedges_h(T) =\mid E_h \mid$ for $T$ in all the cases.

Using the same schema, Figure 11 shows the overall sta-

TABLE VI
RESULTS OF THE ALGORITHMS FOR *Edge Coverage* AND $PL = high$

| | Algorithm / $R$ creation method | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Value of optimality criterion - average for all $\mathcal{G}$ | PPT | RSC | BF *atom* | BF *seq* | SC *atom* | SC *seq* | PG *atom* | PG *seq* |
| $\mid T \mid$ | 2.88 | 3.20 | 5.04 | 4.40 | 5.10 | 4.36 | 4.20 | 4.26 |
| $edges(T)$ | 23.40 | 32.62 | 36.10 | 32.68 | 36.92 | 32.74 | 31.30 | 32.12 |
| $edges_h(T)$ | 8.92 | 11.22 | 12.24 | 11.34 | 13.12 | 11.64 | 11.04 | 11.46 |
| $edges_m(T)$ | 11.64 | 15.50 | 16.64 | 15.30 | 17.42 | 15.60 | 14.74 | 15.34 |
| $uedges(T)$ | 17.08 | 18.90 | 19.64 | 18.76 | 18.86 | 18.02 | 17.96 | 17.96 |
| $uedges_h(T)$ | 7.12 | 7.12 | 7.12 | 7.12 | 7.12 | 7.12 | 7.12 | 7.12 |
| $uedges_m(T)$ | 8.86 | 9.28 | 9.30 | 9.20 | 9.22 | 9.18 | 9.14 | 9.16 |
| $nodes(T)$ | 20.52 | 29.42 | 31.06 | 28.28 | 31.82 | 28.38 | 27.10 | 27.86 |
| $unodes(T)$ | 15.52 | 17.28 | 17.88 | 17.06 | 17.18 | 16.40 | 16.32 | 16.36 |
| $er(T)$ | 0.50 | 0.57 | 0.56 | 0.54 | 0.54 | 0.52 | 0.52 | 0.52 |
| $e_h(T)$ | 0.41 | 0.38 | 0.36 | 0.39 | 0.38 | 0.40 | 0.40 | 0.40 |
| $e_m(T)$ | 0.53 | 0.52 | 0.49 | 0.51 | 0.49 | 0.52 | 0.52 | 0.52 |
| $ue_h(T)$ | 0.36 | 0.28 | 0.25 | 0.29 | 0.25 | 0.30 | 0.30 | 0.30 |
| $ue_m(T)$ | 0.45 | 0.38 | 0.33 | 0.38 | 0.33 | 0.38 | 0.39 | 0.39 |

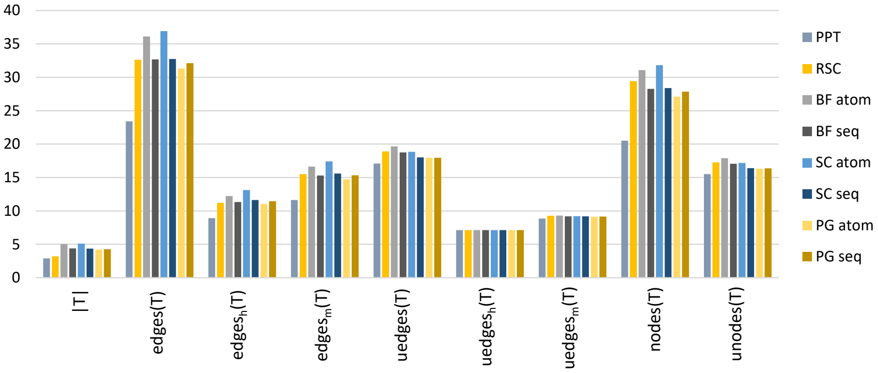


Fig. 2. Algorithm comparison for *Edge Coverage* and $PL = high$

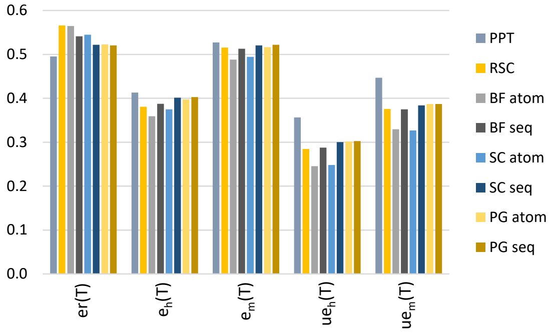


Fig. 3. Algorithm comparison for *Edge Coverage* and $PL = high$

tistical results of the test set selection strategies for the *Edge Coverage* criterion and $PL = medium$. In this case, all the employed algorithms provided a $T$ having $uedges_h(T) = \mid E_h \mid$ and $uedges_m(T) = \mid E_h \mid + \mid E_m \mid$.

The same system is used in Figure 12, which depicts the overall statistics of the test set selection strategies for *Edge-Pair Coverage* and $PL = high$, and in Figure 13, which presents the statistics for *Edge-Pair Coverage* and $PL = medium$.

## V. DISCUSSION

From the data presented in Section IV-B, several conclusions can be made.

Starting with a **comparison of the algorithms** using the average values of optimality criteria computed for 50 different problem instances (Table V), the results differ significantly based on the test coverage level. For *Edge Coverage*, the PPT algorithm provides the best results in terms of average statistics. The difference between the average value for PPT and the other algorithms is the most significant for the optimality criteria$\mid T \mid$, $edges(T)$ and $nodes(T)$. This result can be observed for both $PL = high$ (Table VI, Figure 2) and $PL = medium$ (Table VII, Figure 4). For $PL = high$, the difference in $\mid T \mid$ between PPT and RSC is 10%, and it is greater than 31% between PPT and each of the BF, SC and PG algorithms. The difference in $edges(T)$ between PPT and all the other algorithms is greater than 25%. For $nodes(T)$, this difference is greater than 24%. For $PL = medium$, the differences are slightly lower in general. For $\mid T \mid$, the difference between PPT and RSC is 8%, and it is greater than 27% between PPT and each of the BF, SC and PG algorithms. The difference between PPT and all the other algorithms is greater than 21% for $edges(T)$ and greater than 20% for $nodes(T)$.

Additionally, for the optimality criteria based on unique priority edges, $ue_h(T)$ and $ue_m(T)$, the average value of

TABLE VII
RESULTS OF THE ALGORITHMS FOR *Edge Coverage* AND $PL = medium$

| Value of optimality criterion - average for all $\mathcal{G}$ | Algorithm / $R$ creation method | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | PPT | RSC | BF *atom* | BF *seq* | SC *atom* | SC *seq* | PG *atom* | PG *seq* |
| $\mid T \mid$ | 4.38 | 4.76 | 7.66 | 6.80 | 7.60 | 6.52 | 5.96 | 6.20 |
| $edges(T)$ | 35.34 | 46.04 | 54.18 | 50.46 | 54.20 | 48.94 | 44.84 | 47.28 |
| $edges_h(T)$ | 10.56 | 13.26 | 15.40 | 15.06 | 16.32 | 15.30 | 13.48 | 14.86 |
| $edges_m(T)$ | 17.22 | 21.54 | 24.44 | 24.24 | 25.38 | 24.28 | 21.38 | 23.48 |
| $uedges(T)$ | 22.60 | 24.06 | 24.72 | 23.60 | 23.98 | 22.76 | 22.80 | 22.68 |
| $uedges_h(T)$ | 7.12 | 7.12 | 7.12 | 7.12 | 7.12 | 7.12 | 7.12 | 7.12 |
| $uedges_m(T)$ | 12.08 | 12.08 | 12.08 | 12.08 | 12.08 | 12.08 | 12.08 | 12.08 |
| $nodes(T)$ | 30.96 | 41.28 | 46.52 | 43.66 | 46.60 | 42.42 | 38.88 | 41.08 |
| $unodes(T)$ | 20.80 | 22.28 | 22.84 | 21.80 | 22.16 | 21.04 | 21.06 | 20.96 |
| $er(T)$ | 0.69 | 0.74 | 0.75 | 0.72 | 0.74 | 0.70 | 0.70 | 0.69 |
| $e_h(T)$ | 0.30 | 0.29 | 0.28 | 0.31 | 0.29 | 0.32 | 0.31 | 0.32 |
| $e_m(T)$ | 0.53 | 0.51 | 0.47 | 0.53 | 0.48 | 0.54 | 0.52 | 0.54 |
| $ue_h(T)$ | 0.22 | 0.18 | 0.14 | 0.16 | 0.14 | 0.17 | 0.18 | 0.17 |
| $ue_m(T)$ | 0.42 | 0.33 | 0.26 | 0.31 | 0.25 | 0.33 | 0.34 | 0.33 |

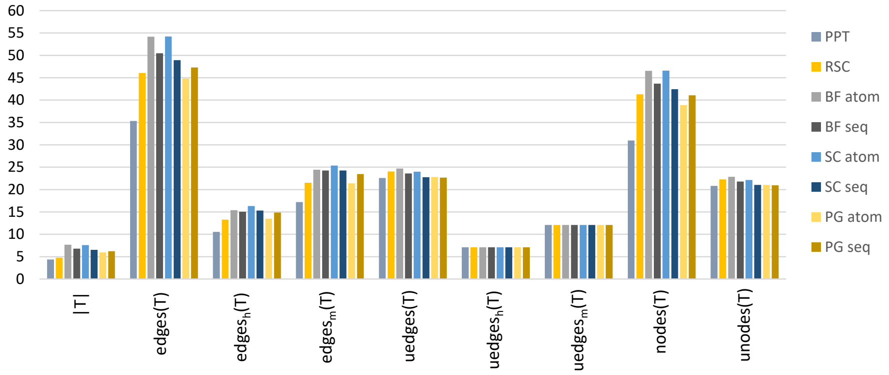


Fig. 4. Algorithm comparison for *Edge Coverage* and $PL = medium$

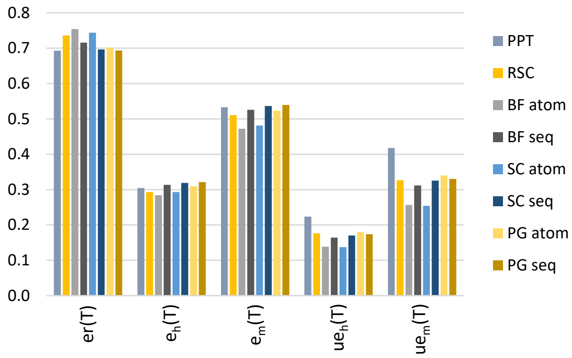


Fig. 5. Algorithm comparison for *Edge Coverage* and $PL = medium$

TABLE VIII
RESULTS OF THE ALGORITHMS FOR EDGE-PAIR COVERAGE

| Value of optimality criterion - average for all $\mathcal{G}$ | $PL = high$ | | $PL = medium$ | |
|---|---|---|---|---|
| | PPT | RSC | PPT | RSC |
| $\mid T \mid$ | 5.28 | 3.20 | 7.78 | 4.76 |
| $edges(T)$ | 43.74 | 32.62 | 62.88 | 46.04 |
| $edges_h(T)$ | 15.50 | 11.22 | 18.32 | 13.26 |
| $edges_m(T)$ | 20.72 | 15.50 | 29.18 | 21.54 |
| $uedges(T)$ | 23.04 | 18.90 | 28.00 | 24.06 |
| $uedges_h(T)$ | 7.12 | 7.12 | 7.12 | 7.12 |
| $uedges_m(T)$ | 9.92 | 9.28 | 12.08 | 12.08 |
| $nodes(T)$ | 38.46 | 29.42 | 55.10 | 41.28 |
| $unodes(T)$ | 21.46 | 17.28 | 26.30 | 22.28 |
| $er(T)$ | 0.65 | 0.57 | 0.86 | 0.74 |
| $e_h(T)$ | 0.36 | 0.38 | 0.27 | 0.29 |
| $e_m(T)$ | 0.49 | 0.52 | 0.47 | 0.51 |
| $ue_h(T)$ | 0.20 | 0.28 | 0.12 | 0.18 |
| $ue_m(T)$ | 0.29 | 0.38 | 0.21 | 0.33 |

optimality criteria differs significantly for the PPT algorithm. Higher values of $ue_h(T)$ and $ue_m(T)$ result in test sets closer to optimum. For $PL = high$, PPT is higher than all the other algorithms by 18% for $ue_h(T)$ and 15% for $ue_m(T)$.

For the rest of the optimality criteria, the differences are not as significant; however, similar results are still present in the data.

Generally, the RSC algorithm yields results relatively similar to the BF, PG and SC algorithms; however, exceptions can be found. For $PL = high$ (Table VI, Figure 2), the RSC algorithm is outperformed by the PG and SC algorithms for $uedges(T)$, $nodes(T)$, $unodes(T)$, and $er(T)$. At this priority level, the RSC algorithm does not outperform any of other algorithms.

The situation changes when $PL = medium$ (Table VII, Figure 4), which, in practical terms, means that the algorithms process more priority edges. From the data, RSC exhibits better performance in this case. It is outperformed by the PG and SC algorithms only for the $uedges(T)$ and $unodes(T)$ optimality criteria. In contrast, the RSC outperforms BF, PG

TABLE IX
RESULTS OF THE TEST SET SELECTION STRATEGY FOR THE INDIVIDUAL PROBLEM INSTANCES FOR *Edge Coverage* AND $PL = high$

| | Optimality criterion | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **ID** | $\mid T \mid$ | $edges(T)$ | $edges_h(T)$ | $uedges(T)$ | $nodes(T)$ | $unodes(T)$ | $er(T)$ | $e_h(T)$ | $e_m(T)$ | **OPT** | **SEQ** |
| 1 | RSC(4)<br>PPT(4) | PPT(18) | RSC(8)<br>PPT(8) | PPT(17) | PPT(14) | PPT(14) | PPT(0.57) | *4*(0.46) | RSC(0.90) | PPT | PPT |
| 2 | RSC(3)<br>PPT(3) | PPT(22) | PG(8) PPT(8) | PPT(22) | PPT(19) | PPT(19) | PPT(0.73) | PPT(0.36) | PPT(0.50) | PPT | PPT |
| 3 | RSC(6)<br>PPT(6) | SC*a*(67) | SC*a*(13) | *4*(35) | SC*a*(60) | *4*(34) | *4*(0.65) | RSC(0.22) | PG*a*(0.41) | SC*a* | PPT |
| 4 | PPT(2) | PPT(31) | PPT(13) | PPT(26) | PPT(29) | PPT(25) | PPT(0.56) | PPT(0.42) | PPT(0.55) | PPT | PPT |
| 5 | PPT(6) | PPT(35) | PPT(19) | PG*a*(30)<br>PPT(30) | PPT(29) | PG*a*(26)<br>PPT(26) | PG*a*(0.67)<br>PPT(0.67) | PPT(0.54) | PPT(0.66) | PPT | PPT |
| 6 | RSC(4)<br>PPT(4) | RSC(28)<br>PPT(28) | RSC(12)<br>PPT(12) | *4*(19) | RSC(24)<br>PPT(24) | *4*(17) | *4*(0.70) | RSC(0.43)<br>PPT(0.43) | BF*a*(0.41) | RSC | RSC<br>PPT |
| 7 | RSC(11)<br>PPT(11) | PPT(99) | PPT(26) | PPT(45) | PPT(88) | PG*a*(42)<br>PG*s*(42)<br>SC*s*(42) | PPT(0.70) | PPT(0.26) | PPT(0.60) | PPT | PPT |
| 8 | PPT(3) | PPT(37) | PPT(14) | PPT(21) | PPT(34) | PPT(20) | PPT(0.70) | BF*s*(0.43) | RSC(0.56) | PPT | PPT |
| 9 | PPT(2) | PPT(31) | PPT(15) | PPT(22) | PPT(29) | PPT(21) | PPT(0.58) | PPT(0.48) | PPT(0.65) | PPT | PPT |
| 10 | RSC(3)<br>PPT(3) | RSC(26)<br>PPT(26) | RSC(10)<br>PPT(10) | RSC(25)<br>PPT(25) | RSC(23)<br>PPT(23) | RSC(22)<br>PPT(22) | RSC(0.32)<br>PPT(0.32) | PG*s*(0.44)<br>SC*s*(0.44) | PG*s*(0.51)<br>SC*s*(0.51) | RSC<br>PPT | RSC<br>PPT |
| 11 | RSC(3)<br>PPT(3) | RSC(22)<br>PPT(22) | PPT(12) | RSC(19) | RSC(19)<br>PPT(19) | RSC(16) | RSC(0.28) | RSC(0.59) | RSC(0.59)<br>PPT(0.59) | RSC | RSC |
| 12 | RSC(3)<br>PPT(3) | RSC(27) | RSC(12)<br>PPT(12) | *4*(22) | RSC(24) | *4*(20) | *4*(0.32) | RSC(0.44) | PG*a*(0.63)<br>PG*s*(0.63)<br>SC*s*(0.63) | RSC | RSC |

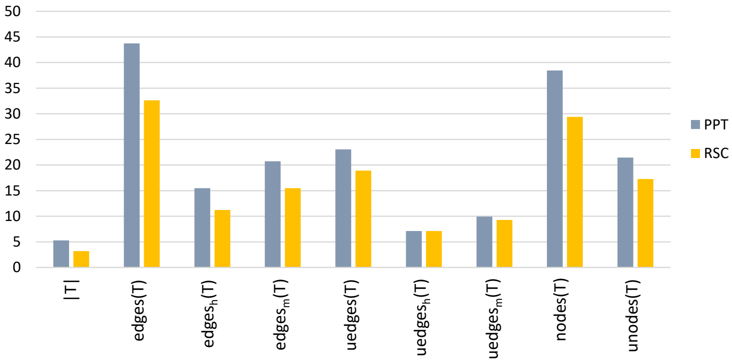


Fig. 6. Algorithm comparison for *Edge-Pair Coverage* and $PL = high$

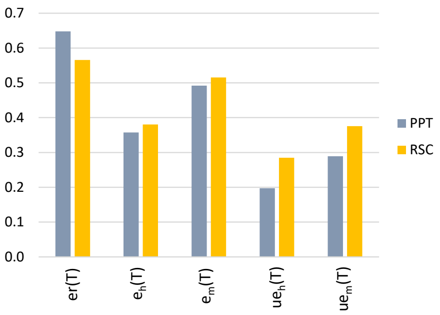


Fig. 7. Algorithm comparison for *Edge-Pair Coverage* and $PL = high$

and SC for $edges(T)$ and $edges_h(T)$, which can be considered as important criteria of test set optimality.

For the total test cases $\mid T \mid$, the RSC yields the better results than do the BF, PG and SC for both $PL = high$ and $PL = medium$. However, $\mid T \mid$ itself as an indicator of test set optimality is probably insufficient; the total number of test steps (e.g. $edges(T)$ or $nodes(T)$) are more reliable metrics.

Some other conclusions can be drawn from the data for the *Edge Coverage* criterion. For $PL = high$, the differences in the results for the atomic and sequence conversion of the test requirements $R$ (refer to Table III) are more significant for the BF and SC algorithms; however, the differences are not so significant for PG for majority of the test set optimality criteria. A similar trend can be found for $PL = medium$ although for the test set optimality criteria $edges_h(T)$ and $edges_m(T)$, the differences caused by the atomic and sequence conversions of the test requirements for the BF and SC algorithms are lower, whereas this difference is higher for the PG algorithm compared to $PL = high$.

Regarding the *Edge-Pair Coverage* criterion, the situation changes: the RSC algorithm outperforms the PPT algorithm on all the test set optimality criteria for $PL = high$ (Table VIII, Figure 6) and for $PL = medium$ (Table VIII, Figure 8).

In some cases, the results are relatively similar, for instance scores for $uedges_m(T)$, $e_h(T)$ and $e_m(T)$ when $PL = high$ and $e_h(T)$ and $e_m(T)$ when $PL = medium$. However, significant differences can be observed for the rest of the test set optimality criteria. For instance, when $PL = high$, the difference in $\mid T \mid$ is 39%, the difference in $edges(T)$ is 25% and the difference in $nodes(T)$ is 24%. When $PL = medium$, the difference in $\mid T \mid$ is 39%, the difference in $edges(T)$ is $27\%$ and the difference in $nodes(T)$ is $25\%$. These results show that the RSC algorithm is a better candidate for *Edge-Pair Coverage* criterion than is PPT.

Regarding $uedges_h(T)$ when $PL = high$ for both Edge Coverage and Edge-Pair Coverage, all the employed algorithms created the test sets that had the same value for the $uedges_h(T)$ criterion. This is a correct result and occurs because of the principle behind the algorithms. The same analogy applies for $uedges_h(T)$ and $uedges_m(T)$ when $PL = medium$.

Regarding the **test set selection strategy** (the second part) other facts can be observed from the data. The most important finding is that for various problem instances $\mathcal{G}$ and different optimality criteria, different algorithms provide the optimal test set. This can be observed similarly for *Edge Coverage* with $PL = high$ (see Figure 10 and Table IX) and for *Edge Coverage* with $PL = medium$ (see Figure 11), *Edge-Pair Coverage* with $PL = high$ (see Figure 12) and *Edge-Pair Coverage* with $PL = medium$ (see Figure 13). This effect is well documented by a sample of the detailed data provided in Table IX.

For certain test set optimality criteria, the algorithms that provide the optimal solution for all or for the majority of the problem instances can be identified. For instance, this is true in the case of the *Edge Coverage* criterion ($PL = high$ and $PL = medium$), using the PPT algorithm and the test set optimality criteria $\mid T \mid$ and $edges(T)$. However, for different optimality criteria (e.g., $e_h(T)$ and $e_m(T)$), a single algorithm that clearly outperforms the other algorithms cannot be identified. For $PL = medium$, this effect is even more obvious and relates to the fact that when $PL = medium$ the algorithms reflect more priority edges. Moreover, when $PL = medium$, the data shows that no single algorithm clearly outperforms the others for the other test set optimality criteria, namely, $uedges(T)$, $unodes(T)$ and $er(T)$. For $uedges(T)$, for instance, PPT provided the optimal test set for 31 problem instances, RSC for 16 instances, BF with atomic conversion of test requirements for 11 instances, BF with

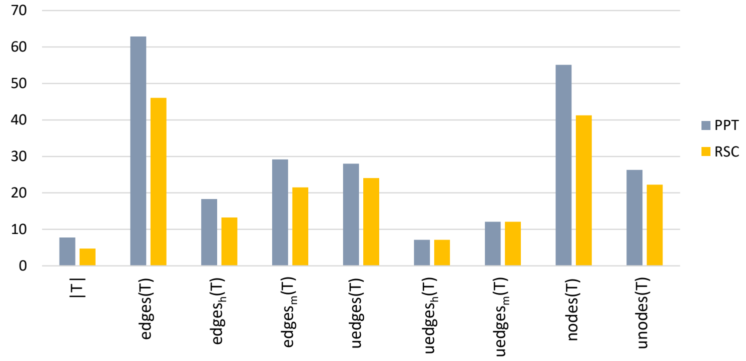


Fig. 8. Algorithm comparison for *Edge-Pair Coverage* and $PL = medium$

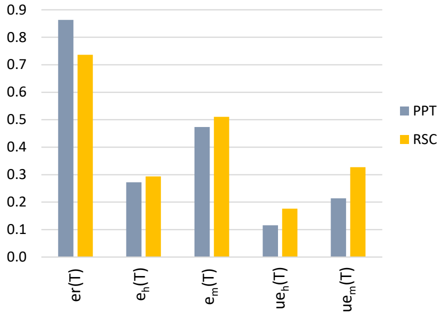


Fig. 9. Algorithm comparison for *Edge-Pair Coverage* and $PL = medium$

sequence conversion of test requirements for 22 instances, SC with atomic conversion of test requirements for 14 instances, SC with sequence conversion of test requirements for 30 instances, PG with atomic conversion of test requirements for 27 instances and PG with sequence conversion of test requirements for 33 instances. On a given problem instance, applying more algorithms can provide an optimal result.

Generally, the results of the *Edge Coverage* criteria (Figures 10 and 11) correlate with the findings presented when the algorithms were compared by their average values of optimality criteria (Figures 2 and 4).

For *Edge-Pair Coverage*, the analysis is simpler, because only two algorithms, PPT and RSC are comparable for this test coverage level. When $PL = high$,the RSC outperforms PPT on most of the test set optimality criteria; however, no clear ”winner” can be identified for criteria $e_h(T)$ and $e_m(T)$. When considering $e_h(T)$, PPT provided the optimal test set for 25 problem instances, RSC provided the optimal test set for 33 problem instances, and both algorithms provided the optimal test set for 8 problem instances. Considering $e_m(T)$ , PPT provided the optimal test set for 27 problem instances, RSC provided the optimal test set for 33 problem instances, and both algorithms provided the optimal test set for 10 problem instances.

For the optimality criteria $uedges(T)$, $nodes(T)$, $unodes(T)$ and $er(T)$, when $PL = high$, RSC outperformed PPT in 39 out of 50 problem instances , while both algorithms provided the same result for 6 problem instances.

For $PL = medium$, the RSC outperformed PPT in most of the test set optimality criteria. For this priority level, RSC yields the better results. This also applies to the $e_h(T)$ and $e_m(T)$ previously discussed for $PL = high$. Considering $e_h(T)$ when $PL = medium$, PPT provided the optimal test set for 18 problem instances, RSC provided the optimal test set for 36 problem instances, and both algorithms provided the optimal test set for 4 problem instances. Considering $e_m(T)$ , PPT provided the optimal test set for 19 problem instances, RSC provided the optimal test set for 36 problem instances, and both algorithms provided the optimal test set for 4 problem instances.

Generally, the results justify the concept proposed in this paper: in situations in which different algorithms provide optimal results for different problem instances (when considering a particular test set optimality criterion), employing more algorithms and then selecting the best set is a practical approach.

## VI. Threats to Validity

Several issues can be raised regarding the validity of the results; we discuss them in this section and describe the countermeasures that mitigate the effects of these issues.

The first concern that can be raised involves the generation of the set of test requirements $R$ from $\mathcal{G}$ for the BF, SC and PG algorithms for the *Edge Coverage* criterion ($TDL = 1$), where $PL$ is used for the test set reduction. The SUT models $\mathcal{G}$ and $G$ with $R$ differ between the methods, how to capture the priority parts of the SUT process, hence the different possibilities for conversion between the edge priorities in $\mathcal{G}$ and the set of test requirements $R$ can be discussed. To mitigate this issue, we employed and analyzed two different strategies for generating the set of test requirements $R$ from $\mathcal{G}$, namely, the atomic and sequence conversion methods, which are specified in Section III.

Another issue relates to the topology of the SUT models. The BF, SC and PG algorithms use a directed graph as the SUT model [3]; consequently, a directed graph is also used for RSC, because RSC employs SC as its main part (refer to Algorithm 2). For PPT, a directed multigraph can be used as input. To mitigate this issue and to ensure the objective

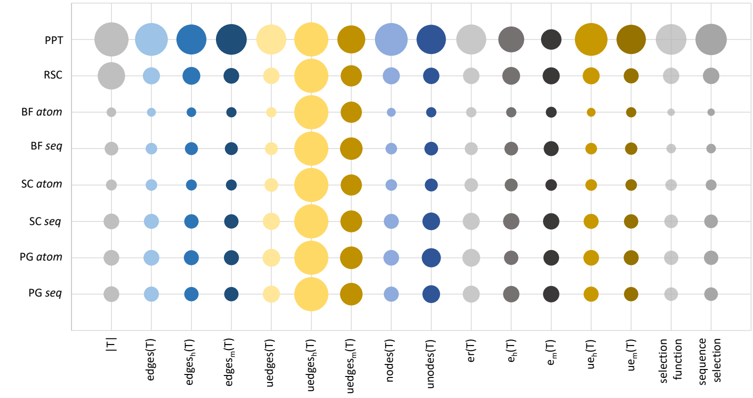


Fig. 10. Overall statistics of the test set selection strategy for *Edge Coverage* and $PL = high$

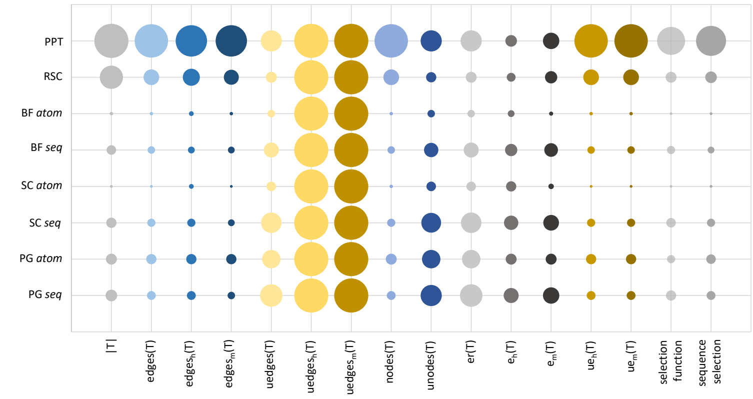


Fig. 11. Overall statistics of the test set selection strategy for *Edge Coverage* and $PL = medium$

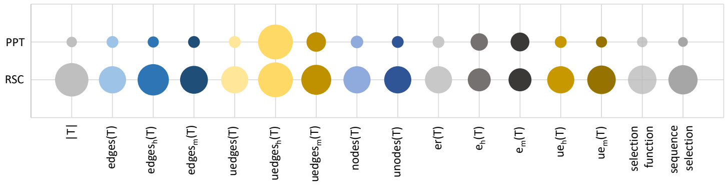


Fig. 12. Overall statistics of the test set selection strategy for *Edge-Pair Coverage* and $PL = high$

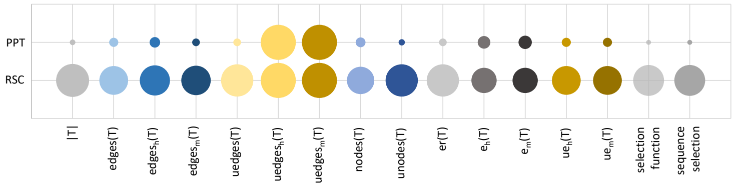


Fig. 13. Overall statistics of the test set selection strategy for *Edge-Pair Coverage* and $PL = medium$

comparability of all the algorithms and the convertibility of $\mathcal{G}$ to $G$ and $R$, we used only directed graphs in the experiments.

A related issue arises at this point: Does this restriction not limit the modeling possibilities when capturing the SUT structure? The answer is that practically speaking, the modeling possibilities are not limited. Using a directed graph leads only to more extensive models. When parallel edges present in the conceptual SUT model (e.g., UML Activity Diagram) are not allowed in its abstraction as captured by a directed graph, we instead use graph nodes to capture the parallel edges. This approach leads to more extensive graphs; however, it does not limit the algorithms and the overall solution.

Another question can be raised regarding the practical applicability of all the test set optimality criteria presented in Table I; many arguments can be brought both for and against this issue. In this study, rather than tackling such discussions, we present the data for all the optimality criteria and let the readers decide.

The last issue to be raised regards the strength of the test cases, which are reduced by the $PL$ concept to cover the priority parts of the SUT processes only. In these defined priority parts, the test coverage and the strength of the test cases are guaranteed. However, it is not guaranteed for the non-priority parts due to the principle of the $PL$ criteria. However, this fact does not invalidate the algorithms, the experimental data, or the conclusions drawn from these data.

## VII. Related Work

In the majority of the current path-based techniques, a SUT abstraction is based on a directed graph [1]. To capture the priority of specific parts of the SUT process or determine the test coverage level, test requirements are used [1], [7]. To assess the optimality of a path-based test set, a number of criteria can be discussed [3], [1], [8]. These criteria are usually based on the number of nodes, the number of edges, the number of paths or the coverage of the test requirements.

To generate path-based test cases, a number of algorithms have been proposed [2], [3], [4], [5], [6], [7], [15], such as the Brute Force algorithm, the Set-Covering Based Solution, or the Matching-Based Prefix Graph Solution [3]. Additionally, genetic algorithms have been employed to generate the prime paths [6] or to generate basis test paths [16]. Other nature-inspired algorithms have also been proposed, for example, ant colony optimization algorithms [5], [17], the firefly algorithm [18] and algorithms inspired by microbiology [4].

Test set optimization based on prioritization is considered essential area to be explored and here; various alternative approaches can be identified. As an example, clustering based on a neural network was examined in [19], fuzzy clustering possibilities were explored in [20], and the Firefly optimization algorithm was utilized in [21]. These approaches also use the internal structure of a SUT as the input to the process.

The path-based testing technique itself is generally applicable to and can be employed for various types of testing. For instance, the composition of end-to-end business scenarios [13], the composition of scenarios for integration tests or path-based testing focusing on the code level of the SUT [22], [8]. On this last level, path-based testing overlaps with the data-flow technique, which focuses on verifying the data consistency of the SUT code [23], [24], [25], [26]. In this area, control-flow graphs are employed as the SUT abstraction [22].

Alternative approaches to the current test requirement concept have been formulated [9], which result in capturing the priorities by the weights of the graph edges. This approach was inspired by the need for more priority levels, which are commonly used in the software engineering and management praxes [10], [11]. Another motivation for this approach regards certain limitations of the test requirements concept: in a number of the algorithms, the test requirements can be practically used either to specify the SUT priority parts or to determine the test intensity. As an alternative, the PPT was formulated, which is an algorithm that combines variable test coverage with SUT part prioritization [9].

Regarding using a combination of algorithms to determine the optimal test set, significantly less work exists. Some work utilizing this idea exists in the area of combinatorial interaction testing, in which different approaches are combined to obtain the optimal test set [27], [28]. Considering the experimental results presented in this paper, this stream can be considered prospectively for the path-based testing domain.

## VIII. Conclusion

In the paper, we proposed a strategy that employs a set of currently available algorithms and one new algorithm to find an optimal set of path-based test cases for a SUT model based on a directed graph with priority parts. The priority is captured as edge weights; for some of the algorithms, it is converted to test requirements. The optimality of the test set is determined by an optimality criterion selected by the user from fourteen indicators of test set optimality, by an optimality function that can be parameterized, or by the sequence selection method specified in this paper. The experimental results from running this strategy on 50 various problem instances justify the proposed approach. For the various problem instances and different optimality criteria, different algorithms provide the optimal test set—an outcome that was observed for all four combinations of test coverage and priority level criteria used in the experiments.

From the exercised algorithms, the PPT provided the best results for the *Edge Coverage* criterion. However, for certain sets of problem instances and certain test set optimality criteria, the PPT is outperformed by other algorithms (i.e., RSC, SC, and PG) and by BF in certain instances. For the *Edge-Pair Coverage* criterion, where the PPT and the RSC were the only comparable candidates for solving the problem (combining *Edge-Pair Coverage* with prioritization of particular SUT model parts), the RSC outperformed the PPT on the majority of the optimality criteria. However, for specific optimality criteria (e.g., $e_h(T)$ and $e_m(T)$), the dominance of the RSC algorithm was weak, and for a significant proportion of the problem instances, the PPT provided better results.

The proposed test set selection strategy is not a substitute for the development of new perspective algorithms to solve the path-based test case generation problem. Using this strategy, the quality of the overall result depends on the quality of the algorithms employed. If new algorithms are developed that provide better results for particular problem instances, this strategy could provide better results in the future.

## Acknowledgments

This research is conducted as a part of the project TACR TH02010296 Quality Assurance System for the Internet of Things Technology.

## References

[1] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.

[2] A. Dwarakanath and A. Jankiti, "Minimum number of test paths for prime path and other structural coverage criteria," in *IFIP International Conference on Testing Software and Systems*. Springer, 2014, pp. 63–79.

[3] N. Li, F. Li, and J. Offutt, "Better algorithms to minimize the cost of test paths," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 2012, pp. 280–289.

[4] V. Arora, R. Bhatia, and M. Singh, "Synthesizing test scenarios in uml activity diagram using a bio-inspired approach," *Computer Languages, Systems & Structures*, 2017.

[5] F. Sayyari and S. Emadi, "Automated generation of software testing path based on ant colony," in *Technology, Communication and Knowledge (ICTCK), 2015 International Congress on*. IEEE, 2015, pp. 435–440.

[6] B. Hoseini and S. Jalili, "Automatic test path generation from sequence diagram using genetic algorithm," in *Telecommunications (IST), 2014 7th International Symposium on*. IEEE, 2014, pp. 106–111.

[7] M. Shirole and R. Kumar, "Uml behavioral model based test case generation: a survey," *ACM SIGSOFT Software Engineering Notes*, vol. 38, no. 4, pp. 1–13, 2013.

[8] N. Li, U. Praphamontripong, and J. Offutt, "An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage," in *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*. IEEE, 2009, pp. 220–229.

[9] M. Bures, T. Cerny, and M. Klima, "Prioritized process test: More efficiency in testing of business processes and workflows," in *International Conference on Information Science and Applications*. Springer, 2017, pp. 585–593.

[10] P. Achimugu, A. Selamat, R. Ibrahim, and M. N. Mahrin, "A systematic literature review of software requirements prioritization research," *Information and software technology*, vol. 56, no. 6, pp. 568–585, 2014.

[11] L. van der Aalst, E. Roodenrijs, J. Vink, and R. Baarda, *TMap NEXT: business driven test management*. Uitgeverij kleine Uil, 2013.

[12] T. Koomen, B. Broekman, L. van der Aalst, and M. Vroon, *TMap next: for result-driven testing*. Uitgeverij kleine Uil, 2013.

[13] M. Bures, "Pctgen: automated generation of test cases for application workflows," in *New Contributions in Information Systems and Technologies*. Springer, 2015, pp. 789–794.

[14] P. Ammann and J. Offutt. (2017) Graph coverage web application, http://cs.gmu.edu:8080/offutt/coverage/graphcoverage. [Online]. Available: http://cs.gmu.edu:8080/offutt/coverage/GraphCoverage

[15] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. Mcminn *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[16] A. S. Ghiduk, "Automatic generation of basis test paths using variable length genetic algorithm," *Information Processing Letters*, vol. 114, no. 6, pp. 304–316, 2014.

[17] P. R. Srivastava, N. Jose, S. Barade, and D. Ghosh, "Optimized test sequence generation from usage models using ant colony optimization," *International Journal of Software Engineering & Applications*, vol. 2, no. 2, pp. 14–28, 2010.

[18] P. R. Srivatsava, B. Mallikarjun, and X.-S. Yang, "Optimal test sequence generation using firefly algorithm," *Swarm and Evolutionary Computation*, vol. 8, pp. 44–53, 2013.

[19] N. Gökçe, M. Eminov, and F. Belli, "Coverage-based, prioritized testing using neural network clustering," in *International Symposium on Computer and Information Sciences*. Springer, 2006, pp. 1060–1071.

[20] F. Belli, M. Eminov, and N. Gökçe, "Coverage-oriented, prioritized testing–a fuzzy clustering approach and case study," in *Latin-American Symposium on Dependable Computing*. Springer, 2007, pp. 95–110.

[21] V. Panthi and D. Mohapatra, "Generating prioritized test sequences using firefly optimization technique," in *Computational Intelligence in Data Mining-Volume 2*. Springer, 2015, pp. 627–635.

[22] J. Yan and J. Zhang, "An efficient method to generate feasible paths for basis path testing," *Information Processing Letters*, vol. 107, no. 3-4, pp. 87–92, 2008.

[23] M. L. Chaim and R. P. A. De Araujo, "An efficient bitwise algorithm for intra-procedural data-flow testing coverage," *Information Processing Letters*, vol. 113, no. 8, pp. 293–300, 2013.

[24] G. Denaro, M. Pezzè, and M. Vivanti, "On the right objectives of data flow testing," in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. IEEE, 2014, pp. 71–80.

[25] G. Denaro, A. Margara, M. Pezze, and M. Vivanti, "Dynamic data flow testing of object oriented systems," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 947–958.

[26] T. Su, K. Wu, W. Miao, G. Pu, J. He, Y. Chen, and Z. Su, "A survey on data-flow testing," *ACM Computing Surveys (CSUR)*, vol. 50, no. 1, p. 5, 2017.

[27] K. Z. Zamli, B. Y. Alkazemi, and G. Kendall, "A tabu search hyper-heuristic strategy for t-way test suite generation," *Applied Soft Computing*, vol. 44, pp. 57–74, 2016.

[28] K. Z. Zamli, F. Din, G. Kendall, and B. S. Ahmed, "An experimental study of hyper-heuristic selection and acceptance mechanism for combinatorial t-way test suite generation," *Information Sciences*, vol. 399, pp. 121–153, 2017.

# 8 Appendix B: Testing the consistency of business data objects using extended static testing of CRUD matrices

[A.2] Miroslav Bures, Tomas Cerny, Karel Frajtak, and Bestoun S. Ahmed. Testing the consistency of business data objects using extended static testing of CRUD matrices. *Cluster Computing*, online, pages 1-14. 2017. (Q2, IF 2.04)

Cluster Comput
DOI 10.1007/s10586-017-1118-7

# Testing the consistency of business data objects using extended static testing of CRUD matrices

Miroslav Bures[1] · Tomas Cerny[2] · Karel Frajtak[1] · Bestoun S. Ahmed[1,3]

Received: 3 May 2017 / Revised: 23 July 2017 / Accepted: 10 August 2017

**Abstract** Static testing is used to detect software defects in the earlier phases of the software development lifecycle, which makes the total costs caused by defects lower and the software development project less risky. Different types of static testing have been introduced and are used in software projects. In this paper, we focus on static testing related to data consistency in a software system. In particular, we propose extensions to contemporary static testing techniques based on CRUD matrices, employing cross-verifications between various types of CRUD matrices made by different parties at various stages of the software project. Based on performed experiments, the proposed static testing technique significantly improves the consistency of Data Cycle Test cases. Together with this trend, we observe growing potential of test cases to detect data consistency defects in the system under test, when utilizing the proposed technique.

✉ Tomas Cerny
tomas_cerny@baylor.edu

Miroslav Bures
miroslav.bures@fel.cvut.cz

Karel Frajtak
frajtak@fel.cvut.cz

Bestoun S. Ahmed
albeybes@fel.cvut.cz

[1] Department of Computer Science, FEE Czech Technical University in Prague, Karlovo Namesti 13, 121 35 Prague, Czech Republic

[2] Department of Computer Science, ECS Baylor University, One Bear Place #97141, Waco, TX 76798, USA

[3] College of Engineering, Salahaddin University-Erbil, Kurdistan, Iraq

**Keywords** Static testing · Data consistency testing · Data Cycle Test · CRUD matrix

## 1 Introduction

Static testing is an efficient method detecting software defects in a phase, where the defect fixing is rather inexpensive when compared to the later project phases. Various concepts and methods exist in this area. In this paper, we focus on static testing related to consistency of business data objects in the Enterprise Information Systems (EIS). Usually, dataflow based techniques apply to data consistency in EIS. On the conceptual level, the Data Cycle Test (DCyT) [14,18] is considered as a template for the data consistency tests. The DCyT bases on a concept of CRUD matrix and proposes fundamental methods of static testing using such CRUD matrices. In this paper, we propose extensions to the static testing methods.

The discussed static testing methods contribute to two positive effects when applied on a software development project:

1. It can detect design errors or inconsistencies related to handling of business data objects of the System Under Test (SUT) (e.g. missing functions, wrong assignment of SUT functions to the business data objects, suboptimal design of particular business data objects)
2. Later, in the test design and test execution phases of the project, proper static testing can lead to the design of more consistent and effective DCyT dynamic test cases (see Sect. 1.2). Since the test basis (design documentation, from which the test cases are derived) is rarely free of design errors and rarely up-to-date in all phases of the software project, defects in the test basis lead promote into the DCyT test cases. For Example, the test case can-

Published online: 24 August 2017

not be executed in the SUT, is missing an important part of functionality to test.

In this paper, we use the following concepts. A ***data entity*** defines a data object consisting of data that are stored in the database of the SUT. For test design purposes, data entities are commonly identified on the conceptual level of the design phase. Typically, we are interested in capturing principal business data entities that correspond to a reality modeled by the SUT. During the SUT run, several ***data objects*** are initiated as instances of a particular data entity.

A ***function*** is an SUT feature, performing any of Create, Update, Read and Delete (C, R, U, D) ***operations*** on a data entity. Further on, $F = \{f_1, \ldots, f_n\}$ is a set of all the SUT functions, and $E = \{e_1, \ldots, e_p\}$ is a set of all the data entities taken into account for the test design.

These notions allow us to define CRUD matrix as $\mathbf{M} = (m_{i,j})_{n,p}$ , $n = |F|$, $p = |E|$, $m_{\{i,j\}} = \{o|o \in \{C, R, U, D\}$ $\Longleftrightarrow$ function $f_i \in F$ performs the respective Create, Read, Update or Delete operations on the data entity $e_j \in E\}$.

The DCyT aims to detect data consistency defects. We consider a ***data entity being*** inconsistent as an SUT defect, when the following scenario occurs:

A data object $o$ is an instance of data entity e. A function $f_1$ changes the values of data object o attributes. Before being processed by the function $f_1$, the data object $o$ with defined attributes $A = \{a, \ldots, a_n\}$ has values of these attributes $V = \{v_1, \ldots, v_n\}$. According to the SUT specification, after being processed by the function $f_1$ , the data object should have values of these attributes set to $V' = \left\{v_1', \ldots, v_n'\right\}$. Due to possible defects in function $f_1$, the attribute values can be set to $V'' = \left\{v_1'', \ldots, v_n''\right\}$. When $V' \neq V''$, we consider the data object o inconsistent. This state can cause a defect in SUT later on, when the inconsistent data object o is handled by another SUT function $f_2$ (or a set of functions). We generalize this situation that data entity e is made inconsistent.

Even the create operation can make data entity inconsistent. Consider the following situation:

According to the SUT specification, after being created by the function $f_1$, the data object $o$ (being an instance of data entity $e$) should have values of these attributes set to $V' = \left\{v_1', \ldots, v_n'\right\}$ . Due to possible defects in function $f_1$, the attribute values can be set to $V'' = \left\{v_1'', \ldots, v_n''\right\}$. When $V' \neq V''$, the data object is created inconsistent, which can cause a defective behavior, when being exercised by next SUT functions.

This paper is organized as follows. In the rest of this section, we discuss the motivation for DCyT and we summarize this technique. In Sect. 2 we present proposed approach for more extensive static testing using various types of CRUD matrices. Section 3 presents method and results of the experiments. Section 4 discusses results of the experiments and analyzes trends documented by the presented data. In Sect. 5 we discuss possible threats to validity. Section 6 presents the related work. In the last section, we conclude the paper.

### 1.1 Motivation for specialized data consistency tests

One may raise a question, whether are the data consistency defects so frequent and difficult to detect by a workflow-based testing techniques (for example [6,12,21]), that they deserve a specialized test design technique? Let us give a practical example of data consistency defect, which can be detected by DCyT. Consider an *eShop* application as the SUT, and its two separate modules: a *Client* module and an *Accounting* module. In the *Client* module, customer creates and further modifies an *Order* (data entity $e$). The *Order* consists of a set of attributes $A$ and *Order Items* $D$, $e = (A, D)$. *Order* can be active or archived. This is represented by an attribute $a \in A$, which can be set to values 'active' or 'archive'.

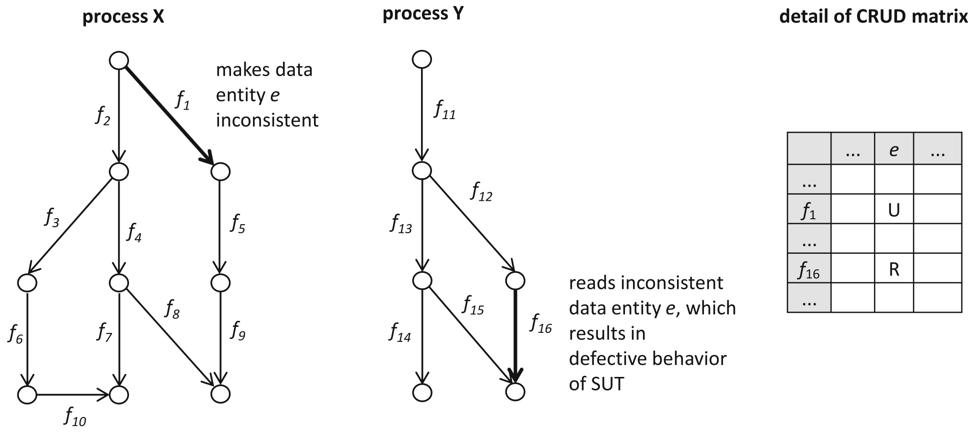
The customer creates an *Order* (during this, value of $a$ is set as 'active'). Then he adds *Order Items* to the *Order* or remove them, fills in shipping details, submits the *Order* and makes a payment. The customer can also aggregate the *Orders* together to save the shipping costs (all this activity is handled by process X). In this aggregation, main *Order* is selected and *Order Items* from the other *Orders* are copied to it. Then, the main *Order* is kept active (value of $a$ remains still as 'active') and the other Orders are kept in the database and archived by setting a as 'archived'. The main *Order* is then paid and shipped to the customer. In the *Accounting* module, monthly report is generated (process Y) and this action consists of several steps. For preparation of this report, data of customers *Orders* are used.

The *Client* module contains a defect: during the aggregation of the *Orders*, archive process is not completed properly and the value of a is not set as 'archived'. Thus, the respective *Order* data objects defined by entity $e$ became inconsistent. Their inconsistency does not cause a defect in the *Customer* module and purchase of the main *Order* (thus, cannot be detected by path-based technique for process X).

Nevertheless, inconsistency of archived *Order* data objects causes a defect in the *Accounting* module (generated by process Y). As value of attribute $a$ is not properly set as 'archived', the monthly report counts also archived *Orders* to the actual numbers.

In practical test design, it could be inefficient or even impossible to detect such types of defects by a pure workflow-based testing technique, for instance [6,12,21]. Recalling the processes X and Y from the previous *eShop* example,

1. If the tester can reach the process X from the process Y in the SUT and vice versa (when we model process X and process Y by directed graphs, these graphs are



|  | ... | *e* | ... |
|---|---|---|---|
| ... |  |  |  |
| $f_1$ |  | U |  |
| ... |  |  |  |
| $f_{16}$ |  | R |  |
| ... |  |  |  |

**Fig. 1** A situation where a data consistency defect is impossible to be detected using a workflow-based technique only

connected together by common edges), it could be quite labor intensive for a practical testing process to explore all possible paths in the graph to detect the discussed data consistency defects.

2. If the tester cannot reach the process X from the process Y in the SUT and vice versa (when we model process X and process Y by directed graphs, these graphs are not connected together by common edges), it is impossible to design test cases detecting discussed data consistency defects by a pure workflow-based testing technique. This situation is depicted in Fig. 1. Practically, in some point, the tester needs to switch from one workflow to another, which we can either model using a directed graph (which would be exhaustive and very difficult task), or we can use a specialized data consistency technique.

In the both cases, a specialized test design technique, for instance DCyT [14,18] is needed. Next, we explain its principle in the following subsection.

### 1.2 Summary of DCyT and its static testing suggestions

The DCyT presented for instance by TMap Next [18] uses a CRUD matrix $\mathcal{M}$, containing Create, Read, Update and Delete operations for data entities and functions of the SUT.

In DCyT, test case is created for particular data entity to verify its consistency during the C, R, U, D operations, which may be performed on this entity. Using the defined concepts, the ***test case*** $c$ for data entity $e$ is a sequence of ***test steps*** $\{s_1, \ldots, s_n\}$, where each of these steps is a pair $(f_s, o_s)$, and $f_s \in F$, $o_s \in \{C, R, U, D\}$ is an operation that is performed on the data entity $e$ by the function $f_s$. Possible functions $f_s \in F$ in the test case $c$ are selected by the CRUD matrix $\mathcal{M}$.

Test case $c$ starts with a Create operation, followed by all possible Update operations and ends with a Delete operation. After every Create, Update and Delete operation, a Read operation is performed. DCyT discusses practically two levels of test coverage:

1. Selected Read operation for entity e is performed once after each Create, Update or Delete (nevertheless it is not defined which particular R operation should be selected), and
2. All Read operations for entity e are carried out once after each Create, Update or Delete operations. The second method produces test cases with more steps, having a larger potential to detect data consistency defects.

Regarding the test coverage, TMap [18] adds: All data operations with data entities of the CRUD matrix performed by individual functions should be covered by created test cases.

Such very brief coverage criteria are too simple for practical use following this principle, we would create either very lightweight, or very intensive test cases. To overcome this shortage, we introduce ***intensity level*** of test case $c$, denoted as $int(c)$.

$reads(e)$ = number of R operations in the respective column of the CRUD matrix $\mathcal{M}$ for entity $e$.

$int(c)$ = number of R operations following each of the C, U or D operations exercised on entity $e$ by the test case $c$.

If $reads(e) < int(c)$ then each of the C, U or D operations exercised on entity $e$ are followed by $reads(e)$ R operations.

Regarding the static testing possibilities, TMap version of the DCyT technique describes these options only in high-level, focusing on verification of the completeness of the C, R, U, D operations for each entity $e \in E$. When entire lifecycle is not designed for and entity $e$, it is suggested as situation for further investigation (this situation does not mean a design defect automatically, nevertheless particular case has to be analyzed to prevent these possible design defects).

### 1.3 Motivation for extension of the DCyT static testing

The DCyT static testing shall lead to more consistent design documentation and test basis, having generally two goals: (a) to lower defect ratio in the implemented system under test

**Table 1** Types of CRUD matrices depending on the method of preparation

| Type | Description | Who typically prepares | Project phase |
|---|---|---|---|
| 1 | The CRUD matrix is constructed directly from the SUT code by a manual analysis or a semi-automated way | Developer or technical analyst | Coding in progress/finished |
| 2 | Test analyst uses a CRUD matrix, which has been created by other party during the system design on the project | System designer or data architect | Design specification |
| 3 | A CRUD matrix is assembled by test designer from the business or technical specification of SUT behavior. In this specification, data entities and SUT functions using these data entities were identified. Respective C, R, U, D operations performed on the data entities by these functions were then added to the matrix | Test designer | Test analysis |
| 4 | Test designer designs a CRUD matrix in a way different to Type 2. The test designer summarizes only a list of data entities and SUT functions. Then, the designer independently proposes corresponding C, R, U, D operations by his/her domain knowledge. In this process, the designer uses the basic facts from the test basis only—he/she tries to create the CRUD matrix in a most independent way, separately from the detail of the test basis. To get more information about this process, we can consult with the potential business users | Test designer | Test analysis |

and (b) to design of more consistent and effective test cases for detection of the data consistency defects in the SUT.

To utilize the potential of static testing based on CRUD matrices fully, we propose extension to these static testing suggestions. Using a CRUD matrix, number of another rules for static testing can be defined. Moreover, cross-verification of the CRUD matrices created by different par-tied during the software development project seems as a method worth exploring.

Regarding the goals of the DCyT static testing presented above, the extension we propose ad-dress both of them. After explanation of these extensions, we further focus specifically on the second goal consistency and effectiveness of the produced DCyT test cases enhanced by per-formed static testing, as this topic has been rarely discussed in the previous literature.

## 2 Static testing using CRUD matrices

The typical presentation of DCyT [14,18] does not discuss the methods of CRUD matrix preparation. The DCyT implicitly works with CRUD matrix created by analyst or architect designing the SUT, or CRUD matrix created by test analyst from available test basis of other type. Nevertheless, there are other possible methods, how to build a CRUD matrix on a software development project. Having more versions of CRUD matrices gives us more possibilities of static testing.

Generally, there are four ways that a CRUD matrix can be created in a software development process. The types are introduced in Table 1.

Type 1 corresponds directly to an SUT implementation. Type 2 provides the information, which will be closest to a particular SUT implementation. Type 4 provides the most independence from a test designers point-of-view, and it can differ from an SUT implementation. Type 3 is in the middle of this scale.

From our observations of industrial projects, if any type is created, the CRUD matrix Type 2 is the most common type. Using the taxonomy presented in Table 1, the common definition of DCyT, which is presented for example by [14,18] implicitly works with Type 1 and Type 2 only. By introducing Types 1, 3 and 4, we provide new opportunities for static testing: a cross-verification of CRUD matrices.

### 2.1 Cross-verification of the CRUD matrices

To extend the opportunities for static testing based on CRUD Matrices, we propose the following cross-verification method:

1. Prepare the test basis: two or more independently prepared types of the CRUD matrices $\mathcal{M}_1, \ldots \mathcal{M}_n$, $n = 2 \ldots 4$ for the SUT.
2. For the two selected CRUD matrices $\mathcal{M}_1$ and $\mathcal{M}_2$, $E_1$ is a set of entities in the matrix $\mathcal{M}_1$; $E_2$ is a set of entities

in the matrix $\mathcal{M}_2$; $F_1$ is a set of functions in the matrix $\mathcal{M}_1$, and $F_2$ is a set of functions in the matrix $\mathcal{M}_2$.

3. Organize the matrices $\mathcal{M}_1$ and $\mathcal{M}_2$ to list the functions $F_1$, $F_2$ and entities $E_1$, $E_2$ in the same order by using the same criteria (e.g., alphabetical sort).
4. If $E_1 \neq E_2$:
   (a) Analyze to determine whether some of the entities from E1 and E2 appearing as different entities are actually the same entity. If yes, unify identification of the entities;
   (b) If still $E_1 \neq E_2$, report the difference $E_1 - E_2$ to the backlog of issues that must be clarified.
5. If $F_1 \neq F_2$:
   (a) Analyze and determine if some of the functions from $F_1$ and $F_2$, which appear as different functions, are actually the same functions. If yes, unify identification of the functions;
   (b) If still $F_1 \neq F_2$, report the difference $F_1 - F_2$ to the backlog of issues that must be clarified.
6. For each of the cells of the compared matrices that correspond to $e$ and $f$, $e \in E_1$, $e \in E_2$, $f \in F_1$, $f \in F_2$, if the cell content differs in the C, R, U, D operations, report the difference to the backlog of issues that must be clarified.
7. When the issues in the backlog are clarified, merge matrices $\mathcal{M}_1$ and $\mathcal{M}_2$ to a final CRUD matrix $\mathcal{M}$, which will represent a corrected version of the expected behavior for the SUT.

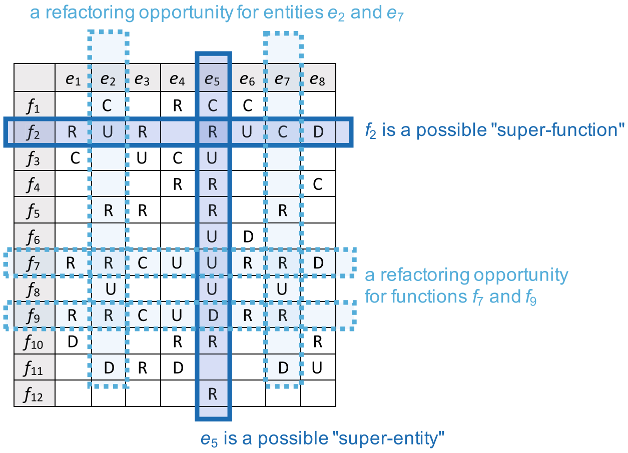
A difference reported to the backlog can denote either incomplete information in one of the CRUD matrices $\mathcal{M}_1$ and $\mathcal{M}_2$ or a potential defect, which is the subject of our investigation.

The next set of static tests can be defined for one CRUD matrix.

### 2.2 Extended static tests using a single CRUD matrix

For static testing performed on a single CRUD matrix, as described in TMap Next [18], we propose the following extension. For each of the cells in the matrix $\mathcal{M}$, $E$ is a set of entities in the matrix $\mathcal{M}$, and $F$ is a set of functions in the matrix $\mathcal{M}$.

1. If entities $e_1 \in E$ and $e_2 \in E$ have the same set of C, R, U, D operations in their respective columns in the CRUD Matrix: analyze the situation to determine if the entities are separated for a certain reason or if this situation indicates unnecessary duplicity in the code (an example is given in Fig. 2);



a refactoring opportunity for entities $e_2$ and $e_7$

| | $e_1$ | $e_2$ | $e_3$ | $e_4$ | $e_5$ | $e_6$ | $e_7$ | $e_8$ |
|---|---|---|---|---|---|---|---|---|
| $f_1$ | | C | | R | C | C | | |
| $f_2$ | R | U | R | | R | U | C | D |
| $f_3$ | C | | U | C | U | | | |
| $f_4$ | | | | R | R | | | C |
| $f_5$ | | R | R | | R | | R | |
| $f_6$ | | | | | U | D | | |
| $f_7$ | R | R | C | U | U | R | R | D |
| $f_8$ | | U | | | U | | U | |
| $f_9$ | R | R | C | U | D | R | R | |
| $f_{10}$ | D | | | R | R | | | R |
| $f_{11}$ | | D | R | D | | | D | U |
| $f_{12}$ | | | | | R | | | |

$f_2$ is a possible "super-function"

a refactoring opportunity for functions $f_7$ and $f_9$

$e_5$ is a possible "super-entity"

**Fig. 2** Examples of refactoring opportunities identified in a CRUD matrix

2. When entities $e_1 \in E$ and $e_2 \in E$ have the similar subset of C, R, U, D operations in their respective columns in the CRUD matrix (let use threshold 90% of these operations), it could indicate a design optimization or refactoring opportunity (an example is given in Fig. 2);
3. If functions $f_1 \in F$ and $f_2 \in F$ have the same set of C, R, U, D operations in their respective lines in the CRUD matrix: analyze to determine whether the functions are separated for a certain reason or if this situation indicates an unnecessary duplicity in the code;
4. When functions $f_1 \in F$ and $f_2 \in F$ have the similar subset of C, R, U, D operations in their respective columns in the CRUD matrix (let use threshold 90% of these operations), it could indicate a design optimization or refactoring opportunity;
5. For each entity $e \in E$, verify how the deletion operation is specified in the source documentation and how it is reflected in the CRUD matrix:
   (a) Entity $e$ has to be deleted, which should be captured by a D operation;
   (b) Entity $e$ has to be archived, instead of deleted, which should be captured by a U operation for the entity $e$, or by a D operation for the entity $e$ and a C operation for an archive entity $e'$, which is copied from $e$.
6. If there are requirements to maintain a history of changes for data entity $e \in E$ after an update of this entity by function $f \in F$, this fact may be explored in detail. In addition to the respective U operations in the CRUD matrix in the cells corresponding to $e$ and $f$, the situation may be captured by other U or C operations in the matrix line corresponding to the function $f$. These additional U or C operations would be performed on the entity that maintains a record of the changes in entity $e$. The particular situation depends on the technical details of the implementation process.

7. If a function $f$ contains name of an entity $e$ in its name or description, but no C, R, U, D operation is indicated in the matrix cell corresponding to $e$ and $f$, investigate the situation for a possible design defect. Minimally, the terminology used in the design shall be made more unambiguous.
8. If more functions $F_C \subset F$ perform C operations for entity $e$, this situation shall be investigated as it could indicate a design optimization or refactoring opportunity - nevertheless, not necessarily a design error.
9. If a set of functions $F_0 \subset F$ perform a C, R, U, D operation on the same entity $e$ and $|F_0| > 0.5\,|F|$, investigate the situation for a design optimization or refactoring opportunity. Keeping a lifecycle of such "super-entity" $e$ consistent could be demanding from the viewpoint of possible data consistency defects (an example is given in Fig. 2).
10. By analogy, if a single function $f$ performs a C, R, U, D operation on a set of data entities $E_0 \subseteq E$ and $|E_0| > 0.5\,|E|$, investigate the situation for a design optimization or refactoring opportunity. Such "super-functions" can be prone to defects, as they are usually complex (an example is given in Fig. 2).

The situations described in steps 1–4 and 7 are interesting from the general redundancy point-of-view in an SUT, which is a frequent source of defects. In this analysis, possible planned future extensions of the system should be considered.

Step 5 aims to detect another possible type of defect that is related to the proper deletion or archival of the data entities. From our experience, this is an area where design mistakes can occur; these are expensive to correct after the implementation phase. The same analysis applies to step 6, which focuses on the requirements for maintaining a history of the changes in the data entities that are processed by an SUT.

Step 7 can detect design errors in the CRUD matrix or ambiguous terminology used in the design documentation. Steps 9 and 10 aims to identify too complex functions and entities which could be potentially prone to defects during the development and regression in the later stages of the project.

## 3 Experiments

To measure the effectiveness of the proposed static testing approach concerning production of more consistent and effective DCyT test cases, we simulated a situation in which an incomplete and inconsistent test basis was used as an input to the creation of DCyT test cases. In this experiment, we gave a group of test designers a design specification of existing opens-source system (referred as experimental SUT further on). Before doing that, we changed this specification in few places to simulate design defects and inconsistencies, which can naturally occur in a real software development project. Without accessing this system and using the design specification only, the group was creating DCyT test cases: the first part of the group has not used any static testing; the second part of the group was using the static testing extension proposed in this paper. Finally, we evaluated the produced DCyT test cases using a set of artificial defects inserted in the experimental SUT to answer the following research questions:

> *Q1.1* How consistent are these DCyT test cases when using the proposed static testing and when not? By consistency of the test case, we mean real ability to execute the test case in the SUT (defined in Table 2 further on). Moreover,
>
> *Q1.2* How is this consistency influenced by intensity level (defined in Sect. 1.2) of these test cases?
>
> *Q2.1* How many potential data consistency defects can be detected by these DCyT test cases when using the proposed static testing and when not?
>
> *Q2.2* How is this defect-detection potential influenced by intensity level (defined in Sect. 1.2) of these test cases?

Details of the experiment follow in this section.

### 3.1 Experimental setup

As the experimental SUT, we used open-source MantisBT[1] issue-tracker application written in PHP. We used its version 1.2.19. We re-engineered the design documentation of this system, consisting of two artifacts: the workflow model and the CRUD matrix corresponding to this model.

The workflow model of the system was covering its principal parts, consisting of system screens S, modeling also the high-level states of the system and business functions $F$ as transitions between them, $|S| = 46$ and $|F| = 122$. We considered this model to describe the correct version of the experimental SUT, and we refer to it as a *Baseline workflow model* further on.

The CRUD matrix was created for principal conceptual business data entities $E$, $|E| = 12$. These entities were: user, project, project hierarchy, issue report, file attached to the issue report, tag and issue tag, custom field, custom field string, filter and issue monitoring. There were 183 C, R, U or D operations in the matrix. By analogy, we considered this CRUD matrix describing the correct version of the experimental SUT, and we refer to it as a *Baseline CRUD matrix* further on. Regarding the types of CRUD matrices presented in Table 1, this matrix is Type 2.

[1] https://www.mantisbt.org/index.php

**Table 2** Metrics for evaluation of DcyT test cases produced by experiment participants

| Metric | Description | Research question |
|---|---|---|
| $Steps(T_p)$ | Average number of test steps of particular test cases of $T_p$<br>$steps(T_p) = \sum \|c\| / \|T_p\|, c \in T_p$ | Q1.1, Q1.2 |
| $inc(c)$ | Number of inconsistent test steps of the test case $c$<br>The test step $s \in c \in T_p$ is inconsistent when two subsequent C, R, U, D operations performed by functions $f_1 \in s$ and $f_2 \in s$ in the test case $c$ cannot be performed in the SUT. This occurs because we cannot reach a proper state in the SUT to execute the function $f_2$ from the SUT state reached by function $f_1$ | Q1.1, Q1.2 |
| $inc(T_p)$ | Average number of inconsistent test steps of all $c \in T_p$<br>$inc(T_p) = \sum inc(c)/ \|T_p\|, c \in T_p$ | Q1.1, Q1.2 |
| $inc\_rate(c)$ | $inc\_rate(c) = inc(c)/c$ | Q1.1, Q1.2 |
| $inc\_rate(T_p)$ | $inc\_rate(Tp) = inc(T_p)/steps(T_p)$ | Q1.1, Q1.2 |
| $\|D_c\|$ | Number of inserted defects, which can be potentially detected by a test case $c \in T_p$. $D_c \subseteq D$.<br>A defect $d \in D$ is considered potentially detected by test case $c$, if function $f_c$ making the data entity e inconsistent and function $f_d \in F_d$ working with the data entity $e$ later on, causing the defective behavior of the SUT are called in sequence in the test case $c$. Using the defined concepts, $d_D$ is considered potentially detected by $c$ if $f_c \in sc$, $f_d \in s_d$, where $f \in c_d$, $f_d \in F_d \in d$, $s_c \in c$, $s_d \in c$ and $s_c$ is preceding $s_d$ in $c$ | Q2.1, Q2.2 |
| $eff(T_p)$ | Average number of inserted defects, which can be potentially detected by one test case,<br>$eff(T_p) = \sum \|D_c\| / \|T_p\|, c \in T_p$ | Q2.1, Q2.2 |

Then, we inserted artificial data consistency defects $D$ to the experimental SUT. D is a set of inserted data consistency defects. An inserted data consistency defect is a tuple $d = (e, f_c, o_c, F_d)$; where $e \in E$ is a data entity, which is in an inconsistent state that causes a defect, $f_c \in F$ is the function that causes the data entity $e$ to be inconsistent (for definition refer to the Sect. 1), $o_c \in \{C, U\}$ is the particular create or update operation that causes the data entity $e$ to be inconsistent when accessed by the function $f_c$, $F_d$ is a set of pairs $(f_d, o_d)$, where $f_d \in F$ is a function that exhibits a defective behavior in the SUT as a result of the inconsistency of the data entity $e$ and $o_d \in \{C, R, U, D\}$ is the particular operation performed by function $f_d$ on data entity $e$ preceding this defective behavior. In our experiment, $|D| = 22$.

To simulate a real software project situation, when the available test basis (design documentation) is not complete, consistent or up-to-date, we entered several artificial design defects to the workflow model. In the Baseline workflow model, we removed or changed ten functions from F and 1 state from $S$, creating the *Inconsistent workflow model*.

Similarly, we created an *Inconsistent CRUD matrix* from the baseline one: In the *Baseline CRUD matrix*, we removed 7 R, 3 U, 1 D and 2 C operations from random positions. Moreover, we added 7 R and 1 U operations to positions, where such a defect was probably to be not obvious at first glance to the test analyst.

The experimental group composed of test analysts and students of the software testing course, who got intensive training to the DCyT testing technique. There were 61 participants in total joining the experiment.

### 3.2 Experiment method

In the experiment, we divided the participants into two disjunctive groups. The Group A was given the inconsistent test basis and was instructed to create DCyT test cases. The Group B was also given the inconsistent test basis, but before the design of the test cases, they were instructed to do the static testing. Then we compared test cases produced by the both groups from several aspects. The process in detail was:

#### *3.2.1 Phase 1: test cases creation*

Group A: design of DCyT test cases with inconsistent test basis as an input without any static testing

1. To set the same level of knowledge, instructions how to create DCyT were given to all the participants. Then, each of the participants:
2. received *Baseline workflow model*
3. received *Baseline CRUD matrix* (Type 1 matrix, see Table 1)
4. was instructed to create DCyT test cases for data entities E, using the TMap version of this technique [18]. Group A was divided to three sub-groups: Group A1 was creating

**Table 3** Experiment results

| Group | Static testing used | $int(c)\ \forall c \in T_p$ | $steps(T_p)$ | $inc(T_p)$ | $inc\_rate(T_p)$ (%) | $eff(T_p)$ |
|---|---|---|---|---|---|---|
| A1 | No | 1 | 9.67 | 3.26 | 33.7 | 0.51 |
| A2 | No | 2 | 15.36 | 5.76 | 37.5 | 0.54 |
| A3 | No | 3 | 18.52 | 6.67 | 36.0 | 0.67 |
| B1 | Yes | 1 | 9.44 | 2.00 | 21.2 | 0.56 |
| B2 | Yes | 2 | 15.88 | 2.66 | 16.7 | 0.67 |
| B3 | Yes | 3 | 17.70 | 3.90 | 22.0 | 0.74 |

test cases with intensity level 1, Group A2 with intensity level 2 and Group A3 with intensity level 3.

Group B: design of DCyT test cases with inconsistent test basis as an input with proposed static testing techniques

1. To set the same level of knowledge, instructions how to create DCyT were given to all the participants.
2. The proposed static testing technique was explained to all the participants. Then, each of the participants:
3. received *Baseline workflow model*
4. received *Baseline CRUD matrix* (Type 1 matrix, see Table 1)
5. was instructed to perform static testing before creation of the DCyT test cases, including cross-verification between CRUD Matrices and extended consistency verification of a CRUD matrix proposed in this paper. For this cross-verification, they were implicitly creating a matrix Type 3. When a question to clarify the inconsistencies in the test basis was raised by the participant as result of performed static testing process, it was answered by the experiment organizers. For this, Baseline CRUD matrix and a Baseline workflow model was used. Practically, experiment organizers were simulating the business analyst, determining the right version of the specification in the discussion with the participant.
6. was instructed to create DCyT test cases for data entities E, using the TMap version of this technique [18]. Group B was divided to three sub-groups: Group B1 was creating test cases with intensity level 1, Group B2 with intensity level 2 and Group B3 with intensity level 3.

### *3.2.2 Phase 2: test cases evaluation*

When the created DCyT test cases were collected from the experiment participants, we evaluated their consistency and efficiency by their simulated run in the experimental SUT with insert-ed defects.

Experiment participants were using predefined MS Excel template to fill the produced test cases. This allowed automated processing of the data. We created semi-automated loader of the test cases to the Tapir (acronym from Test Analysis Process Information Re-engineer) framework [13] connected to MantisBT defect tracker of the same version as we used for the experiment (1.2.19). The Tapir framework reconstructs the workflow model based on the high-level states (pages of the SUT) and transitions between them (SUT functions). Thus, in the Tapir framework, we had available exact workflow model of this SUT, defined by the same elements as the *Baseline workflow model*. Before the experiments, we verified both of these models to be aligned. Together with this, we added artificial data consistency defects $D$ to the SUT model in the Tapir framework.

The produced test cases were loaded to the Tapir framework and evaluated for their consistency and potential to detect inserted data consistency defects $D$. For this evaluation, we used metrics presented in Table 2. We denote a set of all test cases produced by a particular participant as $T_p$. The test case has been defined in Sect. 1.2.

Regarding the $inc(c)$, during the evaluation of the consistency of the test cases, when a step of the test case is identified as inconsistent, the rest of the test case steps are considered also as inconsistent (the test steps cannot be executed in the flow in the sequence defined by the test case).

Regarding the $|D_c|$, during the evaluation of the ability of the test cases to detect inserted defects, when a step of the test case is identified as inconsistent, the rest of the test case is ignored (to simulate a real situation when the test case cannot be executed in the SUT).

## 3.3 Results

In this section, we present the results of the experiment performed by the method described above.

During the evaluation, we analyzed 939 test cases having 15,153 steps in total. During the analysis of the test cases using the method introduced above, 4761 test steps have been evaluated as inconsistent.

Table 3 gives evaluation metrics averaged for all test cases produced by all participants in the particular group. For averaging, arithmetic mean is used.

Figure 3 depicts average ratios of the inconsistent test steps in test cases produced by the individual experiment groups.

In Table 4 we summarize differences of the indicators presented in Table 3 in comparison of the individual experimental groups. For each of three test case intensity levels, we compare a respective group using the proposed static testing methods with the group, which was not using the static testing.

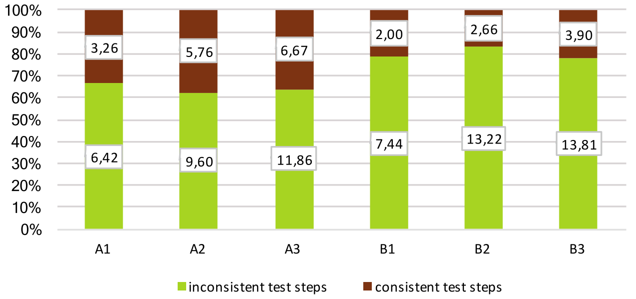


**Fig. 3** Avg. ratio of consistent and inconsistent steps in the test cases for individual groups

An average improvement when using proposed static testing for all three intensities was 15.7%. The average improvement in data consistency defect detection potential of the test cases, measured by $eff(T_p)$ for all three intensities was $-0.08$. Negative value in the column $eff(T_p)$ of Table 4 means improvement.

To have more detailed insight into inconsistency of the analyzed DCyT test cases, we present its distribution in Table 5 for the individual groups. In Table 5, $inc(x, y)$ stands for ratio of test cases, having $inc\_rate(c) \geq x$ and $inc\_rate(c) < y$. Values $x$ and $y$ are in percentage.

Figure 4 depicts the distribution presented in Table 5. In the graphs, we can see the decreasing inconsistency of the test cases produced with the help of static testing, whereas no significant trend in distribution can be observed for the test cases produced without static testing.

The data in Table 3 show, that the average potential of the DCyT test case to detect the data consistency defects measured by $eff(T_p)$ grows with the intensity level of the test cases $inc(T_p)$. The trend is depicted in Fig. 5.

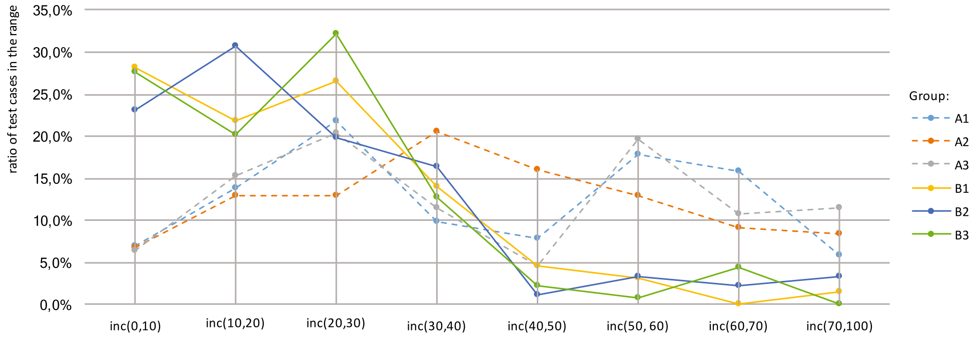
**Table 4** Differences between test cases produced by DCyT with and without proposed static testing

| Comparison groups | $int(c)\ \forall c \in T_p$ | $steps(T_p)_{A_n} - steps(T_p)_{B_n}$ | $inc(T_p)_{A_n} - inc(T_p)_{B_n}$ | $inc\_rate(T_p)_{A_n} - inc\_rate(T_p)_{B_n}$ (%) | $eff(T_p)_{A_n} - eff(T_p)_{B_n}$ |
|---|---|---|---|---|---|
| A1 and B1 | 1 | 0.24 | 1.26 | 12.5 | −0.05 |
| A2 and B2 | 2 | −0.52 | 3.10 | 20.7 | −0.13 |
| A3 and B3 | 3 | 0.82 | 2.77 | 14.0 | −0.07 |

**Table 5** Distribution of test case inconsistencies in the individual experiment groups

| Group | $inc(0, 10)$ (%) | $inc(10, 20)$ (%) | $inc(20, 30)$ (%) | $inc(30, 40)$ (%) | $inc(40, 50)$ (%) | $inc(50, 60)$ (%) | $inc(60, 70)$ (%) | $inc(70, 100)$ (%) |
|---|---|---|---|---|---|---|---|---|
| A1 | 6.9 | 13.9 | 21.8 | 9.9 | 7.9 | 17.8 | 15.8 | 5.9 |
| A2 | 6.9 | 13.0 | 13.0 | 20.6 | 16.0 | 13.0 | 9.2 | 8.4 |
| A3 | 6.5 | 15.3 | 20.3 | 11.5 | 4.5 | 19.6 | 10.8 | 11.5 |
| B1 | 28.1 | 21.9 | 26.6 | 14.1 | 4.7 | 3.1 | 0.0 | 1.6 |
| B2 | 23.1 | 30.8 | 19.8 | 16.5 | 1.1 | 3.3 | 2.2 | 3.3 |
| B3 | 27.6 | 20.1 | 32.1 | 12.7 | 2.2 | 0.7 | 4.5 | 0.0 |

**Fig. 4** Distribution of inconsistent test cases in the individual experiment groups



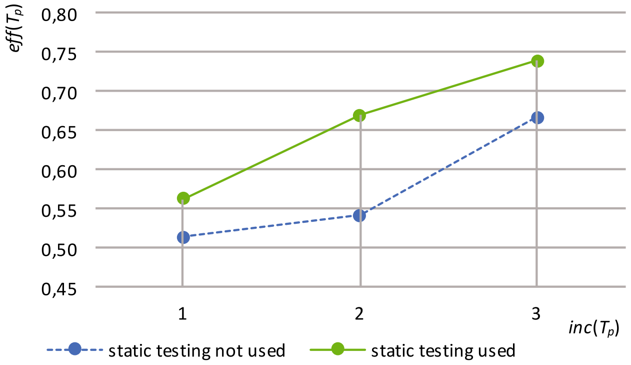


**Fig. 5** Average number of inserted defects, which can be potentially detected by one test case

We observed no significant trend when analyzing numbers of individual test cases detecting particular inserted defects. The patterns in data seemed rather random. Thus, we don't present these details in this section.

## 4 Evaluation of the results and discussion

In this section, we analyze, discuss and conclude the experiment results. Starting with average total number of test steps of the produced DCyT test cases, test cases ($steps(T_p)$ in Table 3), this number grows with the intensity level $int(c) \forall c \in T_p$. This is an expected behavior, as the principle of the defined intensity level is to determine the number of R operation test steps following the C, U, D operations in the DCyT test cases.

Further on, analyzing the average total number of steps of produced test cases ($steps(T_p)$ in Table 3), no significant change has been observed using the static testing technique. In any of three intensity levels, no significant difference can be found. When analyzing the complete aggregated data, for B groups, the differences is caused by (1) corrections of the inconsistent test basis: after its correction, the test steps of produced DCyT test cases differed in few places by principle, and (2) slight, but certain natural inaccuracy of the test cases produced by individual test designers in the same experimental group.

For all three intensity levels, the test cases produced by DCyT with proposed static testing have significantly less inconsistent steps (question Q1.1, $inc(T_p)$ and $inc\_rate(T_p)$ in Table 3, differences summarized in Table 4). These differences in $inc\_rate(T_p)$ are 12.5% for $int(c) = 1$, 20.7% for $int(c) = 2$ and 14.0% for $int(c) = 3, \forall c \in T_p$. This is an important finding, having the consequence for the practical test design process.

To answer question Q1.2, the next step was to analyze, if the intensity level has an influence on these differences. Here, no significant trend has been observed in the complete aggregated data. In absolute numbers ($inc(T_p)$ in Table 3), the number of inconsistent test case steps grow with the intensity level, nevertheless when analyzing the relative inconsistency of the test cases ($inc\_rate(T_p)$ in Table 3), we observe no significant trend. To analyze this issue deeper, experiments with more intensity levels and more SUTs shall be conducted, as also a model of particular SUT would play a role in this case.

As observed from data, the defect detection potential of the test cases created without proposed static testing (question Q2.2, $eff(T_p)$ in Table 3) grows with the intensity level, from 0.51 for $int(c) = 1$ to 0.67 for $int(c) = 3, \forall c \in T_p$. Due to the principle of intensity level, this is a logical result.

What is more important, the defect detection potential grows with performed static testing (question Q2.1, $eff(T_p)$ in Table 3, differences summarized in Table 4). These difference is 0.5 for $int(c) = 1$, 0.13 for $int(c) = 2$ and 0.7 for $int(c) = 3, \forall c \in T_p$. In this point, the trend is the same as in the consistency of the test cases.

The most significant observations are the growth of the consistency of the analyzed DCyT test cases when applying the static testing before the design of the test cases and together with this trend, the growth of potential of the test cases to detect data consistency defects in the SUT.

## 5 Threats to validity

Experiments of this type are very challenging to be performed at a real industry software development project in praxis; it is practically impossible to run the same projects twice with a different quality assurance scenarios. In addition, final software product quality is also influenced by many soft factors, which bias the results. Thus, we made our experiment to be as close to a real case as possible: we used real SUT, for which we knew its correct model (using the Tapir framework), we used an inconsistent test basis composing from workflow model and CRUD matrix and we let the experiment participants behave as when creating the test cases for a real project. For evaluation of the test cases consistency and defect detection potential, we used an automated system built on top of Tapir framework to minimize possible human mistakes. Despite this fact, our experimental setup has potential threats to validity, which we discuss in this section.

Lets start with potential possible flaws in the automated evaluation system of the DCyT test cases. First, the SUT reengineered model of the Tapir could also be inconsistent. Nevertheless, we consider this risk mitigated, as MantisBT model in the Tapir framework underwent many verifications and previous experiments. The next concern can be raised regarding the correspondence of the Baseline workflow model to the SUTmodel in the Tapir framework. We minimized the possible inconsistencies between these models by thorough check at the start of the experiment.

Next set of concerns can be raised regarding the experiment set up and soft factors during the experiment. The participants could know MantisBT before the experiment, as this issue tracker is commonly known and used by the software engineers. Nevertheless, as the MantisBT was not available to the participants during the test design process and they were explicitly instructed to not consult the created test cases with the SUT, this risk shall be minimized. Also, if some of the participants benefited from his knowledge of the SUT during the experiment, such a case was randomly distributed in the experimental groups.

Regarding the possible learning effect, we mitigated this risk by letting each of the experimental group creating only one set of test cases of particular intensity level and style (creation of the test cases with or without proposed static testing).

The effectiveness of the created DCyT test cases are varying by particular experiment participant analytic skills. Nevertheless this is a natural effect present in the test design process; the participants were randomly distributed to the individual groups, which shall also minimize the influence of this effect to the results.

Another concern can be raised regarding the artificial nature of the inserted defects. In their modeling and definition, we tried to capture the nature and principle of data objects consistency defects, as observed in a number of previous industry projects we were involved in or were observing.

## 6 Related work

The Data Cycle Test (DCyT) has been presented for example by [14,18]. A typical description of the technique consists of a guideline of how to create a CRUD matrix, several high-level comments about the possibility of static testing using a CRUD matrix and a method for creating dynamic test cases that are based on a sequence of Create, Update, Read and Delete operations that are exercised on a particular data entity.

In the TMap Next description of the Data Cycle Test [18], static testing uses a standard CRUD matrix and bases on verification of the completeness of the C, R, U, D operations for each entity $e \in E$. This approach is valid; nevertheless, it can be extended by some other techniques, introduced in this paper.

Even though data flow testing is the subject of much research interest, the goal of our study has not been directly addressed in the literature. Nevertheless, some previous results are related, and these results should be analyzed.

In the area of static testing, the work exploring the data flow analysis principle focuses on detection of design errors in workflow design [15,20,30], or in detection and automated correction measures that are used for the same problem [2]. In these proposals, the main use case is validation of the process design and notations different to CRUD matrix are used for SUT modeling. For instance, [28] uses UML activity diagrams, [20] is using UML statechart diagrams. In proposals [2,30], Petris net is used as a data flow modeling structure. In praxis, this approach is suitable for static testing, where BPMN diagrams or UML statechart diagrams are available as the test basis. Similarly, data flow analysis was also used for verification of web services models in WS-BPEL notation [22]. Here, data dependencies are identified and reflected in the verification process.

An alternative approach to static testing of a database design may be based using the Formal Concept Analysis [29]. Applications of this approach are documented either to detect the faults on the source code level [8], to detect the design flaws on the class modeling level [1], or, on the more general level to verify model in terms of used objects and their events, including dependencies between the objects [24]. Conceptually, this approach is similar to a CRUD-matrix based technique, but the proposal focuses only on verification of the SUT design. Also alternative applications in software testing exist for Formal Concept Analysis technique—for instance to generate a Feature Model [7], which can further serve as a basis for test data generation by the Constraint Interaction Testing technique.

An aided database design that is based on an extension of the CRUD matrix is proposed by [17]. The authors of that paper consider the CRUD matrix to be an insufficient source of information, and they extend the matrix to the attribute level.

In the dynamic testing area, the concept of DCyT is, in principle, similar to a Data Flow Analysis technique, presented for example by [9,11,23,26,31]. Generally, data flow analysis is an area where previous research has been conducted more intensely, and it considers an aspect of efficiency and testing coverage, for instance [25,32]. For test data generation in this technique, various alternative algorithms are explored, for instance [19]. This technique is used either on the code-flow level [16], or also on object level, for instance [10].

However, DCyT, which is the subject of our research, works on a higher level of abstraction. Data Flow Analysis is principally a white-box test design technique that operates at the program code level, and it works with particular values of the program variables. Set-use pairs are defined for three atomic actions: definition of the variable, use of the variable and destruction of the variable. From all possible set-use pairs, some are allowed, whereas some indicate suspicion of the possible defect in the code. Such relations do not hold in CRUD matrix, used by the DCyT. In addition, the level of abstraction differs: in DCyT, we are working with general data entities that are stored by the SUT, and are used by the SUT functions, which are usually exercised by functional testing. Here, we typically approach the SUT as a black-box that exists between the data layer and the testers interaction with the SUTs front-end.

Analysis of data flows can be also used to improve coverage criteria for statechart testing, as explored by [3–5]. In this approach, define and use relations are used for data objects. The aim of this approach is to increase test coverage and reduce costs of the statechart-based testing. In difference to our approach, CRUD matrices are not used and more accurate and consistent SUT model is needed.

Sun [27] proposes an alternative approach to data flow verification in SUT processes. In this approach, Data-Flow Matrices are used. The Data-Flow Matrix, similarly to CRUD matrix contains read and wire operations performed by workflow actions on particular data objects. In difference to the CRUD matrix, the proposed Data-Flow Matrix uses read and write operations only. Then, data flow information is integrated with the workflow model and data flow anomalies (principally missing data, redundant data or conflicting data) are detected. In our approach, we work with conventional CRUD Matrix on a conceptual level, as a test basis commonly available (or relatively easy to be created by the analysts and test designers) in the software development projects.

## 7 Conclusion

In this paper, we propose an extension to the conventional approach of static testing based on CRUD Matrices [14,18]. This static testing has two general goals: (a) to lower defect ratio in the implemented SUT and (b) to lead the design of more consistent and effective test cases for detection of the data consistency defects in the SUT.

The proposed extension addresses these both goals. In particular, we propose cross-verification between various types of CRUD Matrices created by different parties in various stages of the project and extension of static tests using the single CRUD matrix.

After the presentation of these extensions, we focused specifically on the second goal consistency, and effectiveness of the produced DCyT test cases enhanced by performed static testing, as this topic has rarely been discussed in the previous literature.

To investigate the effectiveness of the static testing in this context, we conducted an experiment simulating a situation in which an incomplete and inconsistent test basis was used as an input to the creation of DCyT test cases. Real issue tracking system MantisBT was used as experimental SUT. We let several groups of test designers create DCyT test cases, part of them were using the proposed static testing, part of them not. Next, we evaluated the produced DCyT test cases using and automated method. In this process, we utilized the Tapir framework [13] connected to MantisBT issue tracker. This gave us an exact workflow model of this SUT, defined by the same elements as the test basis model given to the experiment participants. Using this support, we evaluated DCyT test cases created by the experimental groups. Together with this process, we added artificial data consistency defects to the SUT model in the Tapir framework to evaluate the potential of the DCyT test cases to detect these defects.

When we compared the DCyT test cases of various intensities produced by the experimental groups using the proposed static testing with the groups not using this method, no significant trend has been identified in the average total number of test steps of the produced DCyT test cases. Also, no significant difference has been observed when analyzing the relation of test cases intensity to their consistency.

Nevertheless, for all three intensity levels, the test cases produced by DCyT with proposed static testing have significantly less inconsistent steps (15.7% in average for all three test case intensities in range 1 to 3), which is the most important result from a practical testing process viewpoint. Together with this effect, defect detection potential of the DCyT test cases produced by the experimental groups using the static testing was significantly higher (0.08 in average for all three test case intensities in range 1 to 3, measured by eff(Tp) metric, giving an average number of inserted defects, which can be potentially detected by one test case) than the same metric for the test cases produced by DCyT without proposed static testing only. Despite the fact we used defect injection technique to evaluate this effectiveness, the results are significantly in favor of the systematic static testing performed as a part of the test design process.

Regarding the related work and concepts already applied in the data consistency testing, innovation of the proposal can be summarized in the following aspects. (1) In software engineering and testing process, usually only one CRUD matrix is created during design or testing phase of the project. By proposing cross-verification of multiple CRUD matrices created by different parties on the project, which is not commonly applied idea, new, potentially cost efficient possibilities of static testing are available. (2) Suggestions to optimize the SUT design (e.g. duplicate data entities or functions, identification of possible “super-entities” or “super-functions”) are in the current praxis performed rather as part of model checking or revisions of the database design. In this process, analysts and database designers play a major role and testing specialists are not involved to this process. However, involvement of the testers in the process with their independent view can make the process more efficient and can lead to detection of more design flaw, in phase, where removal of these design defects is relatively not expensive in comparison to the later phases in the software development life-cycle. (3) The current industry-praxis reference literature, for example [14,18], comments only on a basic possibilities of static testing using the CRUD matrices. This article summarizes possible extensions of this static testing, available to the test practitioners.

**Acknowledgements** This research is conducted as a part of the project TACR TH02010296 Quality Assurance System for Internet of Things Technology and internal grant of CTU in Prague SGS17/097/OHK3/1T/13.

## References

1. Arévalo, G., Falleri, J.R., Huchard, M., Nebut, C.: Building abstractions in class models: formal concept analysis in a model-driven approach. In: MoDELS, vol. 4199, pp. 513–527. Springer, Berlin (2006)
2. Awad, A., Decker, G., Lohmann, N.: Diagnosing and Repairing Data Anomalies in Process Models, pp. 5–16. Springer, Berlin (2010). doi:10.1007/978-3-642-12186-9-2
3. Briand, L., Labiche, Y., Lin, Q.: Improving the coverage criteria of uml state machines using data flow analysis. Softw. Test. Verif. Reliab. **20**(3), 177–207 (2010). doi:10.1002/stvr.v20:3
4. Briand, L., Labiche, Y., Liu, Y.: Combining uml sequence and state machine diagrams for data-flow based integration testing. In: Proceedings of the 8th European Conference on Modelling Foundations and Applications, ECMFA'12, pp. 74–89. Springer, Berlin (2012). doi:10.1007/978-3-642-31491-9-8
5. Briand, L.C., Labiche, Y., Lin, Q.: Improving statechart testing criteria using data flow information. In: 16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05), pp. 10–104 (2005). doi:10.1109/ISSRE.2005.24
6. Bures, M., Cerny, T., Klima, M.: Prioritized Process Test: More Efficiency in Testing of Business Processes and Workflows, pp. 585–593. Springer, Singapore (2017). doi:10.1007/978-981-10-4154-9-67
7. Carbonnel, J., Huchard, M., Miralles, A., Nebut, C.: Feature model composition assisted by formal concept analysis. In: 12th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), pp. 28–29 (2017)
8. Cellier, P., Ducassé, M., Ferré, S., Ridoux, O.: Formal concept analysis enhances fault localization in software. Lect. Notes Comput. Sci. **4933**, 273–288 (2008)
9. Chandra, A., Singhal, A.: Study of unit and data flow testing in object-oriented and aspect-oriented programming. In: Innovation and Challenges in Cyber Security (ICICCS-INBUSH), 2016 International Conference on, pp. 245–250. IEEE (2016)
10. Denaro, G., Margara, A., Pezze, M., Vivanti, M.: Dynamic data flow testing of object oriented systems. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1, pp. 947–958. IEEE Press (2015)
11. Denaro, G., Pezze, M., Vivanti, M.: On the right objectives of data flow testing. In: Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on, pp. 71–80. IEEE (2014)
12. Dwarakanath, A., Jankiti, A.: Minimum number of test paths for prime path and other structural coverage criteria. In: Proceedings of the 26th IFIP WG 6.1 International Conference on Testing Software and Systems—Volume 8763, ICTSS 2014, pp. 63–79. Springer, New York Inc., New York (2014). doi:10.1007/978-3-662-44857-1-5
13. Frajtak, K., Bures, M., Jelinek, I.: Exploratory testing supported by automated reengineering of model of the system under test. Clust. Comput. **20**(1), 855–865 (2017). doi:10.1007/s10586-017-0773-z
14. Grood, D.J.D.: TestGoal: Result-Driven Testing, 1st edn. Springer Publishing Company, Heidelberg (2008)
15. Hema, M., Anup, S., Sen, K., Bagchi, A.: Detecting data flow errors in workflows: a systematic graph traversal approach (2007)
16. Jorgensen, P.C.: Software testing: a craftsmans approach. CRC Press, Hoboken (2016)
17. Jukic, B., Jukic, N., Nestorov, S.: Process and data logic integration: Logical links between uml use case narratives and er diagrams. J. Comput. Inf. Technol. **21**(3), 161–170 (2013)
18. Koomen, T., Aalst, L.V.D., Broekman, B., Vroon, M.: TMap Next, for Result-driven Testing. UTN Publishers, 's-Hertogenbosch (2013)
19. Kumar, S., Yadav, D., Khan, D.: Artificial bee colony based test data generation for data-flow testing. Indian J. Sci. Technol. **9**(39) (2016)
20. Küster, J.M., Ryndina, K., Gall, H.: Generation of business process models for object life cycle compliance. In: Proceedings of the 5th International Conference on Business Process Management, BPM'07, pp. 165–181. Springer, Berlin (2007). http://dl.acm.org/citation.cfm?id=1793114.1793131
21. Li, N., Li, F., Offutt, J.: Better algorithms to minimize the cost of test paths. In: Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, ICST '12, pp. 280–289. IEEE Computer Society, Washington, DC (2012). doi:10.1109/ICST.2012.108
22. Moser, S., Martens, A., Gorlach, K., Amme, W., Godlinski, A.: Advanced verification of distributed WS-BPEL business processes incorporating CSSA-based data flow analysis. In: IEEE International Conference on Services Computing (SCC 2007), pp. 98–105 (2007). doi:10.1109/SCC.2007.22
23. Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis. Springer, Berlin (2015)
24. Poelmans, J., Dedene, G., Snoeck, M., Viaene, S.: Using formal concept analysis for the verification of process-data matrices in conceptual domain models. In: Proceedings of the IASTED International Conference on Software Engineering, pp. 79–86. Acta Press (2010)
25. Prabu, M., Narasimhan, D., Raghuram, S.: An effective tool for optimizing the number of test paths in data flow testing for anomaly detection. In: Computational Intelligence, Cyber Security and Computational Models, pp. 505–518. Springer, Berlin (2016)
26. Su, T., Wu, K., Miao, W., Pu, G., He, J., Chen, Y., Su, Z.: A survey on data-flow testing. ACM Comput. Surv. **50**(1), 5 (2017)
27. Sun, S.X., Zhao, J.L., Nunamaker, J.F., Sheng, O.R.L.: Formulating the data-flow perspective for business process management. Inf. Syst. Res. **17**(4), 374–391 (2006). doi:10.1287/isre.1060.0105
28. Sundari, M.H., Sen, A.K., Bagchi, A.: Detecting data flow errors in workflows: a systematic graph traversal approach. In: WITS 2007—Proceedings, 17th Annual Workshop on Information Technologies and Systems, pp. 133–139 (2007). www.scopus.com
29. Tilley, T., Cole, R., Becker, P., Eklund, P.: A survey of formal concept analysis support for software engineering activities. Formal Concept Anal. **3626**, 250–271 (2005)
30. Trčka, N., van der Aalst, W.M.P., Sidorova, N.: Data-Flow Anti-Patterns: Discovering Data-Flow Errors in Workflows, pp. 425–439. Springer, Berlin (2009)
31. Waheed, S.Z., Qamar, U.: Data flow based test case generation algorithm for object oriented integration testing. In: Software Engineering and Service Science (ICSESS), 2015 6th IEEE International Conference on, pp. 423–427. IEEE (2015)
32. Wedyan, F., Ghosh, S., Vijayasarathy, L.R.: An approach and tool for measurement of state variable based data-flow test coverage for aspect-oriented programs. Information and Software Technology **59**, 233 – 254 (2015). doi:10.1016/j.infsof.2014.11.008. http://www.sciencedirect.com/science/article/pii/S0950584914002547

**Miroslav Bures** received his Ph.D. at Czech Technical University in Prague, Faculty of Electrical Engineering, where he currently works as a researcher and a senior lecturer in software testing and quality assurance. His research interests are model-based testing (process and workflow testing, data consistency testing) efficiency of test automation (test automation architectures, assessment of automated testability, economic aspects) and quality assurance methods for Internet of Things solutions, reflecting specifics of this technology. In these areas he also leads several R&D and experimental projects. He is a member of Czech chapter of the ACM, CaSTB, ISTQB Academia work group and participates in broad activities in professional testing community.

**Tomas Cerny** received his B.S., Engineer and Ph.D. degrees from Czech Technical University, the Faculty of Electrical Engineering of Czech Technical University in Prague, Czech Republic. He received his M.S. degree from Baylor University. He is Assistant Professor of Computer Science currently at Baylor University where he conducts research on Aspect-Oriented Programming, Security, and Software Engineering mostly related to Distributed Architectures including SOA, MSA or IoT.

**Karel Frajtak** has graduated the Czech Technical University in Prague, Czech Republic. He is currently a postgraduate student in Software Engineering Group at the CTU in Prague. In his research he focuses on model-based testing area and explores possible combinations and synergies between model-based testing and exploratory testing techniques to find more efficient methods of automation of the current testing processes.

**Bestoun S. Ahmed** obtained his B.Sc. degree in Electrical and Electronic Engineering from the University of Salahaddin-Erbil in 2004, his M.Sc. degree from University Putra Malaysia (UPM) in 2009, and his Ph.D. degree from University Sains Malaysia (USM), Software Engineering, in 2012. He worked as a research fellow attached to the Software Engineering Research Group in the Universiti Sains Malaysia (USM). His next engagement was as a senior lecturer at the Salahaddin University. He spent one year doing his post doctoral research in the Swiss AI Lab IDSIA (Istituto Dalle Molle di Studi sull'Intelligenza Artificiale), Switzerland. Currently, he is an assistant professor at the department of computer science, Czech Technical University in Prague. His main research interest include Combinatorial Testing, Search Based Software Testing (SBST), Computational intelligence, and High Performance Computing. He serves as a reviewer and editorial member for many international journals and organization committee member of many international conferences.

# 9 Appendix C: On the effectiveness of combinatorial interaction testing: A case study

[A.3] Bures, Miroslav, and Bestoun S. Ahmed. On the effectiveness of combinatorial interaction testing: A case study. In *2017 IEEE International Conference on Software Quality, Reliability and Security* (IEEE International Workshop on Combinatorial Testing and its Applications, Proceedings companion volume), pages 69-76. IEEE, 2017.

2017 IEEE International Conference on Software Quality, Reliability and Security (Companion Volume)

# On The Effectiveness of Combinatorial Interaction Testing: A Case Study

Miroslav Bures
Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University
Karlovo nám. 13, 121 35 Praha 2
Czech Republic
Email: buresm3@fel.cvut.cz

Bestoun S. Ahmed
Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University
Karlovo nám. 13, 121 35 Praha 2
Czech Republic
Email: albeybes@fel.cvut.cz

***Abstract*—Combinatorial interaction testing (CIT) stands as one of the efficient testing techniques that have been used in different applications recently. The technique is useful when there is a need to take the interaction of input parameters into consideration for testing a system. The key insight the technique is that not every single parameter may contribute to the failure of the system and there could be interactions among these parameters. Hence, there must be combinations of these input parameters based on the interaction strength. This technique has been used in many applications to assess its effectiveness. In this paper, we are addressing the effectiveness of CIT for a real-world case study using model-based mutation testing experiments. The contribution of the paper is threefold: First we introduce an effective testing application for CIT; Second, we address the effectiveness of increasing the interaction strength beyond the pairwise (i.e., interaction of more than two parameters); Third, model-based mutation testing is used to mutate the input model of the program in contrast to the traditional code-based mutation testing process. Experimental results showed that CIT is an effective testing technique for this kind of application. In addition, the results also showed the usefulness of model-based mutation testing to assess CIT applications. For the subject of this case study, the results also indicate that $3 - way$ test suite (i.e., interaction of three parameters) could detect new faults that can not be detected by pairwise.**

## I. Introduction

In software systems, the interaction of input parameters is strong potential source of failure. Most of the test design techniques are considered input parameters individually for fault detection of a software-under-test (SUT). However, evidence showed that interaction among input parameters is a strong source of failure[1], [2], [3]. Combinatorial Interaction Testing (CIT) (sometimes called $t - way$ or $t - wise$ testing where $t$ is the interaction strength) is an approach to overcome this shortage in the test design methods. CIT takes the interaction of two or more inputs in a generated test suite to help early fault detection in the testing life cycle [4]. The key insight of this testing approach is that, not every input parameter of the system contributes to the faults of the system and most of the faults are addressed by including interactions of only a few number of input parameters. To this end, the CIT is also useful to minimize the size of test suites by reducing the number of test cases and thus to prevent from the exhaustive testing as it is impractical and often also impossible from project resources viewpoint[5].

For instance, pairwise testing (i.e., $2 - way$ testing) is a common approach and has been applied effectively in many practical testing situations [6]. However, empirical evidence have shown that software failures may be triggered by more than two input parameters and values [4]. Such cases have been identified as $t - way$ CIT where $t > 2$.

Different strategies have been developed to generate $t-way$ test suites. In fact, some strategies have tried to generate test suites for high interaction strengths like $t = 12$ [7]. However, these strategies faced two main criteria, efficiency, and effectiveness. Efficiency is characterized by the size of the generated test suites as compared to other strategies, whereas, the effectiveness is characterized by the applicability of the generated test suites [8]. In this paper, we are focusing on the second criterion.

In general, effectiveness deals with the applicability of the generated test suite by the CIT strategy. Within this context, there are also remarkable issues. In this paper, we are trying to address three main issues. First, we are seeking to address a new testing application of CIT approach, that shows the effectiveness of the generated test suites. Second, we are seeking to show the usefulness of those test suites beyond pairwise testing. Third, we are trying to show the usefulness of model-based mutation testing to assess the effectiveness of CIT by following the concepts of code-based mutation testing. We have used a well-known open source software as a subject of our case study. Model-based mutation testing is used within the case study as a framework to assess the effectiveness. We injected the software with different mutant based on the input models and then tried to detect those mutants by generated $t-way$ test suites. Here, mutation testing concepts is used to mutate the input models of the input parameters and then we tried to measure the number of detected mutated inputs.

The rest of this paper is organized as follows. Section II gives preliminaries and necessary mathematical backgrounds about CIT. Section III shows the related works achieved so far in this direction. Section IV illustrates the experimental setup, and Section V shows and discusses the results of the

978-1-5386-2072-4/17 $31.00 © 2017 IEEE
DOI 10.1109/QRS-C.2017.20

experiments. Section VI presents and discusses the threats that may affect the validity of the experiments. Finally, Section VII gives the concluding remarks of the paper.

## II. Combinatorial Interaction Testing (CIT)

Combinatorial Covering Array (CA) is a mathematical object that is used as a base in CIT. Originally, CA has been gained more attention as a practical alternative of oldest mathematical object called Orthogonal Array (OA) that has been used for statistical experiments [9]. An $OA_\lambda(N;t,k,v)$ is an $N \times k$ array, where for every $N \times t$ sub-array, each $t - tuple$ occurs exactly $\lambda$ times, where $\lambda = N/v^t$; $t$ is the combination strength; $k$ is the number of input functions ($k \geq t$); and $v$ is the number of values associated with each input parameter [10]. Fig. 1(a) illustrates an orthogonal array $OA(9;2,4,3)$ that contains value ($v = 3$), with interaction degree ($t = 2$) , and input factors ($k = 4$) can be generated by nine rows. For real applications, practically, it is very hard to translate these firm rules except for small systems with small number of input parameters and values. Hence, there is no significant benefit in case of medium and large size systems, as it is very hard to generate OA for them. In addition, based on the aforementioned rules, it is not possible to represent OA when there are different levels for each input parameters.

To address the limitations of OA, CA has been introduced. A $CA_\lambda(N;t,k,v)$ is an $N \times k$ array over $(0,...,v-1)$ such that every $N \times t$ sub-array contains all ordered subsets from $v$ values of size $t$ at least $\lambda$ times, where the set of column $B = \{b_0,...,b_{t-1}\} \supseteq \{0,...,k-1\}$ [2]. In this case, each $t - tuple$ is to appear at least once in a CA. Fig. 1(b) shows an example of CA with $N = 9$, $k = 4$, $v = 3$, and $t = 2$.

In the case when the number of component values varies, this can be handled by Mixed Covering Array (MCA). A $MCA(N;t,k,(v_1,v_2,\dots v_k))$, is an $N \times k$ array on $v$ values, where the rows of each $N \times t$ sub-array cover and all $t-tuples$ of values from the $t$ columns occur at least once. For more flexibility in the notation, the array can be presented by $MCA(N;t,v_1^{k_1}v_2^{k_2}..v_k^{k})$. Fig.1(c) shows a MCA with size 9 that has four input parameters, with two input parameters having three values each and the other two input parameters having two values each.

## III. Related Works

CA has been used to solve many complex problems. It has been used widely in different applications such as hardware testing [11], advance material testing [12], gene expression regulation [13], performance evaluation of communication systems [14] and many other applications [15]. It has also been used as an essential testing framework in many software applications and software product lines (SPL) (e.g., [16], [17], [18], [19], [20]). More recently, we have also discovered many applications for CA such as optimization of dynamic voltage scaling (DVS) in high performance processors [21], direct current (DC) servomotor controller [22], tuning of functional order PID controller [23], fault detection of software systems [24], [2], [25], graphical user interface (GUI) testing [26].

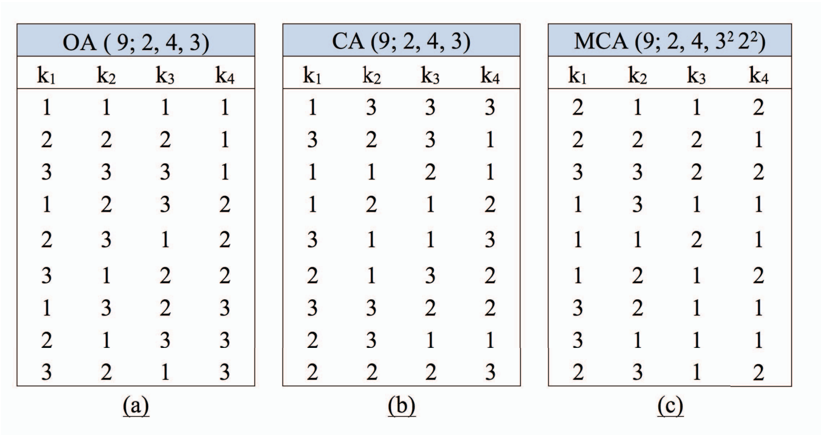
In fact, CA has been used wider in software testing activities for CIT. Bryce and Colbourn [27] introduced the prioritization concept within the CA to prioritize the test suites for regression testing. The key idea here is that the chance of repeated faults is frequent as the produced software is developed or maintained. Developers found that it is better to keep the test suites generated for the earlier version of the software and prioritize them based on specific mechanism. Hence, prioritization in this way increases the effectiveness of the test suites. Qu et al. [28] applied prioritization within CA successfully on two software subjects to examine the effectiveness of CA to find faults in the regression testing process. The results of the research showed that most of the faults can be found when $t = 2$ and 3.

CA has also been used to find faults' location in SUT within an approach of testing called fault characterization or failure diagnosis. Here, CIT is effective when the configuration space of the system is large. It helps to fix the faults quickly and saves a significant amount of time. Yilmaz et al.[29] applied CA for fault characterization on a software called "Skoll" that is used for distributed continuous quality assurance. The research found that even low interaction strengths can detect and localize faults effectively.

CIT is used widely with mutation analysis effectively. In fact, mutation testing is used within other testing approached as a powerful testing technique (see [30], [31]). Code-based mutation testing is used during the debugging process to evaluate fault detection capability of the test cases and thus identify the quality of the generated test suites [30]. Following this approach, few studies have used mutation testing to assess the quality of the test cases generated by CIT strategies (e.g., [32], [19], [18]). More recently, Belli et al. [33] have demonstrated the usefulness and the position of model-based mutation testing in the landscape of mutation testing. Here, model-based mutation testing is following the same concepts of traditional code-based mutation testing. However, the mutants are affecting the input model of the SUT. Contrary to those above-mentioned works in mutation testing, in this research, we are also trying to investigate the effectiveness of CIT using model-based mutation testing as a framework. Here, using model-based mutation testing concepts, we can know the quality of the used test cases and thus investigate the effectiveness. The following sections illustrate the experimental setup and the method of assessment.

## IV. Experimental Setup

As mentioned previously, we aim to introduce a new application of CIT and know the effectiveness of combinatorial interaction test suites. In doing so, we will be able to reach our second aim to understand the effectiveness of $t-way$ test suites. In our experiments, we measured the effectiveness of $t - way$ combinatorial test suites for a defined business case, which was reporting an issue in an issue-tracking system (i.e., SUT). We used the business cases to model the inputs of the issue-tracking system. We then mutate those input models and run the experiments mainly to detect defects in the SUT. To



| OA ( 9; 2, 4, 3) | | | |
|---|---|---|---|
| k1 | k2 | k3 | k4 |
| 1 | 1 | 1 | 1 |
| 2 | 2 | 2 | 1 |
| 3 | 3 | 3 | 1 |
| 1 | 2 | 3 | 2 |
| 2 | 3 | 1 | 2 |
| 3 | 1 | 2 | 2 |
| 1 | 3 | 2 | 3 |
| 2 | 1 | 3 | 3 |
| 3 | 2 | 1 | 3 |

(a)

| CA (9; 2, 4, 3) | | | |
|---|---|---|---|
| k1 | k2 | k3 | k4 |
| 1 | 3 | 3 | 3 |
| 3 | 2 | 3 | 1 |
| 1 | 1 | 2 | 1 |
| 1 | 2 | 1 | 2 |
| 3 | 1 | 1 | 3 |
| 2 | 1 | 3 | 2 |
| 3 | 3 | 2 | 2 |
| 2 | 3 | 1 | 1 |
| 2 | 2 | 2 | 3 |

(b)

| MCA (9; 2, 4, $3^2 2^2$) | | | |
|---|---|---|---|
| k1 | k2 | k3 | k4 |
| 2 | 1 | 1 | 2 |
| 2 | 2 | 2 | 1 |
| 3 | 3 | 2 | 2 |
| 1 | 3 | 1 | 1 |
| 1 | 1 | 2 | 1 |
| 1 | 2 | 1 | 2 |
| 3 | 2 | 1 | 1 |
| 3 | 1 | 1 | 1 |
| 2 | 3 | 1 | 2 |

(c)

Fig. 1. Examples illustrating OA, CA, and MCA [2]

simulate the defects, we prepared ten instances of the SUT, to which artificial defects were injected. We choose to use the SUT for the software development case. Here, base on our industrial experience, we manually mutate the inputs just like the standard code mutation in the code-based mutation testing. For each of these instances, we analyzed how effective are the individual test cases concerning the ability to detect the injected defects. The details of the experimental setup and the mutation process are given in the following sub-sections.

### A. Object of Experiment

As SUT, we have selected the open-source JTrac[1] application, that is written in J2EE. The JTrac is an issue tracking system with configurable data fields and issues workflow. It uses a relational database for data storage with standard HTML and AJAX user interface. In our experiment, we focused on the part of entering the issue and immediate follow-up processing of the entered data. We enriched the data processing logic in the JTrac by more complex verification of the issue data, entered into the system.

We modeled an issue entry form for 9 fields of configuration, which are subjects of our interest. In each of the fields, we identified equivalence classes (ECs), as in a standard industrial test design process. The configuration of the issue entry form is presented in Table I.

Then, we defined several combinations of values, which are not allowed in the issue tracking process. At this point, we copied similar setup from a recent industrial project we have been consulting. We extended that part of the JTrac which is responsible for data verification by a set of conditions ensuring these constraints. We extended the system by 26 conditions in total. Out of these, 16 conditions were guarding particular combinations of 2 input fields, and 10 conditions were guarding particular combinations of 3 input fields.

An example of the 2 input fields condition is: IF ((operational_system==iOS) AND (browse==Edge)) THEN the combination is not valid.

An example of the 3 input fields condition is: IF ((issue_severity==1) AND (issue_priority==1) AND (issue_preliminary_classification==usabiliy_defect)) THEN the combination is not valid.

This was the baseline SUT, which was considered as the correct version. We used this baseline input also for implementation of the test oracle, determining the expected correct results of the tests.

### B. Model-based Mutation Testing Process

We used the input model described in Section IV-A within a model-based mutation framework. Just like traditional code-based mutation testing, here, we prepared ten instances of baseline SUTs (mutated SUTs further on), to which we inserted different sets of artificial defects by standard code mutation techniques. We mutated the code affecting processing of the issue data after its input to the system via the form for entering issues. In this experiment, we focused on mutation of conditions f the input models. More details about mutated SUTs are given in Table II. It is worthy here to mention that we are not dealing with the code of JTrac.

In Table II, Column MUT-ALL gives a total number of mutated conditions in the SUT model. Column MUT-2 gives the number of mutated conditions, where two variables considered for inputs to the decision, whereas column MUT-3 gives the number of mutated conditions, where three variables considered for inputs to the decision. Moreover, columns "VALUE", "AND" and "NOT" are giving the numbers of particular input mutation type, which was made in the SUT input model. These input mutation types are explained in Table III, using pseudo-code notations. Here, x stands for variable, N and M for constants. The input elements presented in Table III can be part of more complex decision expressions.

[1] JTrac Official Web Page, http://jtrac.info/

TABLE I
CONFIGURATION OF ISSUE ENTRY FORM

| Field | Type | Number of ECs |
|---|---|---|
| Issue name | Short text | 5 |
| Issue description | Long text | 4 |
| Issue severity | List of values | 5 |
| Issue priority | List of values | 5 |
| Operational system | List of values | 5 |
| Browser | List of values | 6 |
| Testing environment | List of values | 5 |
| System | List of values | 7 |
| Issue preliminary classification | List of values | 5 |

TABLE II
PROPERTIES OF MUTATED SUTS

| Mutated SUT ID | MUT-ALL | MUT-2 | MUT-3 | VALUE | AND | NOT |
|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 2 | 1 | 0 | 1 | 1 | 0 | 0 |
| 3 | 1 | 0 | 1 | 1 | 0 | 0 |
| 4 | 2 | 0 | 2 | 2 | 0 | 0 |
| 5 | 3 | 1 | 2 | 3 | 0 | 0 |
| 6 | 4 | 1 | 3 | 2 | 2 | 0 |
| 7 | 5 | 2 | 3 | 3 | 2 | 0 |
| 8 | 6 | 2 | 4 | 5 | 0 | 1 |
| 9 | 7 | 3 | 4 | 3 | 0 | 4 |
| 10 | 8 | 4 | 4 | 3 | 1 | 4 |

TABLE III
INPUT MUTATION TYPES USED IN THE EXPERIMENTS

| Mutation type | Baseline code element | Mutated code element | Notes |
|---|---|---|---|
| VALUE | (x==N) | (x==M) | N≠M |
| AND | condition1 AND condition2 | condition1 OR condition2 | Alternatively, OR changed for AND |
| NOT | condition | NOT(condition) | Alternatively remove NOT |

To detect the defects caused by input mutations in the particular SUT instance, we created Selenium WebDriver[2] automated tests. The test scenario consisted of the following steps: (1) login to the system, (2) select the project, (3) go to the issue entry form, (4) enter the issue by prepared testing data, (5) submit the issue, (6) test the verification mechanism for the allowed issue data, (7) display the saved issue to test if it was saved correctly. The sequences (3)-(7) repeated for particular combinations of the test data, which were parametrized in a data grid structure. The experimental setup is outlined in the Fig. 2.

The test data (which were entered to the SUT during the experiments) were generated by our strategy PSTG that is developed recently. PSTG is a CIT strategy that uses Particle Swarm Optimization (PSO) to generate the combinatorial interaction test suite. Details about the PSTG test generation tool can be found in [34], [35]. The strategy generates the test suites by entering the specification of input models. We prepared different sets of test data: starting from $2-way$, where the test data combinations generated with full all pairs criterion for the input values and $t-way$, where the test data combinations were generated with full $t-way$ criterion. The following section shows and discusses the results of the evaluation of these test suites.

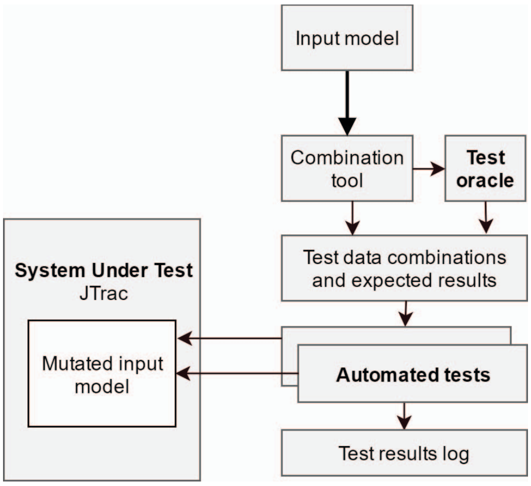


Fig. 2. Experiment setup

[2] http://www.seleniumhq.org/projects/webdriver/

## V. Results and Discussion

Using our PSTG tool, we have generated $t-way$ test suites starting from $2-way$ to $6-way$. We have started by 48 different test cases in the $2-way$ test suite and 322 different test cases in the $3-way$ test suite. We used baseline SUT (without inserted faults) to create a test oracle, determining the correct expected results of individual test cases. These expected results were added to the input data grid of the automated tests exercising particular mutated SUTs. We run each of the prepared test cases (TCs) in $2-way$ and $3-way$ test suites for all 10 mutated SUTs instances. The results are presented in Table IV.

In Table IV, DN denotes number of test cases that have detected the defects caused by the mutated input lines; EFF denotes percentage of test cases that have detected the defects caused by the mutated input lines. For three mutated SUTs (particularly 3,4 and 5), the defects were detected only when using a $3-way$ test cases. The input mutants which were not detected by $2-way$ test suite, but where detected by the $3-way$ test suite are specified in the Table V.

It is also important to know the effectiveness of each test case in addition to the test suites. These individual effectiveness results are presented in Fig. 3 for the $2-way$ test suite and Fig. 4 for the $3-way$ test suite. In Fig. 3, the graph displays the number of mutated SUTs, in which the individual test case of the $2-way$ test suite discovered a defect caused by a input mutation. By analogy, Fig. 4 displays this statistics for the $3-way$ test suite.

Fig. 5 presents the ratio of test cases detecting the input mutation defects for the $2-way$ test suite. On the $x-axis$, the number of mutated input lines in particular mutated SUTs is displayed.

Then, Fig. 6 presents the ratio of test cases detecting the input defects for the $3-way$ test suite.

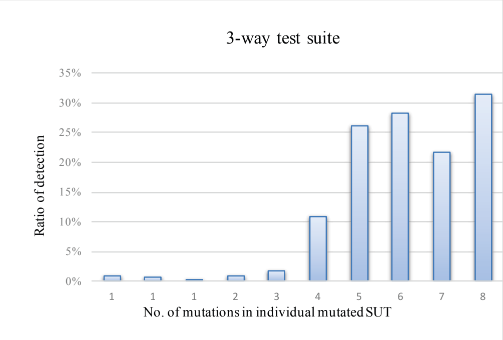


Fig. 6. Ratio of test cases detecting the mutated input defects for the $3-way$ test suite

It is worthy to mention that the other generated test suites (i.e., $4-way$ to $6-way$) may also detect faults in the program. However, for this particular application, we can see from the results that $3-way$ test suites can detect all the mutants successfully.

In addition to the effectiveness of the test suites as a whole, another important observation can be drawn from the results, which is the effectiveness of each test case. As can be seen from Fig.5 and Fig. 6, the mutant detection ration is not distributed uniformly among the test cases. Each test case has a different capability of detection. This could be beneficial in term of test case prioritization while accompanied by regression testing. Here, we can rearrange the test cases in the test suite in which those test cases with higher detection ratio can be prioritized when running in the future.

In particular, 19 out of 48 test cases of the $2-way$ test suite have not detected any defect in all of 10 mutated SUTs. This means, 39,6% of $2-way$ test suite test cases have not detected any defect in all of 10 mutated SUTs used in the experiment. One test case of the $2-way$ test suite has detected 1,176 defects in all of SUTs in average. For $3-way$ test suite, 130 out of 322 test cases have not detected any defect in all of the mutated SUTs, which is 40,4% in ratio. One test case of the $3-way$ test suite has detected 1,230 defects in all of 10 SUTs in average.

When we consider these statistics as a certain measure of test set efficiency, for 2-way and 3-way test suites this efficiency is is practically comparable. From an overall economic point of view, 2-way test set seems more efficient, taking into account the total number of test cases in $2-way$ and $3-way$ test sets. Nevertheless, the significant difference is more capability of $3-way$ test set to detect all the inserted defects; in our example 3 of the inserted defects were detected by $3-way$ test set only.

## VI. Threats to Validity

Like any other experimental and evaluation research, this research has faced few threats to validity. We have tried to eliminate the effect of those threats. However, some threats are out of our control in research. The first threat is the generalization and expansion of the results. For this experimental object of this research, we have selected limited input elements for mutation. There could be different elements of the input that may take various types of mutant. In fact, this threat depends on the aim of the experiment. Here our aim is not to evaluate JTrac extensively, but we are using it to proof the effectiveness.

The second possible threat is the use of front-end testing and model-based mutation testing to detect the defective behavior of the SUT. Although it can be thought as a threat, we took this approach to simulate real-life test cases.

Another threat is that the set of used mutated input types are limited. Here, we tried to estimate the most likely developer's mistakes. It is hard to predict which defects are made by the developers in the input line for the particular case. Quality and consistency of the design documentation, seniority of the development staff and coding standards have an impact on the particular case.

The use of one generation could form a threat to validity. In the literature, there are different approaches for constructing combinatorial interaction test suites. Each approach may lead to different effectiveness impact on the SUT. Here, in line with

TABLE IV
TEST RESULTS

| Mutated SUT ID | DN for $2-way$ | EFF for $2-way$ | DN for $3-way$ | EFF for $3-way$ |
|---|---|---|---|---|
| 1 | 1 | 2,08% | 3 | 0,93% |
| 2 | 1 | 2,08% | 2 | 0,62% |
| 3 | 0 | 0,00% | 1 | 0,31% |
| 4 | 0 | 0,00% | 3 | 0,93% |
| 5 | 0 | 0,00% | 6 | 1,86% |
| 6 | 3 | 6,25% | 35 | 10,87% |
| 7 | 12 | 25,00% | 84 | 26,09% |
| 8 | 12 | 25,00% | 91 | 28,26% |
| 9 | 10 | 20,83% | 70 | 21,74% |
| 10 | 17 | 35,42% | 101 | 31,37% |

TABLE V
INPUT MUTANTS WHICH WERE NOT DETECTED BY $2-way$ TEST SUITE

| Mutated SUT ID | Mutation ID | Baseline input element | Mutated input element |
|---|---|---|---|
| 3 | 1 | IF ((test_environment==DEV) AND (issue_preliminary_classification==performance_defect) AND (system==backoffice)) THEN the combination is not valid | IF ((test_environment==UAT1) AND (issue_preliminary_classification==performance_defect) AND (system==backoffice)) THEN the combination is not valid |
| 4 | 1 | IF ((test_environment==DEV) AND (issue_preliminary_classification==performance_defect) AND (system==backoffice)) THEN the combination is not valid | IF ((test_environment==UAT1) AND (issue_preliminary_classification==performance_defect) AND (system==backoffice)) THEN the combination is not valid |
| 4 | 2 | IF ((issue_severity==2) AND (issue_priority==2) AND (issue_preliminary_classification==usability_defect)) THEN the combination is not valid | IF ((issue_severity==3) AND (issue_priority==2) AND (issue_preliminary_classification==usability_defect)) THEN the combination is not valid |
| 5 | 1 | IF ((issue_severity==1) AND ((issue_priority==4) OR (issue_priority==5)) THEN the combination is not valid | IF ((issue_severity==1) AND ((issue_priority==5) OR (issue_priority==5)) THEN the combination is not valid |
| 5 | 2 | IF ((test_environment==DEV) AND (issue_preliminary_classification==performance_defect) AND (system==backoffice)) THEN the combination is not valid | IF ((test_environment==UAT1) AND (issue_preliminary_classification==performance_defect) AND (system==backoffice)) THEN the combination is not valid |
| 5 | 3 | IF ((issue_severity==2) AND (issue_priority==2) AND (issue_preliminary_classification==usability_defect)) THEN the combination is not valid | IF ((issue_severity==3) AND (issue_priority==2) AND (issue_preliminary_classification==usability_defect)) THEN the combination is not valid |

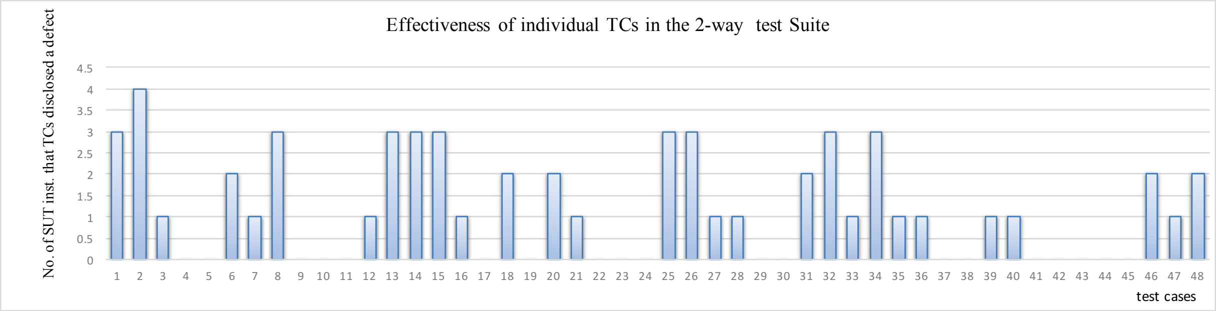


Fig. 3. Effectiveness of the individual test cases for the $2-way$ test suite

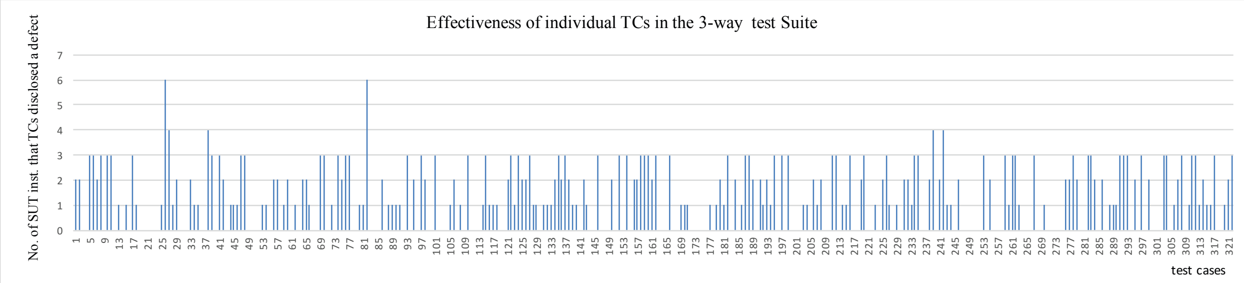


Fig. 4. Effectiveness of the individual test cases for the $3 - way$ test suite

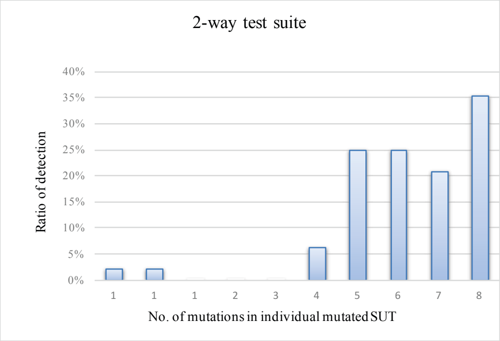


Fig. 5. Ratio of test cases detecting the mutated input defects for the $2 - way$ test suite

our aim, we are not showing the effectiveness of each CIT strategy. However, we keen to demonstrate the effectiveness of CIT within the model-based mutation testing for a new case study. Here, in contrast to the traditional code-based mutation testing, we can detect different input mutants by following the model-based mutation testing process.

## VII. Conclusion

In this paper, we have investigated a new case study of CIT for model-based mutation testing. We have examined different combinatorial interaction test suites on a well-known established open source software. The experiments aimed to find input model mutants that we injected into the SUT to check the effectiveness of the test suites. In addition to the critical application of CIT that can be seen in the paper, we have also shown how to use CIT in the context of model-based mutation testing as a variant of traditional code-based mutation testing. We have also shown that pairwise testing is not sufficient to detect all the input mutants for this case study. The results revealed that $3 - way$ test suite is effective also to detect some of those not detected faults by pairwise test suites. This research is an active research project currently, and we are trying to investigate more case studies with more mutated inputs for CIT and its effectiveness in the future. As part of our future research, we are also planning to examine the effectiveness of different CIT strategies to study the effect of test construction and parameter arrangements on the effectiveness of the whole test suites.

## References

[1] R. N. Kacker, D. Richard Kuhn, Y. Lei, and J. F. Lawrence, "Combinatorial testing for software: An adaptation of design of experiments," *Measurement*, vol. 46, no. 9, pp. 3745–3752, 2013.

[2] B. S. Ahmed, T. S. Abdulsamad, and M. Y. Potrus, "Achievement of minimized combinatorial test suite for configuration-aware software functional testing using the cuckoo search algorithm," *Information and Software Technology*, vol. 66, no. 0, pp. 13–29, 2015.

[3] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton, "The combinatorial design approach to automatic test generation," *IEEE Softw*, vol. 13, 1996.

[4] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys*, vol. 43, no. 2, pp. 1–29, 2011.

[5] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "Ipog: a general strategy for t-way software testing," in *4th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, pp. 549–556, IEEE Computer Society. 1253335 549-556.

[6] A. Hervieu, D. Marijan, A. Gotlieb, and B. Baudry, "Practical minimization of pairwise-covering test configurations using constraint programming," *Information and Software Technology*, vol. 71, pp. 129–146, 2016.

[7] K. Z. Zamli, M. F. J. Klaib, M. I. Younis, N. A. M. Isa, and R. Abdullah, "Design and implementation of a t-way test data generation strategy with automated execution tool support," *Information Sciences*, vol. 181, no. 9, pp. 1741–1758, 2011.

[8] X. Yuan, M. B. Cohen, and A. M. Memon, "Gui interaction testing: incorporating event context," *IEEE Transactions on Software Engineering*, vol. 37, no. 4, pp. 559–574, 2011.

[9] C. S. Cheng, "Orthogonal arrays with variable numbers of symbols," *The Annals of Statistics*, vol. 8, no. 2, pp. 447–453, 1980.

[10] C. Colbourn, G. Kéri, P. R. Soriano, and J.-C. Schlage-Puchta, "Covering and radius-covering arrays: Constructions and classification," *Discrete Applied Mathematics*, vol. 158, no. 11, pp. 1158–1180, 2010.

[11] A. Hartman, *Software and Hardware Testing Using Combinatorial Covering Suites*, vol. 34 of *Graph Theory, Combinatorics and Algorithms*. Springer US, 2005.

[12] J. N. Cawse, *Experimental design for combinatorial and high throughput materials development*. Wiley-Interscience, 2003.

[13] D. E. Shasha, A. Y. Kouranov, L. V. Lejay, M. F. Chou, and G. M. Coruzzi, "Using combinatorial design to study regulation by multiple input signals: A tool for parsimony in the post-genomics era," *Plant Physiology*, vol. 127, no. 4, pp. 1590–1594, 2001.

[14] D. S. Hoskins, C. J. Colbourn, and D. C. Montgomery, "Software performance testing using covering arrays: Efficient screening designs with categorical factors," in *Proceedings of the 5th International Workshop on Software and Performance*, WOSP '05, (New York, NY, USA), pp. 131–136, ACM, 2005.

[15] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test case generation," *IEEE Transactions on Software Engineering*, vol. 36, no. 6, pp. 742–762, 2010.

[16] C. J. Colbourn and D. W. McClary, “Locating and detecting arrays for interaction faults,” *Journal of Combinatorial Optimization*, vol. 15, no. 1, pp. 17–48, 2008.

[17] G. Fraser and A. Gargantini, “Generating minimal fault detecting test suites for boolean expressions,” in *Third International Conference on Software Testing, Verification, and Validation Workshops*, pp. 37–45, 2010.

[18] M. B. Cohen, M. B. Dwyer, and J. Shi, “Constructing interaction test suites for highly-configurable systems in the presence of constraints: a greedy approach,” *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 633–650, 2008.

[19] J. Petke, S. Yoo, M. B. Cohen, and M. Harman, “Efficiency and early fault detection with lower and higher strength combinatorial interaction testing,” in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, (New York, NY, USA), pp. 26–36, ACM, 2013.

[20] P. A. da Mota Silveira Neto, I. d. Carmo Machado, J. D. McGregor, E. S. de Almeida, and S. R. de Lemos Meira, “A systematic mapping study of software product lines testing,” *Information and Software Technology*, vol. 53, no. 5, pp. 407–423, 2011.

[21] D. R. Sulaiman and B. S. Ahmed, “Using the combinatorial optimization approach for dvs in high performance processors,” in *International Conference on Technological Advances in Electrical Electronics and Computer Engineering (TAEECE)*, pp. 105–109, 2013.

[22] M. A. Sahib, B. S. Ahmed, and M. Y. Potrus, “Application of combinatorial interaction design for dc servomotor pid controller tuning,” *Journal of Control Science and Engineering*, vol. 2014, p. 7, 2014.

[23] B. S. Ahmed, M. A. Sahib, L. M. Gambardella, W. Afzal, and K. Z. Zamli, “Optimum design of PI$^{\lambda}$ D$^{\mu}$ controller for an automatic voltage regulator system using combinatorial test design,” *PLOS ONE*, vol. 11, pp. 1–20, 11 2016.

[24] B. S. Ahmed, K. Z. Zamli, and C. P. Lim, “Application of particle swarm optimization to uniform and variable strength covering array construction,” *Applied Soft Computing*, vol. 12, no. 4, p. 1330â1347, 2012.

[25] B. S. Ahmed and K. Z. Zamli, “A variable strength interaction test suites generation strategy using particle swarm optimization,” *Journal of Systems and Software*, vol. 84, no. 12, pp. 2171–2185, 2011.

[26] B. S. Ahmed, M. A. Sahib, and M. Y. Potrus, “Generating combinatorial test cases using simplified swarm optimization (sso) algorithm for automated gui functional testing,” *Engineering Science and Technology, an International Journal*, vol. 17, no. 4, pp. 218–226, 2014.

[27] R. C. Bryce and C. J. Colbourn, “Test prioritization for pairwise interaction coverage,” in *Proceedings of the 1st International Workshop on Advances in Model-based Testing*, A-MOST ’05, (New York, NY, USA), pp. 1–7, ACM, 2005.

[28] X. Qu, M. B. Cohen, and K. M. Woolf, “Combinatorial interaction regression testing: a study of test case generation and prioritization,” in *IEEE International Conference on Software Maintenance, ICSM 2007*, pp. 255–264, IEEE Computer Society, 2007.

[29] C. Yilmaz, M. B. Cohen, and A. Porter, “Covering arrays for efficient fault characterization in complex configuration spaces,” *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 45–54, 2004. 1007519.

[30] J. Offutt, “A mutation carol: Past, present and future,” *Information and Software Technology*, vol. 53, no. 10, pp. 1098 – 1107, 2011. Special Section on Mutation Testing.

[31] Y. Jia and M. Harman, “An analysis and survey of the development of mutation testing,” *IEEE Trans. Softw. Eng.*, vol. 37, pp. 649–678, Sept. 2011.

[32] D. R. Kuhn, D. R. Wallace, and J. Gallo, A.M., “Software fault interactions and implications for software testing,” *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 418–421, 2004.

[33] F. Belli, C. J. Budnik, A. Hollmann, T. Tuglular, and W. E. Wong, “Model-based mutation testing – approach and case studies,” *Science of Computer Programming*, vol. 120, pp. 25 – 48, 2016.

[34] B. S. Ahmed, L. M. Gambardella, W. Afzal, and K. Z. Zamli, “Handling constraints in combinatorial interaction testing in the presence of multi objective particle swarm and multithreading,” *Information and Software Technology*, vol. 86, pp. 20 – 36, 2017.

[35] B. S. Ahmed, K. Z. Zamli, and C. P. Lim, “Constructing a t-way interaction test suite using the particle swarm optimization approach,” *International Journal of Innovative Computing, Information and Control (IJICIC)*, vol. 8, no. 1, pp. 431–452, 2012.

# 10 Appendix D: Constrained Interaction Testing: A Systematic Literature Study

[A.4] Bestoun S. Ahmed, Kamal Z. Zamli, Wasif Afzal, and Miroslav Bures. Constrained Interaction Testing: A Systematic Literature Study. *IEEE Access*, 5, pages 25706-25730. 2017. (Q1, IF 3.24)

IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received August 18, 2017, accepted October 15, 2017, date of publication November 9, 2017, date of current version December 5, 2017.

*Digital Object Identifier 10.1109/ACCESS.2017.2771562*

# Constrained Interaction Testing: A Systematic Literature Study

**BESTOUN S. AHMED[1], KAMAL Z. ZAMLI[2], (Member, IEEE), WASIF AFZAL[3], AND MIROSLAV BURES[1]**

[1] Software Testing Intelligent Laboratory, Department of Computer Science, Faculty of Electrical Engineering, Czech Technical University in Prague, 121 35 Prague, Czech Republic
[2] Faculty of Computer Systems and Software Engineering, University Malaysia Pahang, Gambang 26300, Malaysia
[3] School of Innovation, Design and Engineering, Mälardalen University, 72123 Västerås, Sweden

Corresponding author: Bestoun S. Ahmed (albeybes@fel.cvut.cz)

This work was supported in part by the Fundamental Research Grant: Reinforcement Learning Sine Cosine Based Strategy for Combinatorial Test Suite from the Ministry of Higher Education Malaysia under Grant RDU170103.

**ABSTRACT** Interaction testing can be used to effectively detect faults that are otherwise difficult to find by other testing techniques. However, in practice, the input configurations of software systems are subjected to constraints, especially in the case of highly configurable systems. Handling constraints effectively and efficiently in combinatorial interaction testing is a challenging problem. Nevertheless, researchers have attacked this challenge through different techniques, and much progress has been achieved in the past decade. Thus, it is useful to reflect on the current achievements and shortcomings and to identify potential areas of improvements. This paper presents the first comprehensive and systematic literature study to structure and categorize the research contributions for constrained interaction testing. Following the guidelines of conducting a literature study, the relevant data are extracted from a set of 103 research papers belonging to constrained interaction testing. The topics addressed in constrained interaction testing research are classified into four categories of constraint test generation, application, generation and application, and model validation studies. The papers within each of these categories are extensively reviewed. Apart from answering several other research questions, this paper also discusses the applications of constrained interaction testing in several domains, such as software product lines, fault detection and characterization, test selection, security, and graphical user interface testing. This paper ends with a discussion of limitations, challenges, and future work in the area.

**INDEX TERMS** Constrained interaction testing, constrained combinatorial testing, software testing, test generation tools, test case design techniques.

## I. INTRODUCTION

Software has become an innovative key for many applications and methods in science and engineering. Ensuring the quality and correctness of software is challenging because of the different configurations and input domains of each program. Ensuring the quality of software demands the exhaustive evaluation of all possible configurations and input interactions against their expected outputs. However, such an exhaustive testing effort is impractical because of time and resource limitations. Thus, different sampling techniques have been used to sample these input domains and configurations. The use of these sampling techniques for black box system testing is usually called as interaction testing, which can be used to detect faults that are otherwise undetectable effectively. Interaction testing has been given other alternative names such as combinatorial testing (CT), combinatorial interaction testing (CIT), and $t-way$, $t-wise$, or $n-wise$ testing (where $t$ or $n$ indicate the interaction strength). However, throughout this paper, the term ''interaction testing'' is used as a representative term as it is a popular name in existing software testing literature.

Interaction testing has been used successfully for testing different configurable software systems. Several review and survey papers exist on the topic that covers developed strategies and their applications (see e.g. [1]–[3]). Although useful, interaction testing has suffered from a limitation of efficiently handling constraints. The ability to handle constraints is a crucial aspect for the real-world applicability of interaction testing techniques since most of the real-world systems are subjected to constraints among input parameters or among particular system configurations. Hence, recent times have seen a shift in interaction testing research that concerns the handling of constraints and is typically called as constrained interaction testing [4]. Adding this feature opens up several new directions for research that promises to guide further development of interaction testing [5]. However, there exists no comprehensive and dedicated review paper in this direction.

Although many interaction testing strategies have been developed in the past, only a few of them can satisfy the constraints in the final generated test suite. Handling constraints add extra complexity in designing efficient interaction testing strategies. Hence, in the last decade, researchers have looked into different ways of supporting the generation of constrained interaction test suites. Besides, application of constrained interaction testing on various software systems and the associated empirical evaluations have started to surface. To this end, this paper provides a comprehensive systematic literature study to structure and categorize the available evidence for constrained interaction testing research during the last decade. The goal of the study is to identify the relevant papers, their results and the type of research such that one can discuss future research opportunities in the area. The study uses a systematic method to collect and analyze the related research published during the last decade. In doing so, methods and approaches for the generation as well as their applications are addressed.

The remainder of this paper is organized as follows: Section II presents the motivation and the overview of related work for this study. Section III describes the methodology of the literature study. Section IV presents the results and outcomes of the study. Section V discusses the threats to validity. Finally, Section VI concludes the work.

## II. MOTIVATION AND RELATED WORK

Multiple factors motivated the decision to carry out a literature study in this paper. First, no existing paper aggregates available research in *constrained* interaction testing (although several review papers exist on interaction testing in general). Second, constrained interaction testing is an upcoming research direction in interaction testing [6]; so it is interesting to investigate this topic. Third, a literature study paper is a community service that potentially saves significant time for interested researchers in getting to know about a research topic such as constrained interaction testing.

Nie and Lueng [2] conducted one of the first comprehensive reviews covering combinatorial testing and its applications. The study focuses on the basic concepts of combinatorial testing, the detailed methods of combinatorial test suite construction and the associated applications. The article also reviews constrained interaction testing methods. Kuliamin and Petukhov [7] also presented various methods of constructing combinatorial interaction test suites. On similar lines, Ahmed and Zamli [8] conducted a review study on the application of interaction testing. This paper is a useful complement to these existing review papers as it presents a more complete and recent review of the field. The previous review papers are not exhaustive in its coverage of constrained interaction testing. They are also not conducted as systematic literature studies.

Focusing on search-based test generation, Afzal *et al.* [1] and Ali *et al.* [9] have dedicated parts in their systematic literature reviews for interaction testing. More recently, Lopez-Herrejon *et al.* [3] published the first literature study on interaction testing for software product lines (SPL). Here, due to the presence of different constraints in SPL testing, the study has included and reviewed the constrained interaction testing. However, the study has only discussed constraints from the SPL application point-of-view.

This study considers the above-mentioned review papers as an excellent source of information since some of them include papers on constrained interaction testing. Our study further takes inspiration from other recently published systematic literature studies, e.g., [10], [11].

## III. METHOD

This section illustrates the method that this literature study follows. This study follows the methodology recommendations given by [12]. The methodology has six stages. First is the definition of the research questions. Second is to undertake the search process, in which the search strategy is established, and the primary research papers are selected. The third stage is the selection and the quality assessment; this stage acts as a screening stage in which irrelevant papers are excluded based on the title, abstract, full-text reading and quality assessments. Data extraction is the fourth stage to extract data from the remaining papers. The fifth stage is the analysis and the data classification to classify the extracted information from the papers by tabulating and analyzing them. The last stage is the validity evaluation in which the threats to validity are evaluated and presented. For better illustration and organization of these steps, they were further classified into three main phases, as shown in Figure 1.

- *Phase 1. Searching:* The research questions that determine the focus of the study are defined in this phase. Based on these research questions, the search string is designed. The search string has undergone an experimental refinement process to identify and return only closely relevant papers.
- *Phase 2. Filtering:* Here, the relevant papers are selected, and their quality is assessed. Irrelevant papers are excluded based on the title, abstract, full-text reading and quality assessments.
- *Phase 3. Analysis:* The relevant data answering the research questions are extracted from the primary set of papers in this phase (in the case of this study 103 papers). The extracted data from the primary papers are classified and analyzed to visualize and understand the outcome. Here, Tables and Figures are used. Threats to validity are also analyzed and presented in this phase with the aim to disclose possible limitations of this study.

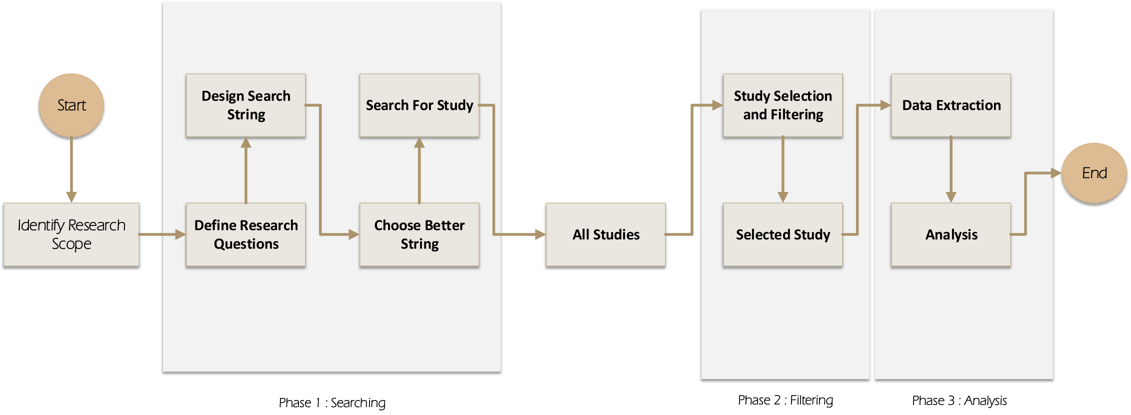


**FIGURE 1. The systematic literature detail steps.**

The following sections illustrate different stages covering the phases mentioned above that have been undertaken in detail. Input and output of each step are depicted in Figure 1. Some of the following subsections describe more steps joined. For example, defining research questions needs the research scope to be defined first. Hence, they are included in one subsection. In addition, the design, selection, and optimization of the search string are merged into one section.

### A. RESEARCH QUESTIONS

As mentioned before, this study is a systematic literature study, and the goal is to structure and categorize the available evidence for constrained interaction testing research during the last decade. A number of research questions (RQs) were formulated to help achieve our goal:

- *RQ1:* What is the evolution in the number of published studies over the last decade in constrained interaction testing?
- *RQ2:* Which individuals, organizations, and countries are active in conducting constrained interaction testing research?
- *RQ3:* What topics/subjects have been addressed in the constrained interaction testing research and what is their distribution?
- *RQ4:* What are the existing strategies, tools, and techniques that support the generation of constrained interaction test suites?
- *RQ5:* What kinds of benchmarks (industrial or otherwise) are used to evaluate constrained interaction testing techniques and what is their provenance?
- *RQ6:* What are the applications of constrained interaction in software testing?
- *RQ7:* What are the current limitations and challenges in constrained interaction test generation?
- *RQ8:* What are the possible directions for future research?

### B. SEARCH STRATEGY

Identifying the keywords for textual search is a challenging task. Kitchenham and Charters [12] established the PICO (Population, Intervention, Comparison, and Outcomes) criteria to identify the keywords formally. 'Population' refers to a role in software engineering, an application area or a discipline in the field, while the 'intervention' refers to software engineering tools, methodologies, procedures or strategies to address a specific issue. 'Comparison' identifies the different procedures or methods that have been used for comparison. 'Outcomes' deal with those keywords that are outcomes of the research and development, which are essential for practitioners such as improving performance or reliability.

Based on the research questions and the PICO guidelines, the keywords were categorized into three sets. The first set is related to the scoping the search for constrained interaction testing, i.e., "constrained interaction testing" or "constrained combinatorial interaction." To make the search broader, the second set of keywords is constructed to form terms and strings related to the generation of constrained test suites such as "strategy." The third set is related to the application of constrained interaction testing. These strings were combined to form one string but with different trials. To combine the search terms, Boolean AND is used, whereas Boolean OR is used to join alternate terms.

A preliminary string was constructed and then it took several trials to form the final search string (Set # 5 in Table 1). These trials were needed mainly due to the close relationship between combinatorial interaction testing and constrained interaction testing. Since constrained interaction testing is the scope of this paper, a search string is required that returns only those combinatorial interaction strategies that support constraints. In doing so, these search strings were evaluated based on how much the returned results were related to the scope. In order to make sure that the search string is not missing out relevant papers, 20 "pilot" papers were selected, to make sure that different changes to the search (when experimented on IEEExplore and ScienceDirect indexing databases) are always able to find these 20 core/pilot set of studies. Finally, the string that is more related to the study area and can return these studies was selected.

Table 1 shows the result of five trials of different search strings. The first three sets of the search strings were excluded

**TABLE 1. Search strings tries on the indexing data bases.**

| Keyword | Searching String | Returned Results | Missing Studies |
|---|---|---|---|
| Set # 1 | (constrained) AND ((interaction testing) OR (combinatorial testing) OR (covering array) OR (t-way testing)) AND (testing) AND (strategy OR technique OR method OR approach) | 242,439 | 0 |
| Set # 2 | (constrained OR constraint OR constraints) AND ((interaction testing) OR (combinatorial testing) OR (covering array) OR (t-way testing)) AND (testing) AND (strategy OR technique OR method OR approach) | 511,049 | 0 |
| Set # 3 | (((constrained OR constraint OR constraints) AND ("interaction testing" or "combinatorial testing" or "combinatorial interaction testing" or "covering array" or t-way testing) AND (strategy OR technique OR method OR approach) )) | 226,894 | 0 |
| Set # 4 | (constrained OR constraint OR constraints) AND ("interaction testing" OR "combinatorial testing" OR "covering array" OR "t-way testing") AND (testing) AND (strategy OR technique OR method OR approach) | 1,102 | 3 |
| Set # 5 | (constrained OR constraint OR constraints) AND ("interaction testing" OR "combinatorial testing" OR "combinatorial interaction" OR "combinatorial test design" OR "covering array" OR "t-way testing") AND (testing) AND (strategy OR technique OR method OR approach OR tool OR application) | 1,172 | 0 |

in the selection process since there were many irrelevant results returned by them, even though they returned the pilot papers also. The reason behind these results is the general terms in the strings that led to return many irrelevant papers. In addition, during the trials, it has been found that the term "constraint" is used in different ways depending on its situation in the sentences. To this end, three different terms were used, (constraint OR constrained OR constraints), which led to covering more papers. Additionally, it was also found that these synonyms were used with different terms of interaction testing. As a result, those terms were observed in the literature and used in different ways ("interaction testing" OR "combinatorial testing" OR "combinatorial interaction" OR "combinatorial test design" OR "covering array" OR "t-way testing"). To make sure that the scope was fully covered in the research questions, additional terms were added such as ("strategy," "technique," "method," "approach" and "tool").

The databases were selected based on the guidelines and suggestions provided by [13] and [14]. Based on these guidelines, the following databases were selected:

- IEEE Xplore
- ScienceDirect
- ACM Digital Library
- Scopus
- SpringerLink

During searching, indexing, and sorting of a large number of references, different duplicate references appeared due to the slight differences in the reference indexing in the databases. To manage the references and to remove duplicates, the well-known reference management software EndNote X7 was used. For more accuracy, Mendeley v1.16 reference manager software was also used. As mentioned earlier, this is a literature study covering the last decade starting from 2005 as there has been increasing research trends from that time. It should be mentioned here that this study started in early 2016 and finished early 2017. The papers from 2017 are not included in this study. To figure out the number of published research in this direction, Table 2 summarizes the number of research papers published in the mentioned period for each considered database.

**TABLE 2. Number of published research.**

| Database | Search Results |
|---|---|
| IEEE Xplore | 456 |
| ScienceDirect | 716 |
| ACM Digital Library | 43 |
| Scopus | 659 |
| SpringerLink | 794 |
| Total | 2,668 |

### C. PAPER SELECTION CRITERIA AND QUALITY ASSURANCE

The papers were excluded or selected based on the title, abstract and full-text reading. The quality of the papers was also considered for the selection. To increase the reliability of selection, this process was conducted by the first author and reviewed again by other authors of the study. It should be mentioned that some papers can be selected or excluded based on the title and abstract. However, rest of them required full-text reading for deciding on their selection. For better understanding, the studies with the following criteria were selected:

- Studies for interaction testing with the support of constraints.
- Studies dealing with the applications of constrained interaction.
- Studies that are in the field of Software Engineering/Computer Science.
- Studies that are published online in the last decade (i.e., from 2005 when the first constraint-related paper got published).

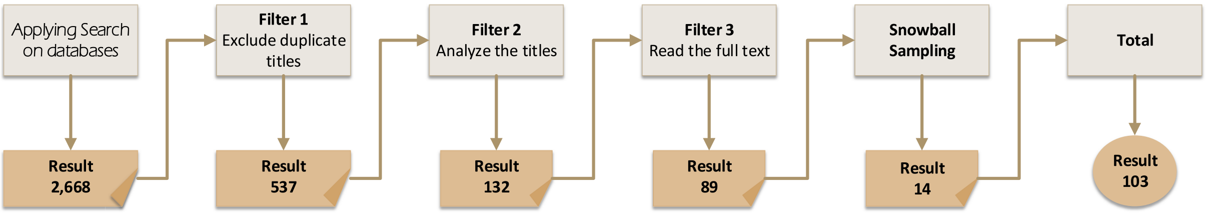

Applying Search on databases
Result 2,668
Filter 1 Exclude duplicate titles
Result 537
Filter 2 Analyze the titles
Result 132
Filter 3 Read the full text
Result 89
Snowball Sampling
Result 14
Total
Result 103

**FIGURE 2.** Number of included papers in the selection and filtering process.

It is also worth mentioning here the excluding criteria for the studies. Following the guidelines provided by [14], the published studies were excluded based on the following criteria:

- Studies dealing with interaction testing but without the support of constraints.
- Application of constrained interaction in fields other than Software Engineering/Computer Science.
- Studies not published in the English language.
- Studies without full text.
- Books and gray literature.
- Studies from non-peer reviewed sources.

Applying these criteria helped to capture better the number of papers that should be included in the study scope. As mentioned previously, many of these papers were duplicated, and they were removed finally. For example, some of those published papers in ScienceDirect were also indexed in Scopus database. In fact, Scopus acted as a valuable resource for double checking the results from other databases. Figure 2 shows these studies and the selection stages clearly.

As can be seen from Figure 2, to choose papers from a large set given by the selected databases, four filtering stages were applied. These stages have also been used in other literature studies [10], [15]. First, the related papers were identified in the selected databases (i.e., IEEE Xplore, ScienceDirect, ACM Digital Library, Scopus, SpringerLink). As mentioned previously, the outcome of this stage is 2,668 papers. It should be mentioned here that different papers were shared between these databases. Hence, "Filter 1" stage was performed in which all the duplicated titles, proceeding abstracts, and PowerPoint presentations were excluded. In the "Filter 2" stage, the papers were analyzed by reading the titles, abstracts, and if necessary the introduction sections of the papers. The selected papers were based on the exclusion and inclusion criteria provided earlier. In the "Filter 3" stage, the authors read the full-text of the chosen papers. Here, another set of papers was excluded due to multiple reasons. For example, many conference papers described an idea for research but do not include study results. In addition, the focus of some published papers was not entirely in software engineering and also not falling within the scope of the research questions. Finally, a snowballing stage was conducted by checking the references of the selected studies to not miss any relevant papers. Here, 14 more papers were added as an outcome of this stage. Hence, at the last stage, 103 papers were selected to answer the research questions by extracting information from them. Appendix A lists the studied papers. Appendix A shows the references for these papers along with their full names and the publication years.

**TABLE 3.** Data extraction template.

| Data item | Value |
|---|---|
| Study ID | Integer |
| Paper Title | Name of the paper |
| Author Name | Name of author(s) |
| Year of Publication | Calendar year |
| Venue | Name of publication venue |
| Country | Name of the country for each participated author |
| Area of research | Knowledge area of research |
| Research topic | Main topic or theme addressed by the study |
| Research problem | Research problem addressed by the study |
| Proposal | Proposed solution to the problem |
| Contribution | Main contribution of the paper |
| Challenges | Challenges addressed in the paper |
| Evaluation process | Which benchmark adopted for evaluation? |
| Case study | Which case study used? |

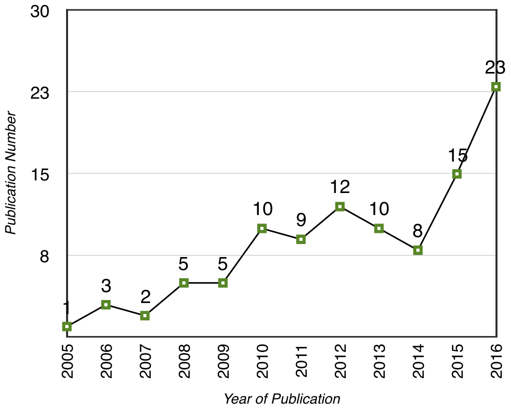


**FIGURE 3.** Publication per year.

## D. DATA EXTRACTION AND ANALYSIS

This phase aims to extract data from the selected studies and analyze them to answer the research questions. A spreadsheet was created to retrieve the required data from these identified studies. The template developed by [14] and [16] was followed and adopted to construct the sheet. More fields of data were also added to the table. Table 3 shows the template that was used. General information about the paper was recorded, including paper ID, publication title, publication year, authors' names and countries, venue, and area

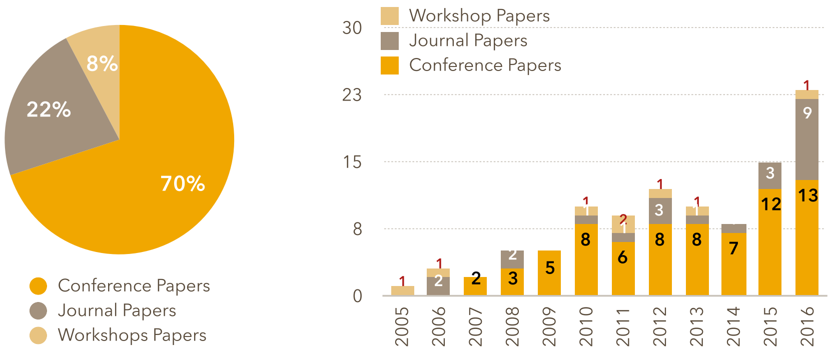


**FIGURE 4.** Publication ratio and number categorized by source.

of research. More specific data was extracted by including the research approach for the study, evaluation process, case study, applied techniques and challenges addressed. The data extraction process helped to understand each paper's aim better and to get answers to the posed research questions. At the end of the process, the frequencies of papers were also calculated.

Each paper has a table with the information specified in Table 3. To extract and analyze the information for all papers, a reliable method was followed. The information is extracted first by the first author and then double checked by the other authors separately. For better reliability, automatic text analyzer also used to verify the obtained information.

## IV. RESULTS

This section is dedicated to answering the research questions in detail. Each research question from section III-A is addressed here individually. A short title is used for each section that is extracted from the main research questions in section III-A.

### A. FREQUENCY OF PUBLICATIONS (RQ1)

The identified studies were analyzed over the last decade (2005–2016) to know the frequency and evolution of the number of publications. Figure 3 shows the results of this analysis process. As mentioned earlier, 103 publications were considered in which the average number of publications per year is of 10 papers. The average number is influenced strongly by the growth of publication numbers after 2009.

It should be mentioned here that the first model of constrained interaction for interaction testing purposes is proposed in a Ph.D. thesis in 2004 by Cohen [17]. However, this study did not appear in any database and was only published on the author (and university) website. In 2005, Hnich *et al.* [18] showed how to model handle constraints in Covering Array (CA); however, they did not explicitly address the constraints among the values of the input parameters, and they mention this as a problem to be solved in the future. In 2006, Bryce and Colbourn [19] formalized the constrained interaction testing with CA mathematical object, while Hnich *et al.* [20] also defined and formalized the constraint models for CA in the same year. In the same time, Cohen *et al.* [21] tried to investigate the application of constrained interaction testing for SPL testing.

The interest in constrained interaction testing moderately increased between 2005–2007, whereas, a significant increase of research can be observed in 2008 and beyond. This increase in the publication number is an indication of the increasing interest in researching constrained interaction testing in the software engineering community. Another potential reason behind this increase is a shift of constrained interaction testing from theory to practice, with more and more papers investigating the application of this testing technique in different case studies.

Regarding the type of the publication venue, Figure 4 shows this information. Majority of publications (around 70%) are conference publications, about 22% are journal publications, and around 8% are workshop publications. It should be mentioned here that some conference publications are ultimately published as book chapters; however, their original venues, which are conferences, were considered.

Given these results, it is also important to know the popular peer-reviewed journal, conference and workshop venues for constrained interaction testing. Figure 5 shows those active peer-reviewed journals where relevant papers are published. Journals names are given in abbreviations of Thomson Reuters Science Citation Index[1] due to their long names. The full journal names corresponding to those abbreviations are given in Appendix B. Respectively, Figure 6 shows active conferences involved in constrained interaction testing research. The conference names are given in abbreviations, and the full names can be found in Appendix C.

Figures 5 and 6 gives a clear picture of targeted venues for publication by authors of the considered studies. Looking at the journal publications specifically, it is clear that "Software Quality," "Information and Software Technology," "IEEE Transactions on Software Engineering," and "Systems and Software," journals are the most active and top four journals

[1] https://apps.webofknowledge.com

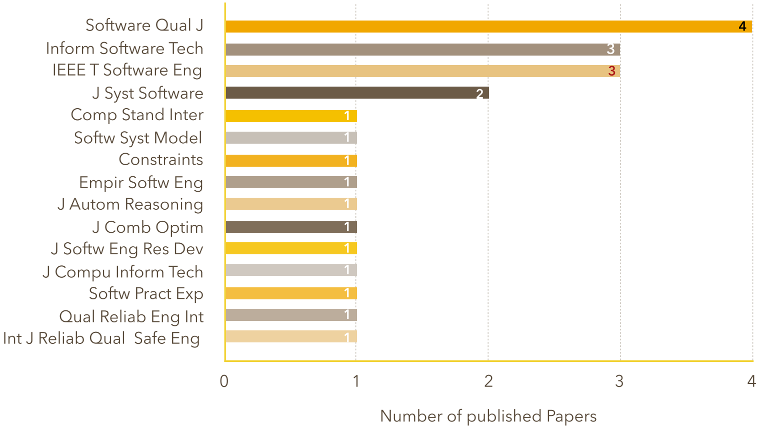


**FIGURE 5.** Number of Published Papers vs. Journal Name.

in terms of publication target (more than 52% of the journal publications). Considering the conference publications, it be can noted that many papers have been published in individual conferences, however, "Software Testing, Verification and Validation (ICST)" and "Software Product Lines (SPLC)" are the most targeted venues for the authors (more than 36% of conference papers; 19 in ICST and 7 in SPLC). However, if we consider those conferences which published more than two papers, more than 68% of the papers were published in annual conferences. The remainder of the papers were published in 30 individual conferences (represented as others in Figure 6).

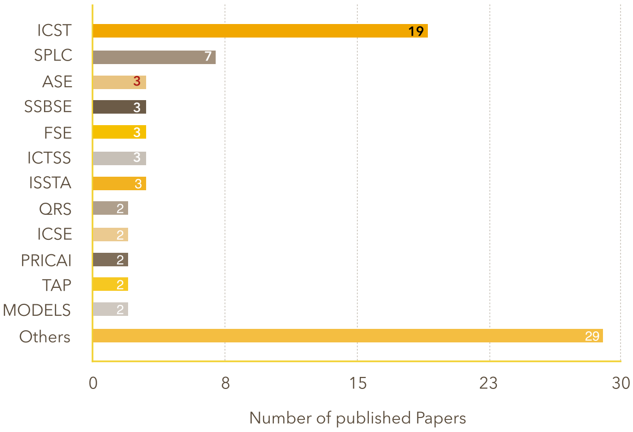


**FIGURE 6.** Amount of Published Papers vs. Conference Venues.

### B. ACTIVE INDIVIDUALS, ORGANIZATIONS AND COUNTRIES (RQ2)

This research question aims to identify active researchers publishing studies about constrained interaction testing. A quick analysis reveals that many researchers are engaged in researching constrained interaction testing. In fact, it is clear that many authors were participating in a single research paper. Most active researchers were designated as those who author/co-author more than three research papers. Such researchers are producing more than 83% (86/103) of the publications. Figure 7 shows the ranking of these researchers based on them being authors or co-authors of the papers. From the ranking, it is clear that "Myra B. Cohen", "Angelo Gargantini" and "Andrea Calvagna" are top three researchers with 13, 10 and 6 published papers respectively. These three authors participated in almost more than 28% (29/103) of the total publications.

Table 4 shows the ranking of the organizations and active research groups based on the published papers. The name of the group, participating researchers, reference to the published papers and the total number of papers is also presented in the table. The table complements the observations from Figure 7. In addition to the organizational ranking, the table also shows the collaboration among the authors. From the analysis of the table, it is clear that the research group from University of Nebraska - Lincoln (a collaboration between "Myra B. Cohen" and "Matthew B. Dwyer") is the most active research group in constrained interaction testing. In addition, that the group from Chinese Academy of Sciences (a collaboration between Jian Zhang and Feifei Ma) and the group from Technischen Universität Darmstadt (a collaboration between Malte Lochau and Sebastian Oster) are second active research groups since they are collaborating with other researchers in publishing 7 papers for each group. There are two active groups in the third rank. The group from Università di Bergamo (a collaboration between "Angelo Gargantini", and "Paolo Vavassori"), and University of Catania (active author "Andrea Calvagna") have participated in 6 published papers.

Another analysis can be drawn from the active countries in published papers. Figure 8 shows the most active countries in the publishing of constrained interaction testing papers. The figure shows the participation of each country in published papers based on the organizational affiliation of the authors. It is clear that USA, Germany, and Italy are top three countries in publications, with 28, 14, and 12 publications respectively. For example, 13 papers out of those published

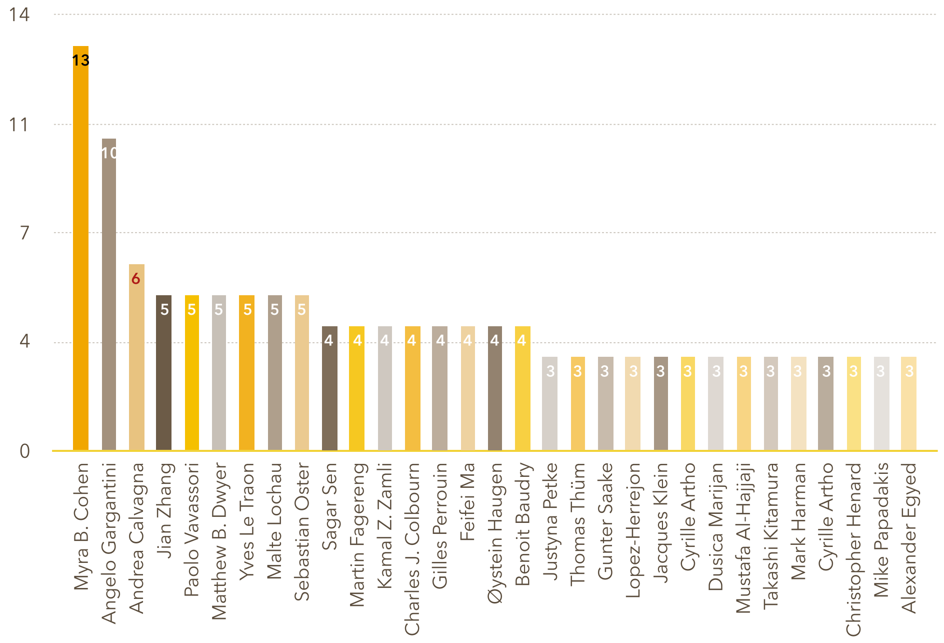


**FIGURE 7.** Active researchers.

**TABLE 4.** Researchers and organizations involved in constrained interaction testing research.

| Organization | Author(s) | Papers Published | Total |
|---|---|---|---|
| University of Nebraska - Lincoln | Myra B. Cohen and Matthew B. Dwyer | [21]–[33] | 13 |
| Chinese Academy of Sciences | Jian Zhang and Feifei Ma | [34]–[40] | 7 |
| Technische Universität Darmstadt | Malte Lochau and Sebastian Oster | [41]–[47] | 7 |
| Università di Bergamo | Angelo Gargantini, and Paolo Vavassori | [48]–[53] | 6 |
| University of Catania | Andrea Calvagna | [30], [49], [51], [52], [54], [55] | 6 |
| University of Luxembourg | Yves Le Traon , Jacques Klein, Christopher Henard, and Mike Papadakis | [44], [56]–[59] | 5 |
| IRISA/INRIA | Sagar Sen and Benoit Baudry | [44], [57], [60]–[62] | 5 |
| University of Oslo and SINTEF ICT | Martin Fagereng Johansen | [63]–[66] | 4 |
| University Sains Malaysia / University Malaysia Pahang | Kamal Z. Zamli | [67]–[70] | 4 |
| SINTEF ICT | Øystein Haugen | [63]–[66] | 4 |
| University of Namur | Gilles Perrouin | [44], [57], [59], [71] | 4 |
| University College London | Justyna Petke and Mark Harman | [5], [29], [31] | 3 |
| Technische Universität Braunschweig | Thomas Thüm | [41], [43], [72] | 3 |
| University of Magdeburg | Gunter Saake and Mustafa Al-Hajjaji | [41], [43], [72] | 3 |
| Johannes Kepler University Linz | Roberto E. Lopez-Herrejon and Alexander Egyed | [73]–[75] | 3 |
| Arizona State University | Charles J. Colbourn | [19], [76], [77] | 3 |
| National Institute of Advanced Industrial Science and Technology-Japan | Cyrille Artho, and Takashi Kitamura | [78]–[80] | 3 |
| Simula Research Laboratory | Dusica Marijan | [60], [61], [81] | 3 |

from the USA comes from a collaboration between "Myra B. Cohen" and "Matthew B. Dwyer". Germany is the second most active country in constrained interaction testing publications. This comes from the collaboration of three German universities with other groups. These organizations are, "Technischen Universität Darmstadt" (active researchers: Malte Lochau and Sebastian Oster), "Technischen Universität Braunschweig" (active researcher: Thomas Thüm) and the "University of Magdeburg" (active researchers: Gunter Saake and Mustafa Al-Hajjaji).

Important observations can be made from the output of this research question. Although the topic is important and promising, few researchers participate in constrained interaction testing publications. For instance, 31 active authors are

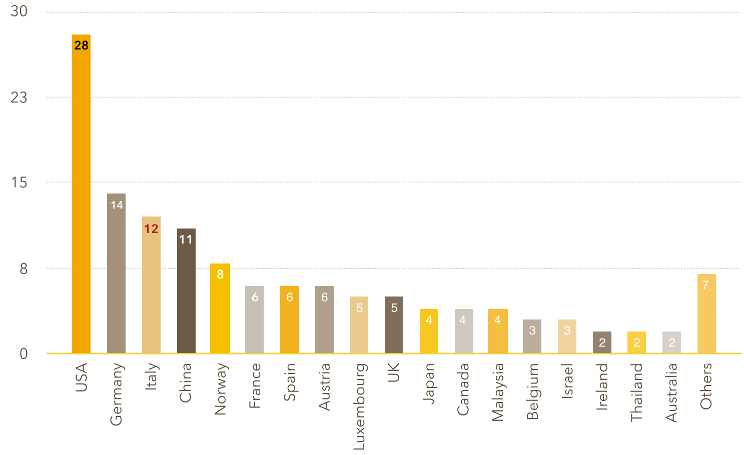


**FIGURE 8.** Active countries.

**TABLE 5.** Distribution of papers by research type (high level classification).

| Research Type | Number | Papers |
|---|---|---|
| Constraint Test Generation Studies | 36 | [18]–[20], [22]–[26], [34]–[38], [40], [48]–[51], [56], [67]–[69], [76]–[79], [82]–[91] |
| Application Studies | 37 | [27]–[30], [41], [42], [44], [45], [47], [52], [57], [58], [62]–[64], [71], [73], [80], [92]–[110] |
| Generation and Application Studies | 21 | [5], [31]–[33], [39], [43], [59]–[61], [65], [66], [70], [72], [74], [75], [81], [111]–[115] |
| Model Validation Studies | 9 | [21], [46], [54], [55], [116]–[120] |

participating in more than two research papers, and they are responsible for more than 83% (86/103) of the publications. In fact, many more authors are participating in combinatorial interaction testing without constraint's support.

### C. DISTRIBUTION OF THE STUDIES AND TOPICS ADDRESSED (RQ3)

Full-text of the considered papers was scanned carefully to answer this research question. Different topics and subjects have been addressed. Although the topics are broad in range, this study distributed the topics into four main high-level categories. Within each category, many topics can be discussed. Table 5 shows these four high-level categories and refers to the papers in each category. Based on a careful analysis, the studies are placed into the following categories:

- *Constraint Test Generation Studies:* Papers in this category discuss the generation strategies, methods, and approaches for constrained interaction testing. The papers are addressing the problem of generation by using different approaches including, exact methods, computational algorithms, meta-heuristic algorithms, and constraint solvers.
- *Application Studies:* Papers in this category consider only the application of constrained interaction testing for a specific domain of research (such as Graphical User Interface (GUI) testing).
- *Generation and Application Studies:* Papers in this category shows those studies that are considering a combination of generation and application. Here, the research papers are introducing a generation approach for a specific application(s).
- *Model Validation Studies:* Papers in this category either introduce models of constrained interaction testing, formalize the constrained interaction testing mathematically, or introduce models of problems that can be solved by constrained interaction testing.

Figure 9 shows the detail distribution of the papers according to the classified topics. The following sub-sections illustrate these categories and the papers related to them in detail.

#### 1) CONSTRAINT TEST GENERATION STUDIES

As can be seen from Table 5, 36 papers are proposing and evaluating different generation strategies of constrained interaction test suites. In addition, there are also 21 papers proposing customized generation strategies for specific applications. Hence, in total, there are 57 papers containing generation methods in some form. Here, it is essential to know the used methods for solving and dealing with constraints and also the generation strategies. In fact, the generation strategies will be addressed in RQ4, while this section addresses the constraint solving methods.

As can be seen from Figure 10, different methods have been used to solve the constraints during the generation of constrained interaction test suites. By analyzing those 57 papers, a classification for those constraint solving methods can be made. Out of those 57 papers, 43 papers (75.43%)

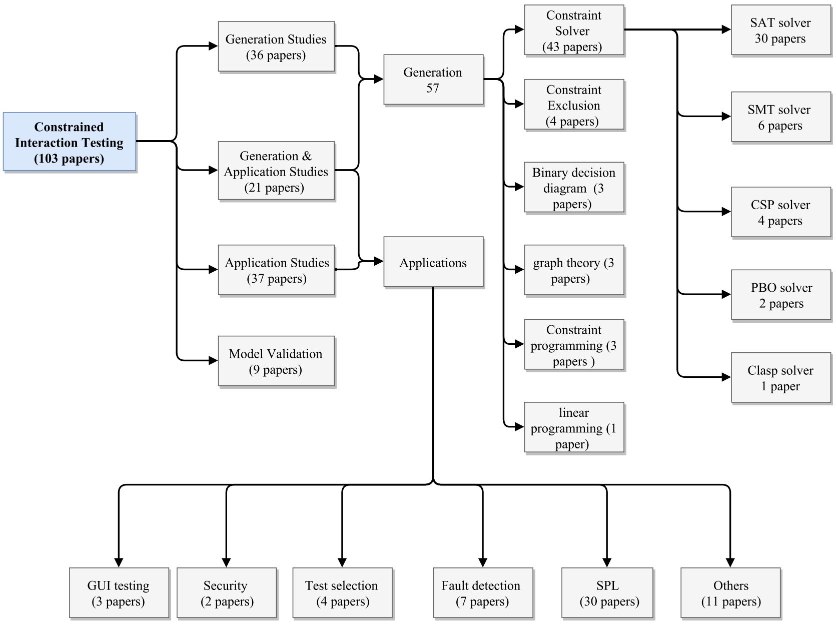

Constrained Interaction Testing (103 papers)
Generation Studies (36 papers)
Generation & Application Studies (21 papers)
Application Studies (37 papers)
Model Validation (9 papers)
Generation 57
Applications
Constraint Solver (43 papers)
Constraint Exclusion (4 papers)
Binary decision diagram (3 papers)
graph theory (3 papers)
Constraint programming (3 papers )
linear programming (1 paper)
SAT solver 30 papers
SMT solver 6 papers
CSP solver 4 papers
PBO solver 2 papers
Clasp solver 1 paper
GUI testing (3 papers)
Security (2 papers)
Test selection (4 papers)
Fault detection (7 papers)
SPL (30 papers)
Others (11 papers)

**FIGURE 9.** Distribution of papers according to classification.

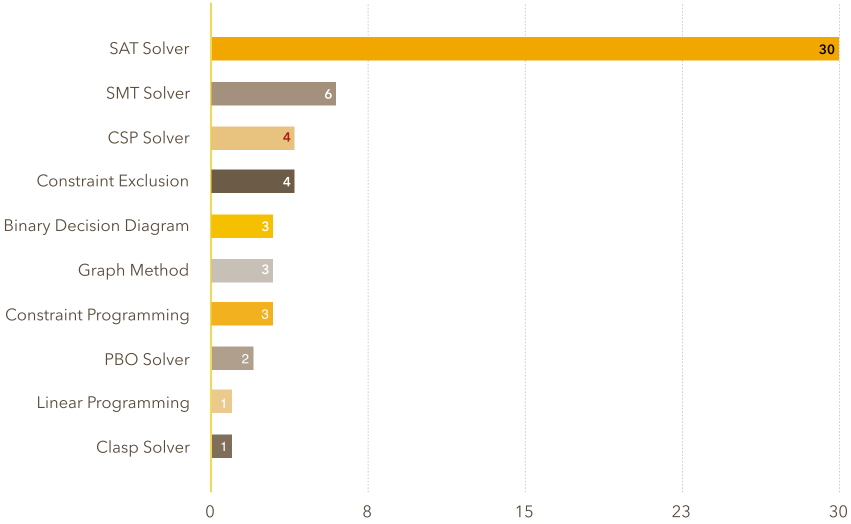


**FIGURE 10.** Constraint solving methods.

used a constraint solver package. Those constraint solvers are, SAT, SMT, CSP, PBO and Clasp solvers. SAT solver has the highest usage rate 69.7% (30/43) [5], [18], [20], [22]–[26], [28], [30], [31], [34], [38], [40], [41], [44], [45], [56]–[59], [65], [72], [73], [78], [79], [85], [95], [100], [113] then SMT solver 13.9% (6/43) [33], [44], [51], [54]–[56], then CSP solver 9.3% (4/43) [5], [42], [60], [99], and then PBO solver 4.6% (2/43) [37], [39]. Recently, one paper uses Clasp solver also [36]. The use of SAT solver seems to attract researchers because of its performance, accuracy, and simplicity for solving the constraints. Henard *et al.* [56] validated this result recently by conducting a comparative study between SAT and SMT solvers in case of flattening the CIT into a Boolean model. The research found that the SAT solver can process the flattening models faster than the SMT solver.

Another reported way of solving the constraints in the literature is by excluding the constraints from the search process. For instance, 4 papers [19], [42], [87], [98] of the generation papers follow this method. The method removes those tuples which are related to the constraints, in other words, the meaningless tuples. In this way, there is no need for a constraint solver. However, this method just considers the exclusion of constraints rather than their inclusion. For real applications, in some cases, there could be inclusion constraints. For example, some input parameters might come exclusively with a specific set of parameters.

Binary decision diagram [86], [115], [118] and graph theory [27], [88], [89] methods are also used to deal with the constraints. Within the considered papers for generation approach, there are three papers for each of these meth-

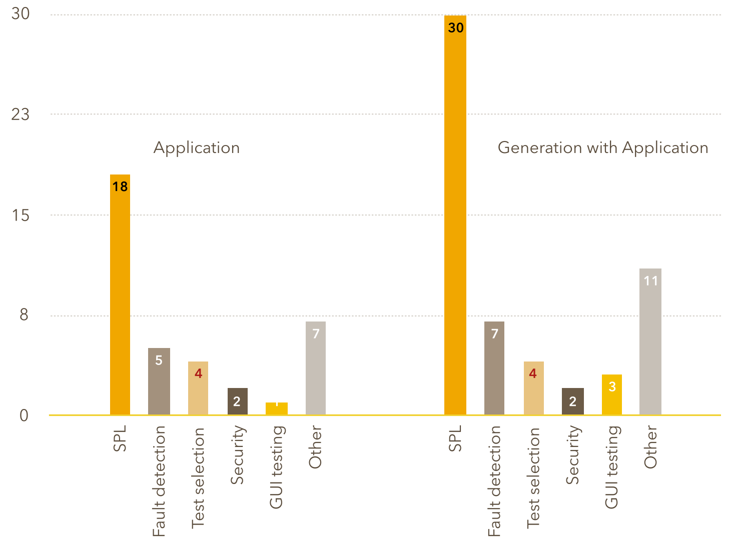


**FIGURE 11. Application areas of constrained interaction testing.**

ods. The binary decision diagram prevents the appearance of constraints in the final test suite by considering some form of cause-effect graph relationships. By considering this relationship, the generation method prevents the generation path from passing through those constraints. The graph theory method follows the well-known graph algorithms like graph coloring to generate test small test suites without violating the specified constraints.

Constraint programming [61], [62], [81] and linear programming [74] methods are also used rarely in the literature to deal with the constraints in combinatorial interaction testing. Constraint programming is mainly dedicated for solving hard combinatorial problems. Not much different from this definition, constraint programming is used within constrained interaction testing as an exact method to generate test suites without violating the constraints. Although it can be used with generation strategies in general, this method has been used only with SPL testing. Linear programming is also used with some similarities with this approach to solving the constraints but follows certain mathematical models. In general, linear programming is used with a mathematical model when the inputs have a linear relationship. Lopez-Herrejon *et al.* [74] followed this approach by using zero-one mathematical linear program with a multi-objective algorithm to solve the constraint problem for SPL feature models. In fact, this approach is difficult to follow for big and general constrained interaction testing strategies as it is a non-deterministic polynomial (NP) hard problem [4], [121].

### 2) APPLICATION STUDIES

From the Table 5, it is clear that 37 published papers are dealing with the application of constrained interaction testing without addressing the details of the generation process. The majority of these strategies used well-known and established tools and algorithms for generating constrained test suites. Those generation tools and algorithms are either adopted from other studies or used without much focus on them in the published studies due to the more significant focus on the application itself. For generating the test suites, there is an essential need to represent the system-under-test as a model to recognize the input parameters and constraints. These inputs are then used to generate test cases. Figure 12 shows those adopted methods in the application studies. Out of those 37 published papers in the application direction, eight papers are using SAT solver directly to generate the test cases and for solving the constraints. Eight papers used ACTS[2] tool for the generation since it is a well-established, efficient, and an excellent performance tool. Specifically, those published papers used the IPO-family algorithms inside the ACTS tool.

As can be seen from Figure 11, the applications are distributed into six main areas. Constrained interaction testing is frequently used within five of those areas actively. Those active areas (with the citations to them) are, SPL testing [27], [41], [42], [44], [45], [47], [52], [57], [58], [62]–[64], [71], [99], [101], [104], [109], [110], fault detection and characterization [28], [29], [80], [95], [105], test selection [96], [97], [107], [108], security [93], [102], and GUI testing [114]. The left-hand side of the graph shows the results of those research papers which are presenting the application focus. It should be mentioned here that the papers are showing the results of the testing process without much detail about the generation process. In fact, most of the research papers use one or more generation tools either from their previous research or other research. For instance, 15 research papers show the use and the benefits of the constrained interaction testing for the SPL. In total, 29 published papers are focusing on the use of constrained interaction testing for SPL.

Fault detection and characterization is another active research direction here. Fault detection has been examined

[2] http://csrc.nist.gov/groups/SNS/acts/index.html

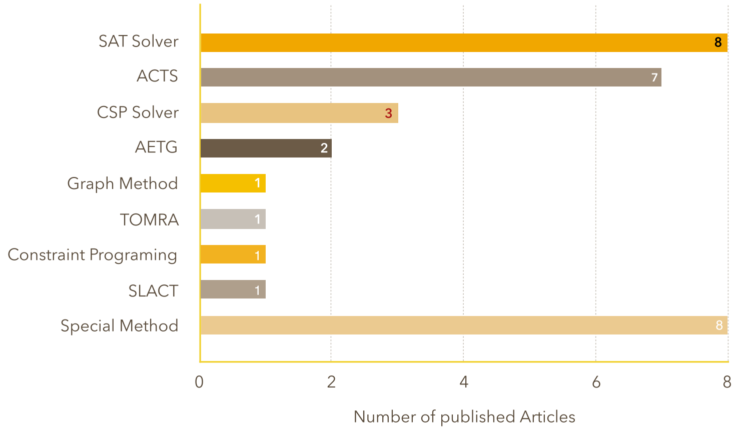


**FIGURE 12. Adopted generation methods for applications.**

in the literature with normal interaction testing without constraints. Within constrained interaction testing, fault detection and characterization is used for different purposes. In fact, there are five research papers published in this direction without the generation details and two more papers with the test generation details. In total, there are seven published papers, but there are no frequent author names in them.

Another active research direction for application of constraint testing is the test selection direction. From Figure 11, it is clear that in total there are four published papers here. The test selection research originated from the normal combinatorial interaction testing. Again, there are no frequent names in these publications. This direction of research seems to catch less attraction by the researchers, but it still could be a productive future research direction.

Constrained interaction testing has been used for testing security issues. Security application is investigated first for normal combinatorial interaction testing for few studies (e.g., [122], [123]). In case of constraints, so far, two studies investigated its use in security. Bozic *et al.* [93] assessed the concept of constrained interaction for web security testing and Bouquet *et al.* [102] assessed it with model-based testing.

Combinatorial interaction testing has been used effectively to generate test cases for applications from the GUI point of view. Here, the constrained interaction testing is used to solve invalid test cases in the final generated test suite. In total, there are three published papers in this direction.

In addition to those aforementioned frequent areas, constrained interaction testing has been used in total within 11 different studies to solve real-life problems. More details on the above mentioned active applications and other areas are given while answering RQ6 in subsection IV-F.

### 3) GENERATION AND APPLICATION STUDIES

Another category of research area within constrained interaction testing belongs to the "generation and application studies." As can be noted from Table 5, 24 published papers are dealing with the application of constrained interaction testing and also addressing the generation process. It is clear from the right-hand side of Figure 11, there are still six frequent main areas for research. The right-hand side of the figure extends the left side by adding more papers to the application categories.

More papers are dealing with the SPL application but with more details about the generation of test cases. Here, the papers considered the generation of test cases for the SPL as a special case of constrained interaction testing due to different constraints in the feature models used within SPL. In addition, researchers tried to integrate the generation algorithms with the testing tools. Some of those algorithms were depending mainly on constraint solving solutions. For example, Model-based Software Product Line Testing framework (MoSo-PoLiTe) [66] uses CSP solver for solving the constraints in the feature models of the SPLs. Other tools deployed well-known algorithms for constrained interaction testing to generate test cases for the SPLs. CITLAB tool [52] deployed IPO-family algorithms that are available in ACTS [91] tool to generate test cases for SPL. In fact, the publication in this direction aimed to increase the efficiency of the generation algorithms. Other researchers used special algorithms to reach this aim. For instance, Al-Hajjaji *et al.* [72] proposes an incremental approach for product sampling for pairwise interaction testing (called IncLing), which enables developers to generate samples on demand in a step-wise manner. Here, the performance of the generation algorithm is also improved by this incremental approach since there is no need to wait until the generation algorithm produces all the configurations. Lopez-Herrejon *et al.* [75] proposed Parallel Prioritized product line Genetic Solver (PPGS) algorithm, a parallel genetic algorithm for the generation of prioritized pairwise suites for SPLs. The study compared PPGS with the greedy algorithm prioritized-ICPL and in most cases, PPGS was found to be producing better results.

Marijan *et al.* [61] has also introduced an algorithm to generate a minimal set of valid test cases for SPL that covers 2-wise feature interactions for a given time interval. Mainly, the algorithm is based on constraint solver and constraint programming. The study identifies the generation of a minimal configuration set for SPL from the feature model as a constraint optimization problem. The algorithm can capture the relationships among the features in the input feature model by a constraint model that identify these relationships as constraints for the solvers. The algorithm then tries to generate a set that covers the $2 - wise$ interactions of the features and also satisfy the constraints.

#### 4) MODEL VALIDATION STUDIES

Model validation studies form another category of published research papers on constrained interaction testing. From Table 5, it is clear that seven published papers ( [21], [46], [54], [55], [116]–[120]) address this issue for constrained interaction testing. The papers in this category deal with input models of an application under test. These models are then used to generate test cases accordingly. The rationale behind this research direction is the deployment obstacles to find the correct set of parameters and values with the restrictions (i.e., constraints) among them for real-life applications.

Generally, in combinatorial interaction testing, a model is composed of parameters and values for each of them. Constrained interaction testing adds some complex entity to the model validation process which is the constraint model among the parameters. The test generation strategies start from this model to construct the final test suite. Having a good model for the system under test will improve the quality of the produced test suites by the strategy and also it could help in early defect discovery. The model will be more useful when we want to validate the produced test suites and assess their quality.

Calvagna and Gargantinia [55] presented a new approach to construct constrained $2 - wise$ test suites. More parts of the paper discuss the models of input domains and how the constraints should be presented in the strategy. The paper presented a model also to generate customized test suites by exclusion and inclusion of ad-hoc combinations of input parameters. The research also formalized constrained interaction testing as a logic problem. A model checker tool is presented in the paper to validate the model. In line with the approach of the paper, it presented a prototype tool for implementation. In another paper later, Calvagna and Gargantini [54] extended this approach to include more formalization of the model with more extensive evaluation process for the generation and validation process.

Segall *et al.* [118] then suggested a different approach to handle the complex relationship models between the parameters. The research suggested two different construction models. The first construction model is the type counter parameter, which is a special type of parameter that counts the number of specific values appearance for each input parameter in each test case. The second construction model is the type value property, which specifies the number of properties related to each parameter and values. The research showed that these two approaches will reduce the modeling complexity needed significantly. The research validated the two approaches on two real-life case studies. Although the approach is new in this direction, significant experimental results for validation could not be recognized.

Arcaini *et al.* [116] proposed a model validation approach for the constrained interaction testing, focusing more on the validation of the constraints rather than the interactions of parameters. The approach checks the consistency of the constraint among the input parameters, to be sure that they are not contradicted by each other, and hence the input parameters and their values are necessary for the inclusion in the test suite. This has been done by checking whether the produced test suites are consistent with the provided requirements in the beginning for the input parameters, values, and constraints among them. The research also provides a technique to identify potential causes for any appeared error and how to fix it. The research suggested four sets of checkpoints that each constrained interaction test suite must have. Those checkpoints are, the consistency of the constraints, the usefulness of the parameters and values, the correctness of the final test suite in term of constraints violation, and finally the importance of each test case in term of coverage. These checkpoints were validated by the benchmarks available in the CITLAB tool [49] and the constraints were validated with the help of SMT solver.

Tzoref-Brill and Maoz [119] explore the evolution of constrained interaction test space modeling. The research showed that the Boolean semantics are inadequate for constrained interaction model evolution. The research extends the Boolean semantics to a lattice-based semantics to provide consistency of the interpretation for model changes. This has been done via Galois connection to establish the connection among the elements in an abstract domain. The research provides an extensive formulation of the models without showing any experimental results.

In contrast with the approaches mentioned above of general model validation, Spichkova and Zamansky [120] proposes a formal framework for model and validation framework of constrained interaction testing at multiple levels of abstraction between elements. The framework is human-centric in which it provides queries to testers for helping to analyze and assess the quality of specific test plans, models, and constraints to check if they are complete and valid. The study presented a formal analysis of the framework without an extensive experimental evaluation.

While the model validation frameworks and approaches are feasible for constrained interaction testing, Khalsa and Labiche [117] introduced the base choice criteria to account for constraints. Specifically, the researchers discovered two extensions for the base choice to address complex constraints for complex systems. The criteria and models were evaluated on industrial and academic cases studies using different generated test suites by various testing tools. The evaluation was based on cost and effectiveness of the test suites in term of code coverage and fault detection.

### D. EXISTING GENERATION STRATEGIES, TOOLS AND TECHNIQUES (RQ4)

For the answering of RQ4, this study is interested in knowing available generation strategies and tools. By answering this question, it would also be possible to understand the features and drawbacks of each strategy. The generation process and the efficient algorithms of the strategies are also essential to show here. It is clear from discussions mentioned above that each generation strategy of constrained interaction testing needs a mechanism to deal with the constraints. In Section IV-C.1, it is illustrated that to solve the constraints, the researchers usually either using a constraint solver through constraint programming or they exclude the constraints. They may also avoid the inclusion of constraints by following a particular algorithm. Having discussed those constraint issues, this Section is more about to know the algorithms used to derive the interactions for the final constrained interaction test suite.

There are many generation approaches for normal combinatorial interaction testing, with few of them supporting the inclusion of constraints. From the examined papers, in general, there are three approaches for the generation. The first approach is to use a computational algorithm to construct and optimize the test suites; the second approach is to use a metaheuristic algorithm to optimize the test suite. With these two approaches, a constraint handling mechanism (explained in Section IV-C.1) is used to handle the constraints in the final test suite. The third approach is to use a constraint solver to construct the final test suite directly without violating the constraints. The presented generation methods in the considered studies are falling under one of these approaches.

Following the first approach, Yu *et al.* [91] presented ACTS tool for generating different instances of combinatorial interaction test suites including the support for constraint handling. In fact, ACTS is a composition of many algorithms, mainly the IPO-family algorithms like IPOG [124], IPOG-D [125], and IPOG-F [126]. To handle the constraints, the ACTS tool relies on the Choco[3] solver. The tool can handle large configurations with a large set of constraints and large values of interaction strength, especially in the command line script. As mentioned previously, the tool has been used as a base for many applications in the literature.

Garvin *et al.* [25] followed the second approach and presented an improved implementation of the original simulated annealing-based (SA) constrained test suite generator [26] called CASA. The tool includes the original implementation of the SA generator algorithm with some improvements. The improvement is concentrated in the long run time for highly configurable systems with large constraint support. The research reformulated the search algorithm in the SA to efficiently incorporate the constraints. The tool uses SAT solver to handle the constraints. Through the evaluation process in the study, the tools perform well for the 35 benchmarks used. However, the tool becomes slow, and the performance degrades as the configuration benchmarks become large and it is not able to generate test cases for some complex configurations. In addition, it could be noted that the tool does not scale well when the interaction strength grows. As mentioned earlier, the tool has been used in other studies to investigate the application of constrained interaction test suites for real-world problems.

Remaining with the second approach, Kalaee and Rafe [115] presented an algorithm to generate constrained interaction test suites using the PSO algorithm. The constraints are handled by an ROBDD graph that helps to prevent them appearing during the construction. In the same way, Alsariera *et al.* [70] presented an algorithm to generate the test suites using the Bat-inspired algorithm for SPL. The study handled the constraints by excluding them from the final test suite. Both studies need more extensive evaluation, and also their performance is not comparable with ACTS and CASA.

Following the third approach, Banbara *et al.* [113] presented an algorithm and a method to generate constrained interaction test suites using SAT solver directly. The study proposed modern CDCL SAT solvers with two encodings, one is order encoding while the other is a combination of order encoding. The use of these encodings helps to enhance the efficiency of generating better constrained interaction test suites in terms of size. The evaluation experiment showed this enhancement by generating better test suites sizes as compared with well-known results in the literature.

### E. USED BENCHMARKS FOR EVALUATION (RQ5)

By analyzing the considered papers for this study, it is clear that different studies are using standard benchmarks or what one can call ‘‘custom’’ benchmarks. As in the case of any benchmark, the aim of using it is to evaluate a particular approach within constrained interaction testing. In addition, it is also used to compare different approaches. Mainly, there are two types of benchmarks: benchmarks used to evaluate constrained interaction test generation efficiency in general and benchmarks used to evaluate the constrained interaction test generation efficiency for SPL.

Cohen *et al.* [26] used a set of experiments that have been used as a benchmark to evaluate a strategy for generating constrained interaction testing suites. Later in her study [23], Cohen has extended the set to include more real-world configurations generated from well-known software systems. These benchmarks are input models of Apache, GCC, Bugzilla, SPIN simulator and SPIN verifier. Apache HTTP Server[4] is an open source software system that works on different platforms to feed web services. GCC[5] is a multiple input language infrastructure for supporting many programming languages like C, C++, Java, and Fortran. Bugzilla[6] is a defect tracker from Mozilla that comes in open source to be used by software developers. SPIN model checker [127] is a software

[3] http://www.choco-solver.org/

[4] httpd.apache.org/docs/2.2/mod/quickreference.html

[5] http://gcc.gnu.org/onlinedocs/gcc-4.1.1

[6] https://bugzilla.readthedocs.org/en/latest

tool that is used as a case study for evaluation. In addition to these systems, 30 more system configurations were generated from the above systems to be used as benchmarks. These models are used as configuration benchmarks for 15 more published papers (see [22], [24]–[26], [36], [37], [54], [55], [68], [78], [79], [82], [85], [86], [90]). Other published papers rely on custom benchmarks, which are normally random configurations to simulate different scenarios of inputs.

Benchmarks for SPL test generation are commonly feature model files like XML files and contain different features and the constraints among them. These benchmarks may come from industry partners that use SPLs or may come from previously constructed sets for experimental use. In general, many papers use SPLOT,[7] which is a website for SPL tools and repositories. In fact, 8 of the considered studies in this paper have been using the features models available on SPLOT for benchmarks. These papers are: [44], [45], [58], [59], [62], [75], [99], [101].

### F. APPLICATIONS OF CONSTRAINED INTERACTION TESTING (RQ6)

As can be seen from Figure 11, there are mainly five areas in which the constrained interaction testing has frequently been used. In addition, there are 11 different areas where the constrained interaction testing has shown significant results concerning the application. Hence, in total, there are 16 areas of the application so far for constrained interaction testing. The following subsections discuss these application areas with more details.

#### 1) SPL

SPL engineering is getting more attention recently due to considerable industrial interest. The approach of SPL is to establish a platform for common products in a product line by identifying variability and commonality of features during the development of individual products [128]. The testing foundation in the SPL is to produce test assets that can be reused by the products during the process of product line development. The test case includes entities that form common parts related to a variety of possible products that must be realized by the domain engineer. The testing process of the SPL is getting more and more attention in the last decade. Many approaches have been published for testing SPL. da Mota Silveira Neto *et al.* [10] summarized different testing approaches for SPL in an extensive systematic literature study.

Due to huge features of modern products, it is very hard to generate a complete and feasible test suite and configurations. Here, combinatorial interaction testing can be a useful approach to construct an optimal and practical configuration for different potential products from a large set of features. However, in reality, there are different constraints among these features that tend to produce infeasible product configurations. Constrained interaction testing can be an excellent solution for this problem by generating an optimal configuration set without violating the constraints among the features.

To apply the constrained interaction testing for SPL, a set of activities need to be followed, as in the 29 papers identified earlier in Figure 11. These activities are the feature model adaptation including the constraints identification, interaction testing adaptation, and configuration set generation.

Feature model adaptation and constraint identification probably is the starting point in any approach. This step starts by taking a standard input file that can also be represented in a feature diagram. For instance, Perrouin *et al.* [57] formalized different types of potential constraints from the feature models. The study also proposed a toolset to generate the final test suite for SPL. Calvagna *et al.* [52] presented a more extensive and mature method to translate the feature models into combinatorial interaction models in a framework named CITLAB. The framework gives many advantages for a tool to generate constrained interaction testing such as editing facilities, seeding, and generation algorithms.

Interaction testing adaptation is an important stage to use the constrained interaction testing for SPL. In fact, the majority of the published papers are converting the feature models to the CA mathematical object using the combinatorial model obtained by conversion. As previously mentioned, the CA consist of a set of $t - wise$ sub-array, where $t$ indicates the strength of feature combination. The t is the number of features that we want to test in combination.

Because the number of possible features to be combined grows exponentially with the number of features designed for a particular SPL, there is a need for an efficient and practical strategy for $t - wise$ generation to get the most optimum set of combinations within an affordable testing cost. The adaptation takes care of four main entities for the CA which are input parameters, number of features, the constraints among the features and the combination strength. This stage is tied to the configuration set generation stage because the CA is an outcome of the stage. Here, different studies have proposed various algorithms for the generation. Majority of the studies used well-known algorithms for constrained interaction testing like ACTS, CASA, IPO-family algorithms and AETG. For example, Matnei Filho and Vergilio [101] developed a strategy that relies on the AETG for multi-objective test data generation for feature models' mutation testing. Following the same approach, Lamancha and Usaola [104] proposed a strategy to generate $2 - wise$ test products to cover all feature in the feature model using AETG. Johansen *et al.* [66] proposed a strategy for generating $t - wise$ test suites for SPLs from large feature models using CASA algorithms.

On the other hand, other studies proposed specialized tools for generating constrained interaction test suites for SPL. For example, PLEDGE is an editor and test generation tool developed by Henard *et al.* [59] for SPL that rely on special algorithms based on SAT solver. PACOGEN [61], [81] is another tool for generating $2 - wise$ test suites for SPL using constraint programming. Researchers are also using heuristic algorithms to generate optimum test suites. For example,

[7] http://www.splot-research.org/

Alsariera *et al.* [70] developed a strategy called SPLBA that relies on the bat-inspired algorithm to generate constrained interaction test suites. The strategy depends on constraint exclusion to resolve the constraints among the feature models. Jia *et al.* [31] developed a Hyperheuristic strategy to generate test suites for SPL. The strategy depends on SAT solver to resolve the constraints.

### 2) FAULT DETECTION AND CHARACTERIZATION

Here, the straight-forward aim is to construct test cases for possible fault detection. The characterization process is used within fault detection for drawing a better understanding of the faults in the system and to enable fault avoidance in the future development. Another purpose is to design test cases for regression testing purposes. It could also be used for test case prioritization. Fault detection has been examined in the literature with normal interaction testing without constraints. For example, Yilmaz *et al.* [129] uses normal $t - wise$ approach with covering array to generate test cases to detect faults. The approach is integrated in the Skoll system [130] to allow parallel execution of test cases across a grid of computers. The results of the executed test cases were returned to a central server. The central server then classified the faults to cover and uncover faults. This classification of faults during the detection process helped the developers for providing descriptions of failing configurations to find the causes of the faults. In fact, this method can be used even with other test generation methods; however, Yilmaz found that interaction testing can produce better classification models.

### 3) TEST SELECTION

With the recent growing complexity of applications, it has appeared that constrained interaction testing is a more practical testing approach. The research in this direction considers the complex and large number of inputs and generating test cases from these inputs. Taking which values of these inputs and how to take them is a primary question here. Constrained interaction testing can sample these inputs systematically and take care of the constraints available between input values during the generation of the test cases. As previously mentioned, the test selection applications are more assessed within the normal combinatorial interaction testing. More recently, the focus of research has also shifted to consider the constraints among the input values due to the large and complex input variables and complex structures (e.g., JSON) in the real-world systems. For large scale modern software systems like commercial applications, selecting input for a test case is a difficult task due to the large input domain. In this case, exhaustive testing usually is impractical and not possible. One possible solution is to select one test design method (like boundary-value analysis or cause and effect) to select the inputs for test cases. However, these test design methods are not applicable for every situation. Constrained interaction testing could form another test design method that is suitable for sampling test cases based on their interactions among them while also considering the constraints among them.

Zhong *et al.* [108] generated constrained interaction test suites for large and complex software systems using comKorat, which is a combination of Korat and ACTS algorithms. Four systems were used to test the effectiveness of the generated test suites. These four systems are large-scale software systems developed at eBay and Yahoo!. The approach detected almost 59 bugs that were not detected by other approaches.

Nakornburi and Suwannasart [107] use a different approach to select entities necessary for test generation like the input parameters and values and then identify the constraints among them. The approach depends on the statistical profile of the software's user for selecting the entities. By selecting those entities, the study proposes a flow to generate an optimal $2 - wise$ test suite for the software under test by eliminating the unrealistic combinations from the final test suite. However, the study did not mention the algorithm for the generation, and the presented approach is just for resolving constraints by excluding them. The study shows preliminary evaluation results for small size software under testing, and there is still need to show an extensive experiment.

For the case of large and complex software under test, Yilmaz [97] introduced a new combinatorial interaction object called test case-aware covering array, which is a refinement to the original covering array mathematical object used traditionally with constrained interaction testing but with different coverage criteria of the interactions. The study selects the input configurations of the software under test and considers a set of test cases for these algorithms to satisfy all test case-specific constraints. The study presented three algorithms, two algorithms for test generation and minimization and the other algorithm is dedicated to minimize the number of test runs. The evaluation experiments used two highly configurable software systems (Apache and MySQL). The results of the evaluation showed that the test case-aware covering array is practically better than the traditional covering array due to the consideration of handling real-world and test case-specific constraints.

Finally, Kruse *et al.* [96] presented in a short paper an approach to handle constraints in the numerical constrained interaction test suites using the classification tree method. The study showed a prototype for the implementation without giving a complete description of the final solution.

### 4) SECURITY

As previously mentioned, two studies were focusing on the application of constrained interactions for security testing. These studies are by Bozic *et al.* [93] and Bouquet *et al.* [102].

Bozic *et al.* [93] assessed the concept of constrained interaction for web security testing using the IPO-family algorithm. Specifically, the author evaluated IPOG and IPOG-F algorithms which are available within the ACTS tool. The algorithms generate test cases for exploring and detecting injection attacks which are remote exploits that can lead to security breaches. Specifically, the study focused on the cross-site scripting (XSS) vulnerabilities detection. Here, the inputs have been modeled using XSS attack vectors' modeling method to identify the number of inputs. The model is also used to identify the relations and constraints among the input parameter values. Four test sets are generated using ACTS tool, each two of these test sets were generated for IPOG and IPOG-F respectively. Then, these generated test suites are applied on a set of web applications that are used as cases studies. These applications are included in Open Web Application Security Project (OWASP)[8] and in the Exploit Database Project.[9] In general, the results of the study indicate that constrained interaction test suite can significantly reveal security leaks in web applications. The consideration of constraints during the modeling process of attack grammars will increase the number of test cases that can detect more vulnerabilities by examining more security breaches. The study also investigated that the quality of the test suites generated by IPOG-F is slightly better as compared to IPOG.

Bouquet *et al.* [102] discussed constrained interaction testing with model-based testing used for security testing. Although the paper is a general study without specific experiments, many important issues are discussed regarding the security test generation using model-based approach. In fact, the paper illustrated and discussed many cases studies in security where constrained interaction testing could be applied. Such case studies can very well form a stable base for experimental investigations.

### 5) GUI TESTING

Constrained interaction testing principles have been used in its simple form for GUI testing. Recently, the functional testing of GUIs has shifted to an advanced process by using graph theory and modeling concepts. The basic idea is to convert all events and positions on the GUI into nodes and edges and consider all possible sequences of events. This model has been used later as an input to the generation algorithms to generate test suites that simulate the event sequences. In fact, this method is initiated and solved by normal combinatorial interaction testing. However, with the normal combinatorial interaction testing, it has appeared that many of the generated test cases were not valid since the sequences were not visible for execution on the actual application's GUI. Huang *et al.* [32] recognized and illustrated this situation. Huang *et al.* investigated that there are constraints for the events and some events may not be available for execution. The study proposes a method to repair the generated test suites by avoiding those invalid combination of events and generated new feasible test cases, which in other words, solve the constraints. To evolve new test cases from the repaired test suites, a genetic algorithm is used to utilized event flow graph (EFG) for generating new test cases. The approach is tested using different synthetic programs that contain different GUIs with different constraints among the events. Although the use of EFG has been investigated in other research, it is not complete, and there is a need for manual verification in different cases.

Go *et al.* [106] introduced a different approach to apply $2 - wise$ constrained interaction testing to GUI testing by analyzing system specifications. The analysis process of the specification will identify the class partitions of the input data to be tested. Here, they categorized the constraints into data, semantic rules, and GUI constraints. The data constraints are identified from the system inputs; semantic rules constraints are identified from system specification documents; while the GUI constraints can be collected from initial GUI requirements. The constraints have been solved by using conventional equivalent class partitioning without utilizing any constraint solvers. Here, the study did not mention the $2-wise$ test suite generation algorithm since the application used is not complex, and the test suite can be generated even by conventional methods. The study evaluated the approach on a real-world problem. The approach showed significant results concerning fault detection as compared to the conventional random testing approach.

Finally, Bauersfeld *et al.* [114], tried to complement and build the approach started by Huang *et al.* [32] for identifying and generating test sequences for applications with GUI. The study treated the test sequence generation of GUIs as an optimization problem and employed ant colony optimization (ACO) algorithm to solve it optimally. The study tries to avoid the limitations of EFG use by using a new metric called MCT (Maximum Call Tree) and avoids the inclusion of those invalid interactions from the beginning thus preventing repairing process of the test suites later. The approach is tested using an implemented Java SWT environment for application testing. A graphical editor is used to test the effectiveness of the approach. The study showed that the approach can generate better sequences as compared with the random approach. The study claimed that the approach is better than the use of EFG; however, there is no evidence for this claim.

### 6) OTHER APPLICATIONS

In addition to the applications mentioned above of constrained interaction testing, other applications have been referred to in one published paper. As in the case of above applications, the test cases were generated either using well-known tools or by developing specific algorithms for the generation. In fact, the application domains are broad in range.

For instance, Sherwood [92] assessed the application of constrained interaction testing for embedded functions and anticipated benefits of programming languages. Here, PHP language is used for its flexibility and prevalence. An individual model is employed in the study to identify and validate the constraints and generate test cases. In the same way, Salecker and Glesner [112] applied a similar approach for grammar-based test selection and generation.

[8] https://www.owasp.org/index.php/OWASP

[9] https://www.exploit-db.com/

Li *et al.* [94] applied the constrained interaction testing to big data applications. Here, two real-world big data ETL applications are used. The ETL (Extract, Transform, and Load) applications are common type applications in big data employed by the developers to report and analyze data by writing scripts in Hive, SQL, or Pig. To generate the test cases for the SUT from the input domain models, the study used ACTS tool. The study proved the effectiveness of the constrained interaction testing by using a minimized sizes of the test suites but detecting all the faults found in the original data source. Fischer *et al.* [73] performed an empirical case study to explore the application of constrained interaction testing. Seven Java and C programs are used as subjects of the case study in which they composed over nine million lines of codes in total. SAT solver is used to generate and resolve the constraints in the test suites.

Palacios *et al.* [98] addressed the use of constrained interaction testing for testing Service Level Agreements (SLAs). In this study, the classical combinatorial interaction testing has been combined with the Classification Tree Method to generate test suites and resolve input constraints. For this generation process, the study presented an automated tool called SLACT (SLA Combinatorial Testing) that has been applied successfully to an eHealth case study.

Arrieta *et al.* [100] used constrained interaction testing to generate configurations for cyber-physical systems (CPSs). CPSs are systems to integrate physical processes with digital technologies. In the study, simulation-based test cases were generated to test different configurations of CPSs. To resolve the constraints, SAT solver is used with other approaches for the generation to form a tool called ASTERYSCO. The tools are used successfully with a cases study for configurable Unmanned Aerial Vehicle.

Calvagna *et al.* [30] assessed the application of constrained interaction testing in the context of random and combinatorial effectiveness. The study uses this approach to know the differences between random and combinatorial concerning effectiveness for conformance testing. Here, the conformance testing used to verify components of Java Virtual Machine (JVM). To generate test cases model checking is applied to the considered specification.

Zhong *et al.* [108] applied constrained interaction testing on four large-scale software applications. The software systems were developed by eBay and Yahoo companies. To generate the test cases, the study presented a newly developed tool called comKorat which is an integration between Korat and ACTS generation tools. By applying the test suites generated by comKorat, more faults were detected in the SUT. Specifically, 59 new faults were detected while other test suites did not detect them.

In another study, Grieskamp *et al.* [33] investigated and proved the application of constrained interaction testing for path coverage of software systems. To generate the test suites, the study used SMT solver. The solver is also sued to resolve the constants among the paths in the software.

Finally, Kalaee and Rafe [115] used the graph theory and Particle Swarm Optimization (PSO) to generate constrained interaction test suites. The developed generation algorithm is applied to a case study for a boiler system.

### G. CURRENT LIMITATIONS AND CHALLENGES (RQ7)

Fundamentally, the limitations and challenges in the application of constrained interaction test generation in practice related to four main factors: the parameters (P), the values (v), the interaction strength ($t$) as well as the constraints (C).

In line with the advancement of new technologies such as the Internet of Things, big data analytic as well as cloud computing, the intertwined dependencies and constraints between components and their sub-systems can be massive. To put this issue into perspective, Microsoft Windows source code was merely around 4.5 million LOC in 1993. In 2003, after ten years, Microsoft Windows source code increased to over 50 million LOC [131], a tremendous increase. Here, the growth of parameters ($P$) and the values ($v$) and the constraints (C) can be problematic (although such effect many be controllable as far as ($t$) is concerned as empirical evidence suggest that most faults are detectable at $t = 6$) [132]–[134]).

Two potential stumbling blocks can be attributed to the above issues. Firstly, as the parameters ($P$), values ($v$) and constraints (C) increase, the coverage of tuples for consideration also increases tremendously. Putting constraints (C) aside for simplicity, consider a scenario when $P = 1000$, $v = 5$ and $t = 2$. Here, the search space for exhaustive testing is $5^{1000}$ and the search tuple is equal to 1.

$$\binom{p}{t} v^t = \frac{p!}{t!(p-t)} v^t = \frac{1000!}{2!(1000-2)} 5^2 \tag{1}$$

Similarly, when $v$ is large (even with small $P$ and $t$), the search space can also be large. Consider another scenario when $v = 1000$, $P = 2$, and $t = 2$. In this case, the search space for exhaustive testing is $1000^5$ and the search tuple is equal to 2

$$\binom{p}{t} v^t = \frac{p!}{t!(p-t)} v^t = \frac{2!}{2!(2-2)} 1000^2 \tag{2}$$

Here, if both $P$ and $v$ are large so do the required search space and the required tuples (even if t is bounded at $t = 6$). To this end, no known strategy can handle both large parameters ($P$) and large values ($v$) with rich constraints (C). Currently, there is a limit on the size of the search space as well as the number of tuples to be processed owing to limited hardware and computational resources, rendering the need for much research as far as scalability is concerned. Concerning SPL as a special case of constrained interaction testing, while the parameter ($P$) can be large with rich constraints (C), the values ($v$) are always limited to two Boolean (true or false).

The second stumbling block relates to the process of finding and specifying constraints for large parameters and values. The use of feature diagram and constraints solver appears widespread for SPL; however, such an approach has not been sufficiently adopted for the general constrained interaction

**TABLE 6.** List of studied papers.

| ID# | Ref. | Paper Title | Year |
|---|---|---|---|
| 1 | [52] | Combinatorial Testing for Feature Models Using CITLAB | 2013 |
| 2 | [76] | Test Algebra for Combinatorial Testing | 2013 |
| 3 | [74] | Multi-Objective Optimal Test Suite Computation for Software Product Line Pairwise Testing | 2013 |
| 4 | [116] | Validation of Models and Tests for Constrained Combinatorial Interaction Testing | 2014 |
| 5 | [5] | Practical Combinatorial Interaction Testing: Empirical Findings on Efficiency and Early Fault Detection | 2015 |
| 6 | [78] | Optimization of Combinatorial Testing by Incremental SAT Solving | 2015 |
| 7 | [92] | Embedded Functions in Combinatorial Test Designs | 2015 |
| 8 | [93] | Evaluation of the IPO-Family Algorithms for Test Case Generation in Web Security Testing | 2015 |
| 9 | [82] | Constraint Handling In Combinatorial Test Generation Using Forbidden Tuples | 2015 |
| 10 | [56] | Flattening or Not of the Combinatorial Interaction Testing Models? | 2015 |
| 11 | [31] | Learning Combinatorial Interaction Test Generation Strategies using Hyperheuristic Search | 2015 |
| 12 | [60] | Evaluating Reconfiguration Impact in Self-adaptive Systems | 2015 |
| 13 | [70] | SPLBA: An Interaction Strategy for Testing Software Product Lines Using the Bat-Inspired Algorithm | 2015 |
| 14 | [34] | TCA: An Efficient Two-Mode Meta-Heuristic Algorithm for Combinatorial Test Generation | 2015 |
| 15 | [117] | An extension of category partition testing for highly constrained systems | 2016 |
| 16 | [83] | Test oracles and test script generation in combinatorial testing | 2016 |
| 17 | [111] | Effective test generation for combinatorial decision coverage | 2016 |
| 18 | [79] | Greedy combinatorial test case generation using unsatisfiable cores | 2016 |
| 19 | [94] | Applying Combinatorial Test Data Generation to Big Data Applications | 2016 |
| 20 | [80] | Test effectiveness evaluation of prioritized combinatorial testing: a case study | 2016 |
| 21 | [84] | A tool for constrained pairwise test case generation using statistical user profile based prioritization | 2016 |
| 22 | [73] | A source level empirical study of features and their interactions in variable software | 2016 |
| 23 | [22] | Exploiting constraint solving history to construct interaction test suites | 2007 |
| 24 | [23] | Constructing interaction test Suites for highly-configurable systems in the presence of constraints: a greedy approach | 2008 |
| 25 | [24] | An improved meta-heuristic search for constrained interaction testing | 2009 |
| 26 | [85] | Improved extremal optimization for constrained pairwise testing | 2009 |
| 27 | [95] | Generating minimal fault detecting test suites for boolean expressions | 2010 |
| 28 | [57] | Automated and scalable t-wise test case generation strategies for software product lines | 2010 |
| 29 | [32] | Repairing GUI test suites using a genetic algorithm | 2010 |
| 30 | [86] | Calculating prioritized interaction test sets with constraints using binary decision diagrams | 2011 |
| 31 | [67] | Input-input relationship constraints in T-way testing | 2011 |
| 32 | [48] | CITLAB: A laboratory for combinatorial interaction testing | 2012 |
| 33 | [118] | Simplified modeling of combinatorial test spaces | 2012 |
| 34 | [112] | Combinatorial interaction testing for test selection in grammar-based testing | 2012 |
| 35 | [96] | Numerical constraints for combinatorial interaction testing | 2012 |
| 36 | [97] | Test case-aware combinatorial interaction testing | 2013 |
| 37 | [49] | Combinatorial interaction testing with CITLAB | 2013 |
| 38 | [35] | Cascade: A test generation tool for combinatorial testing | 2013 |
| 39 | [87] | PROW: A Pairwise algorithm with constRaints, Order and Weight | 2015 |
| 40 | [36] | Generating combinatorial test suite using combinatorial optimization | 2014 |
| 41 | [98] | Automatic test case generation for WS-Agreements using combinatorial testing | 2015 |
| 42 | [19] | Prioritized interaction testing for pair-wise coverage with seeding and constraints | 2006 |
| 43 | [68] | Design and implementation of a harmony-search-based variable-strength t-way testing strategy with constraints support | 2012 |
| 44 | [81] | Practical minimization of pairwise-covering test configurations using constraint programming | 2016 |
| 45 | [41] | Effective product-line testing using similarity-based product prioritization | 2016 |
| 46 | [20] | Constraint Models for the Covering Test Problem | 2006 |
| 47 | [25] | Evaluating improvements to a meta-heuristic search for constrained interaction testing | 2011 |
| 48 | [54] | A Formal Logic Approach to Constrained Combinatorial Testing | 2010 |
| 49 | [77] | Locating and detecting arrays for interaction faults | 2008 |
| 50 | [44] | Pairwise testing for software product lines: comparison of two approaches | 2012 |
| 51 | [42] | Model-based pairwise testing for feature interaction coverage in software product line engineering | 2012 |
| 52 | [99] | Testing variability-intensive systems using automated analysis: an application to Android | 2016 |
| 53 | [100] | Automatic generation of test system instances for configurable cyber-physical systems | 2016 |
| 54 | [101] | A multi-objective test data generation approach for mutation testing of feature models | 2016 |
| 55 | [58] | Mutation-Based Generation of Software Product Line Test Configurations | 2014 |
| 56 | [102] | Model-Based Testing for Functional and Security Test Generation | 2014 |
| 57 | [50] | Efficient Combinatorial Test Generation Based on Multivalued Decision Diagrams | 2014 |

**TABLE 6.** ***(Continued.)* List of studied papers.**

| | | | |
|---|---|---|---|
| 58 | [119] | Lattice-Based Semantics for Combinatorial Model Evolution | 2015 |
| 59 | [88] | Graph Methods for Generating Test Cases with Universal and Existential Constraints | 2015 |
| 60 | [103] | The comKorat Tool: Unified Combinatorial and Constraint-Based Generation of Structurally Complex Tests | 2016 |
| 61 | [37] | Generating Covering Arrays with Pseudo-Boolean Constraint Solving and Balancing Heuristic | 2016 |
| 62 | [55] | A Logic-Based Approach to Combinatorial Testing with Constraints | 2008 |
| 63 | [89] | Covering Arrays Avoiding Forbidden Edges | 2008 |
| 64 | [38] | Finding Orthogonal Arrays Using Satisfiability Checkers and Symmetry Breaking Constraints | 2008 |
| 65 | [51] | Combining Satisfiability Solving and Heuristics to Constrained Combinatorial Interaction Testing | 2009 |
| 66 | [33] | Interaction Coverage Meets Path Coverage by SMT Constraint Solving | 2009 |
| 67 | [45] | Automated Incremental Pairwise Testing of Software Product Lines | 2010 |
| 68 | [27] | Improving the Testing and Testability of Software Product Lines | 2010 |
| 69 | [113] | Generating Combinatorial Test Cases by Efficient SAT Encodings Suitable for CDCL SAT Solvers | 2010 |
| 70 | [104] | Testing Product Generation in Software Product Lines Using Pairwise for Features Coverage | 2010 |
| 71 | [114] | A Metaheuristic Approach to Test Sequence Generation for Applications with a GUI | 2011 |
| 72 | [65] | Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible | 2011 |
| 73 | [39] | Faulty Interaction Identification via Constraint Solving and Optimization | 2012 |
| 74 | [69] | Constraints Dependent T-Way Test Suite Generation Using Harmony Search Strategy | 2012 |
| 75 | [63] | Generating Better Partial Covering Arrays by Modeling Weights on Sub-product Lines | 2012 |
| 76 | [64] | A Technique for Agile and Automatic Interaction Testing for Product Lines | 2012 |
| 77 | [71] | Coverage Criteria for Behavioural Testing of Software Product Lines | 2014 |
| 78 | [18] | Constraint-Based Approaches to the Covering Test Problem | 2005 |
| 79 | [43] | Similarity-based Prioritization in Software Product-line Testing | 2014 |
| 80 | [66] | An Algorithm for Generating t-wise Covering Arrays from Large Feature Models | 2012 |
| 81 | [59] | PLEDGE: A Product Line Editor and Test Generation Tool | 2013 |
| 82 | [26] | Interaction Testing of Highly-Configurable Systems in the Presence of Constraints | 2007 |
| 83 | [28] | Incremental Covering Array Failure Characterization in Large Configuration Spaces | 2009 |
| 84 | [61] | Practical Pairwise Testing for Software Product Lines | 2013 |
| 85 | [40] | Constraint Solving Techniques for Software Testing and Analysis | 2010 |
| 86 | [75] | A Parallel Evolutionary Algorithm for Prioritized Pairwise Testing of Software Product Lines | 2014 |
| 87 | [30] | Random Versus Combinatorial Effectiveness in Software Conformance Testing: A Case Study | 2015 |
| 88 | [29] | Efficiency and Early Fault Detection with Lower and Higher Strength Combinatorial Interaction Testing | 2013 |
| 89 | [105] | Generating an Automated Test Suite by Variable Strength Combinatorial Testing for Web Services | 2016 |
| 90 | [90] | Constraint Test Cases Generation Based on Particle Swarm Optimization | 2016 |
| 91 | [120] | A Human-centred Framework for Combinatorial Test Design | 2016 |
| 92 | [106] | Pairwise testing for systems with data derived from real-valued variable inputs | 2016 |
| 93 | [107] | Constrained Pairwise Test Case Generation Approach based on Statistical User Profile | 2016 |
| 94 | [115] | An Optimal Solution for Test Case Generation Using ROBDD Graph and PSO Algorithm | 2016 |
| 95 | [72] | IncLing: Efficient Product-Line Testing using Incremental Pairwise Sampling | 2016 |
| 96 | [108] | Combinatorial Generation of Structurally Complex Test Inputs for Commercial Software Applications | 2016 |
| 97 | [91] | ACTS: A Combinatorial Test Generation Tool | 2013 |
| 98 | [62] | PACOGEN: Automatic Generation of Pairwise Test Configurations From Feature Models | 2011 |
| 99 | [110] | Reducing combinatorics in testing product lines | 2011 |
| 100 | [109] | Eliminating products to test in a software product line | 2010 |
| 101 | [21] | Coverage and adequacy in software product line testing | 2006 |
| 102 | [46] | Model-based coverage-driven test suite generation for software product lines | 2011 |
| 103 | [47] | Pairwise feature-interaction testing for SPLs: potentials and limitations | 2011 |

testing. If general constrained interaction testing is to be adopted for large parameters ($P$) and large values ($v$) with rich constraints (C), existing strategies need to integrate and embed constraints solver (or constraints programming) as part of their implementations.

### *H. POSSIBLE RESEARCH DIRECTIONS FOR FUTURE (RQ8)*

Constrained interaction testing can be formulated as an optimization problem, resulting in the adoption of meta-heuristic based strategies in the literature. Meta-heuristic based strategies offer superior solutions (i.e., concerning test suite size) compared to existing approaches (i.e., owing to the systematic exploration and exploitation of the search space). Although useful, much-existing meta-heuristic based strategies have explored single meta-heuristic algorithm. As such, the exploration and exploitation of existing strategies have been limited based on the (local and global) search operators derived from a particular meta-heuristic algorithm. In this case, choosing a proper combination of search operators (termed as hybridization) can be the key to achieving good performance (as hybridization can capitalize on the strengths and address the deficiencies of each algorithm collectively and synergistically).

Hyper-heuristics have recently received considerable attention to addressing some of the hybridization as mentioned earlier issues [135], [136]. Specifically, hyper-heuristic represents an approach of using (meta)-heuristics to choose (meta)-heuristics to solve the optimization problem

**TABLE 7. List of active journals with abbreviations.**

| Acronym | Journal Full Name |
|---|---|
| IEEE T Software Eng | IEEE Transactions on Software Engineering |
| J Syst Software | The Journal of Systems and Software |
| Eur J Combin | European Journal of Combinatorics |
| Comp Stand Inter | Computer Standards & Interfaces journal |
| Inform Software Tech | Information and Software Technology Journal |
| Softw Syst Model | Software & Systems Modeling Journal |
| Constraints | Constraints Journal |
| Empir Softw Eng | Empirical Software Engineering journal |
| J Autom Reasoning | Journal of Automated Reasoning |
| J Comb Optim | Journal of Combinatorial Optimization |
| Software Qual J | Software Quality Journal |
| J Softw Eng Res Dev | Journal of Software Engineering Research and Development |
| J Compu Inform Tech | Journal of Computing and Information Technology |
| Softw Pract Exp | Software: Practice and Experience journal |
| Qual Reliab Eng Int | Quality and Reliability Engineering International Journal |

in hand. Unlike traditional meta-heuristics, which directly operates on the solution space, hyper-heuristics offer flexible integration and adaptive manipulation of the complete (low-level) meta-heuristics or merely partial adoption of a particular meta-heuristic search operator through non-domain feedback. In this manner, the hyper-heuristic can evolve its heuristic selection and acceptance mechanism in the search for a high-quality solution.

Apart from adopting hybridization with hyper-heuristic, the integration with machine learning appears to be a viable approach to improve the state-of-the-art of existing meta-heuristic algorithms for constrained interaction testing [137]. Machine learning relates to the study of fundamental laws that govern the computer learning process (i.e., concerning how to build systems that can automatically learn from experience). Machine learning techniques can be classified into three types: supervised, unsupervised, and reinforcement. Supervised learning involves learning the direct functional input-output mapping based on some set of training data and being able to perform a prediction on new data (e.g., deep learning approaches). Unlike supervised learning, unsupervised learning does not require precise training data. Specifically, unsupervised learning involves learning by drawing inferences (e.g., clustering) on the input datasets. Reinforcement learning relates to learning that allows mapping between states and actions to maximize reward signal by experimental discovery. This type of learning differs from supervised learning by the fact that it relies upon punish and reward mechanism and never corrects pairs of input-output data (even when dealing with sub-optimal responses).

To maximize the effectiveness of the fault finding efficiency, the constrained interaction testing can also be made as a multi-objective problem. Apart from avoiding forbidden tuples and maximizing the tuples coverage to obtain the most minimum test suite size, one possibility is to include minimizing the similarity among the test cases within the test suite. Here, the effectiveness of similarity distance metrics such as Euclidean distance, Hamming distance, Manhattan distance, Cosine similarity, Jaccard index and its variants can be investigated further. With a more diverse set of test cases, the fault detection rate may increase but potentially at the expense of a slight increase of test suite size. Additionally, from a different perspective, the same approach of using similarity metrics can also be used to prioritize existing test suite.

Finally, to leverage on the need for supporting large parameters (P), values (v), with rich constraints (C), there is a need to explore the new hybrid strategy based on cloud-based service-oriented architecture. In this manner, the user can exploit the computing power of the cloud to generate the constrained test suite. Additionally, the user can also explore the generation of constrained test suite as a service without having to purchase the actual tool.

## V. THREATS TO VALIDITY

Like any other literature study, this study has threats to its validity. Many threats were eliminated by following well-known recommendations and advice on conducting literature studies.

First, as can be seen in the paper, several search strings were tried to assure maximum coverage of related papers. Although most of the related works have been selected here, 100% coverage of the papers cannot be guaranteed. There could be some uncovered papers by the search string; however, to overcome this threat a pilot search for several papers and also snowball sampling search was conducted.

Second, systematic literature studies could suffer from single-author bias during data extraction. To eliminate this threat, a double-checking and reviewing process by all authors individually were conducted. Besides, automatic mining and extraction tools were also tried in the spreadsheet to verify the results of the data extraction process.

Third, several papers during the study were excluded. The selection criteria are discussed explicitly in SectionIII-C. Some studies may also get excluded due to the scope of the paper itself. This study is dedicated to combinatorial interaction testing with constraints. Hence, those papers that cover combinatorial interaction testing without constraint support are excluded. The rationale behind this selection is that several studies cover these papers (see [2], [7], [138]). In addition, the constraint support in combinatorial testing

**TABLE 8. List of active conferences with abbreviations.**

| Acronym | Conference Full Name |
|---|---|
| ICST | International Conference on Software Testing, Verification and Validation Workshops |
| ASE | International Conference on Automated Software Engineering |
| SSBSE | International Symposium on Search Based Software Engineering |
| FSE | International Symposium on Foundations of Software Engineering |
| SPLC | International Conference on Software Product Lines |
| ISSTA | International Symposium on Software Testing and Analysis |
| QRS | International Conference on Software Quality, Reliability and Security Companion |
| ICTSS | IFIP International Conference on Testing Software and Systems |
| ICSE | International Conference on Software Engineering |
| PRICAI | Pacific Rim International Conference on Artificial Intelligence |
| TAP | International Conference on Tests and Proofs |
| MODELS | International Conference on Model Driven Engineering Languages and Systems |

was covered because it is an important aspect for the practical application of combinatorial interaction testing. The papers not published electronically were excluded; however, any contribution in the area of constrained interaction testing that is not published electronically is not expected.

## VI. CONCLUSION

In this paper, an extensive literature study of 103 research papers published on constrained interaction testing between 2005 and 2016 is presented. In the study, the selected papers were analyzed from different perspectives based on a set of research questions. The results indicate that there is an increasing trend to address constrained interactions for the testing purpose. There is a mix of contributions in conferences, workshops, and journals with the majority of papers being published in conferences. The active authors and research groups in the field are further highlighted. Based on the analysis, the contributions in constrained interaction testing are grouped into four categories of constrained test generation studies, application studies, generation and application studies and model validation studies. The study found that to solve constraints, the researchers usually either use a constraint solver or exclude the constraints. The study further showed that the applications of constrained interaction testing are within software product lines, fault detection and characterization, test selection, security and GUI testing, among others. The study ends with a discussion of limitations, challenges, and areas for future research for constrained interaction testing.

## APPENDIX A

See Table 6.

## APPENDIX B

See Table 7.

## APPENDIX C

See Table 8.

## REFERENCES

[1] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," *Inf. Softw. Technol.*, vol. 51, no. 6, pp. 957–976, 2009.

[2] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surv.*, vol. 43, no. 2, pp. 1–29, 2011.

[3] R. E. Lopez-Herrejon, S. Fischer, R. Ramler, and A. Egyed, "A first systematic mapping study on combinatorial interaction testing for software product lines," in *Proc. IEEE 8th Int. Conf. Softw. Test., Verification Validation Workshops (ICSTW)*, Apr. 2015, pp. 1–10.

[4] B. S. Ahmed, L. M. Gambardella, W. Afzal, and K. Z. Zamli, "Handling constraints in combinatorial interaction testing in the presence of multi objective particle swarm and multithreading," *Inf. Softw. Technol.*, vol. 86, pp. 20–36, Jun. 2017.

[5] J. Petke, M. B. Cohen, M. Harman, and S. Yoo, "Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection," *IEEE Trans. Softw. Eng.*, vol. 41, no. 9, pp. 901–924, Sep. 2015.

[6] J. Petke, "Constraints: The future of combinatorial interaction testing," in *Proc. IEEE/ACM 8th Int. Workshop Search-Based Softw. Test. (SBST)*, May 2015, pp. 17–18.

[7] V. Kuliamin and A. Petukhov, "A survey of methods for constructing covering arrays," *Programm. Comput. Softw.*, vol. 37, no. 3, pp. 121–146, 2011.

[8] B. S. Ahmed and K. Z. Zamli, "A review of covering arrays and their application to software testing," *J. Comput. Sci.*, vol. 7, no. 9, pp. 1375–1385, 2011.

[9] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test case generation," *IEEE Trans. Softw. Eng.*, vol. 36, no. 6, pp. 742–762, Nov. 2010.

[10] P. A. da Mota Silveira Neto, I. do Carmo Machado, J. D. McGregor, E. S. de Almeida, and S. R. de Lemos Meira, "A systematic mapping study of software product lines testing," *Inf. Softw. Technol.*, vol. 53, no. 5, pp. 407–423, 2011.

[11] S. Zein, N. Salleh, and J. Grundy, "A systematic mapping study of mobile application testing techniques," *J. Syst. Softw.*, vol. 117, pp. 334–356, Jul. 2016.

[12] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," Keele Univ., Keele, U.K., Tech. Rep. EBSE-2007-01, 2007.

[13] T. Dyba, T. Dingsoyr, and G. K. Hanssen, "Applying systematic reviews to diverse study types: An experience report," in *Proc. 1st Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, 2007, pp. 225–234.

[14] K. Petersen, S. Vakkalanka, and L. Kuzniarz, "Guidelines for conducting systematic mapping studies in software engineering: An update," *Inf. Softw. Technol.*, vol. 64, pp. 1–18, 2015.

[15] N. S. R. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman, "Identification and management of technical debt: A systematic mapping study," *Inf. Softw. Technol.*, vol. 70, pp. 100–121, Feb. 2016.

[16] C. V. C. de Magalhães, F. Q. B. da Silva, R. E. S. Santos, and M. Suassuna, "Investigations about replication of empirical studies in software engineering: A systematic mapping study," *Inf. Softw. Technol.*, vol. 64, pp. 76–101, Aug. 2015.

[17] M. B. Cohen, "Designing test suites for software interaction testing," Ph.D. dissertation, Dept. Comput. Sci., Univ. Auckland, Auckland, New Zealand, 2004.

[18] B. Hnich, S. Prestwich, and E. Selensky, *Constraint-Based Approaches to the Covering Test Problem*. Berlin, Germany: Springer, 2005, pp. 172–186.

[19] R. C. Bryce and C. J. Colbourn, "Prioritized interaction testing for pairwise coverage with seeding and constraints," *Inf. Softw. Technol.*, vol. 48, no. 10, pp. 960–970, 2006.

[20] B. Hnich, S. D. Prestwich, E. Selensky, and B. M. Smith, "Constraint models for the covering test problem," *Constraints*, vol. 11, no. 2, pp. 199–219, 2006.

[21] M. B. Cohen, M. B. Dwyer, and J. Shi, "Coverage and adequacy in software product line testing," in *Proc. ACM*, 2006, pp. 53–63.

[22] M. B. Cohen, M. B. Dwyer, and J. Shi, "Exploiting constraint solving history to construct interaction test suites," in *Proc. Test., Acad. Ind. Conf. Pract. Res. Techn.-MUTATION (TAICPART-MUTATION)*, 2007, pp. 121–132.

[23] M. B. Cohen, M. B. Dwyer, and J. Shi, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach," *IEEE Trans. Softw. Eng.*, vol. 34, no. 5, pp. 633–650, Sep. 2008.

[24] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "An improved meta-heuristic search for constrained interaction testing," in *Proc. 1st Int. Symp. Search Based Softw. Eng.*, 2009, pp. 13–22.

[25] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "Evaluating improvements to a meta-heuristic search for constrained interaction testing," *Empirical Softw. Eng.*, vol. 16, no. 1, pp. 61–102, 2011.

[26] M. B. Cohen, M. B. Dwyer, and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, 2007, pp. 129–139.

[27] I. Cabral, M. B. Cohen, and G. Rothermel, "Improving the testing and testability of software product lines," in *Proc. 14th Int. Conf. Softw. Product Lines, Going Beyond (SPLC)*, 2010, pp. 241–255.

[28] S. Fouché, M. B. Cohen, and A. Porter, "Incremental covering array failure characterization in large configuration spaces," in *Proc. 18th Int. Symp. Softw. Test. Anal. (ISSTA)*, 2009, pp. 177–188.

[29] J. Petke, S. Yoo, M. B. Cohen, and M. Harman, "Efficiency and early fault detection with lower and higher strength combinatorial interaction testing," in *Proc. 9th Joint Meet. Found. Softw. Eng. (ESEC/FSE)*, 2013, pp. 26–36.

[30] A. Calvagna, A. Fornaia, and E. Tramontana, "Random versus combinatorial effectiveness in software conformance testing: A case study," in *Proc. 30th Annu. ACM Symp. Appl. Comput. (SAC)*, 2015, pp. 1797–1802.

[31] Y. Jia, M. B. Cohen, M. Harman, and J. Petke, "Learning combinatorial interaction test generation strategies using hyperheuristic search," in *Proc. Learn. Combinat. Interact. Test Generat. Strategies Using Hyperheuristic Search*, 2015, pp. 540–550.

[32] S. Huang, M. B. Cohen, and A. M. Memon, "Repairing gui test suites using a genetic algorithm," in *Proc. 3rd IEEE Int. Conf. Softw. Test., Verification Validation*, Apr. 2010, pp. 245–254.

[33] W. Grieskamp, X. Qu, X. Wei, N. Kicillof, and M. B. Cohen, *Interaction Coverage Meets Path Coverage by SMT Constraint Solving*. Berlin, Germany: Springer, 2009, pp. 97–112.

[34] J. Lin, C. Luo, S. Cai, K. Su, D. Hao, and L. Zhang, "TCA: An efficient two-mode meta-heuristic algorithm for combinatorial test generation (T)," in *Proc. 30th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, Nov. 2015, pp. 494–505.

[35] Y. Zhao, Z. Zhang, J. Yan, and J. Zhang, "Cascade: A test generation tool for combinatorial testing," in *Proc. IEEE 6th Int. Conf. Softw. Test., Verification Validation Workshops*, Mar. 2013, pp. 267–270.

[36] Z. Zhang, J. Yan, Y. Zhao, and J. Zhang, "Generating combinatorial test suite using combinatorial optimization," *J. Syst. Softw.*, vol. 98, pp. 191–207, Dec. 2014.

[37] H. Liu, F. Ma, and J. Zhang, *Generating Covering Arrays With Pseudo-Boolean Constraint Solving and Balancing Heuristic*. Cham, Switzerland: Springer, 2016, pp. 262–270.

[38] F. Ma and J. Zhang, *Finding Orthogonal Arrays Using Satisfiability Checkers and Symmetry Breaking Constraints*. Berlin, Germany: Springer, 2008, pp. 247–259.

[39] J. Zhang, F. Ma, and Z. Zhang, *Faulty Interaction Identification Via Constraint Solving and Optimization*. Berlin, Germany: Springer, 2012, pp. 186–199.

[40] F. Ma, "Constraint solving techniques for software testing and analysis," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng. (ICSE)*, vol. 2. May 2010, pp. 417–420.

[41] M. Al-Hajjaji, T. Thüm, M. Lochau, J. Meinicke, and G. Saake, "Effective product-line testing using similarity-based product prioritization," in *Software & Systems Modeling*. Berlin, Germany: Springer, 2016, pp. 1–23.

[42] M. Lochau, S. Oster, U. Goltz, and A. Schürr, "Model-based pairwise testing for feature interaction coverage in software product line engineering," *Softw. Quality J.*, vol. 20, no. 3, pp. 567–604, 2012.

[43] M. Al-Hajjaji, T. Thüm, J. Meinicke, M. Lochau, and G. Saake, "Similarity-based prioritization in software product-line testing," in *Proc. 18th Int. Softw. Product Line Conf. (SPLC)*, vol. 1. 2014, pp. 197–206.

[44] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. le Traon, "Pairwise testing for software product lines: Comparison of two approaches," *Softw. Quality J.*, vol. 20, no. 3, pp. 605–643, 2012.

[45] S. Oster, F. Markert, and P. Ritter, *Automated Incremental Pairwise Testing of Software Product Lines*. Berlin, Germany: Springer, 2010, pp. 196–210.

[46] H. Cichos, S. Oster, M. Lochau, and A. Schürr, *Model-Based Coverage-Driven Test Suite Generation for Software Product Lines*. Berlin, Germany: Springer, 2011, pp. 425–439.

[47] S. Oster, M. Zink, M. Lochau, and M. Grechanik, *Pairwise Feature-Interaction Testing for SPLs: Potentials and Limitations*. Munich, Germany: ACM, 2011.

[48] A. Gargantini and P. Vavassori, "CITLAB: A laboratory for combinatorial interaction testing," in *Proc. IEEE 5th Int. Conf. Softw. Test., Verification Validation*, Apr. 2012, pp. 559–568.

[49] A. Calvagna, A. Gargantini, and P. Vavassori, "Combinatorial interaction testing with citlab," in *Proc. IEEE 6th Int. Conf. Softw. Test., Verification Validation*, Apr. 2013, pp. 376–382.

[50] A. Gargantini and P. Vavassori, *Efficient Combinatorial Test Generation Based on Multivalued Decision Diagrams*. Cham, Switzerland: Springer, 2014, pp. 220–235.

[51] A. Calvagna and A. Gargantini, *Combining Satisfiability Solving and Heuristics to Constrained Combinatorial Interaction Testing*. Berlin, Germany: Springer, 2009, pp. 27–42.

[52] A. Calvagna, A. Gargantini, and P. Vavassori, "Combinatorial testing for feature models using CitLab," in *Proc. IEEE 6th Int. Conf. Softw. Test., Verification Validation Workshops*, Mar. 2013, pp. 338–347.

[53] A. Gargantini and G. Fraser, "Generating minimal fault detecting test suites for general Boolean specifications," *Inf. Softw. Technol.*, vol. 53, no. 11, pp. 1263–1273, 2011.

[54] A. Calvagna and A. Gargantini, "A formal logic approach to constrained combinatorial testing," *J. Automated Reason.*, vol. 45, no. 4, pp. 331–358, 2010.

[55] A. Calvagna and A. Gargantini, "A logic-based approach to combinatorial testing with constraints," in *Proc. 2nd Int. Conf. Tests Proofs (TAP)*, 2008, pp. 66–83.

[56] C. Henard, M. Papadakis, and Y. L. Traon, "Flattening or not of the combinatorial interaction testing models?" in *Proc. IEEE 8th Int. Conf. Softw. Test., Verification Validation Workshops (ICSTW)*, Apr. 2015, pp. 1–4.

[57] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. le Traon, "Automated and scalable T-wise test case generation strategies for software product lines," in *Proc. 3rd Int. Conf. Softw. Test., Verification Validation (ICST)*, Paris, France, Apr. 2010, pp. 459–468.

[58] C. Henard, M. Papadakis, and Y. Le Traon, *Mutation-Based Generation of Software Product Line Test Configurations*. Cham: Springer, 2014, pp. 92–106.

[59] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon, "PLEDGE: A product line editor and test generation tool," in *Proc. 17th Int. Softw. Product Line Conf. Co-Located Workshops (SPLC)*, 2013, pp. 126–129.

[60] S. Sen, S. Di Alesio, D. Marijan, and A. Sarkar, "Evaluating reconfiguration impact in self-adaptive systems—An approach based on combinatorial interaction testing," in *Proc. 41st Euromicro Conf. Softw. Eng. Adv. Appl.*, 2015, pp. 250–254.

[61] D. Marijan, A. Gotlieb, S. Sen, and A. Hervieu, "Practical pairwise testing for software product lines," in *Proc. 17th Int. Softw. Product Line Conf. (SPLC)*, 2013, pp. 227–235.

[62] A. Hervieu, B. Baudry, and A. Gotlieb, "PACOGEN: Automatic generation of pairwise test configurations from feature models," in *Proc. IEEE 22nd Int. Symp. Softw. Rel. Eng. (ISSRE)*, Nov. 2011, pp. 120–129.

[63] M. F. Johansen, Ø. Y. Haugen, F. Fleurey, A. G. Eldegard, and T. Syversen, *Generating Better Partial Covering Arrays by Modeling Weights on Sub-Product Lines*. Berlin, Germany: Springer, 2012, pp. 269–284.

[64] M. F. Johansen, Ø. Y. Haugen, F. Fleurey, E. Carlson, J. Endresen, and T. Wien, *A Technique for Agile and Automatic Interaction Testing for Product Lines*. Berlin, Germany: Springer, 2012, pp. 39–54.

[65] M. F. Johansen, Ø. Y. Haugen, and F. Fleurey, *Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible*. Berlin, Germany: Springer, 2011, pp. 638–652.

[66] M. F. Johansen, Ø. Y. Haugen, and F. Fleurey, "An algorithm for generating T-wise covering arrays from large feature models," in *Proc. 16th Int. Softw. Product Line Conf. (SPLC)*, vol. 1. 2012, pp. 46–55.

[67] R. R. Othman and K. Z. Zamli, "Input-input relationship constraints in T-way testing," in *Proc. IEEE Symp. Ind. Electron. Appl.*, Sep. 2011, pp. 527–531.

[68] A. R. A. Alsewari and K. Z. Zamli, "Design and implementation of a harmony-search-based variable-strength T-way testing strategy with constraints support," *Inf. Softw. Technol.*, vol. 54, no. 6, pp. 553–568, 2012.

[69] A. A. Al-Sewari and K. Z. Zamli, *Constraints Dependent T-Way Test Suite Generation Using Harmony Search Strategy*. Berlin, Germany: Springer, 2012, pp. 1–11.

[70] Y. A. Alsariera, M. A. Majid, and K. Z. Zamli, "SPLBA: An interaction strategy for testing software product lines using the bat-inspired algorithm," in *Proc. 4th Int. Conf. Softw. Eng. Comput. Syst. (ICSECS)*, 2015, pp. 148–153.

[71] X. Devroey, G. Perrouin, A. Legay, M. Cordy, P.-Y. Schobbens, and P. Heymans, *Coverage Criteria for Behavioural Testing of Software Product Lines*. Berlin, Germany: Springer, 2014, pp. 336–350.

[72] M. Al-Hajjaji, S. Krieter, T. Thüm, M. Lochau, and G. Saake, "IncLing: Efficient product-line testing using incremental pairwise sampling," in *Proc. ACM SIGPLAN Int. Conf. Generat. Programm., Concepts Exper. (GPCE)*, 2016, pp. 144–155.

[73] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "A source level empirical study of features and their interactions in variable software," in *Proc. IEEE 16th Int. Workshop Conf. Source Code Anal. Manipulation (SCAM)*, Oct. 2016, pp. 197–206.

[74] R. E. Lopez-Herrejon, F. Chicano, J. Ferrer, A. Egyed, and E. Alba, "Multi-objective optimal test suite computation for software product line pairwise testing," in *Proc. ICSM*, 2013, pp. 404–407.

[75] R. E. Lopez-Herrejon, J. J. Ferrer, F. Chicano, E. N. Haslinger, A. Egyed, and E. Alba, "A parallel evolutionary algorithm for prioritized pairwise testing of software product lines," in *Proc. Annu. Conf. Genet. Evol. Comput. (GECCO)*, 2014, pp. 1255–1262.

[76] W.-T. Tsai, C. J. Colbourn, J. Luo, G. Qi, Q. Li, and X. Bai, "Test algebra for combinatorial testing," in *Proc. 8th Int. Workshop Autom. Softw. Test (AST)*, 2013, pp. 19–25.

[77] C. J. Colbourn and D. W. McClary, "Locating and detecting arrays for interaction faults," *J. Combinat. Optim.*, vol. 15, no. 1, pp. 17–48, 2008.

[78] A. Yamada, T. Kitamura, C. Artho, E. H. Choi, Y. Oiwa, and A. Biere, "Optimization of combinatorial testing by incremental sat solving," in *Proc. IEEE 8th Int. Conf. Softw. Test., Verification Validation (ICST)*, Apr. 2015, pp. 1–10.

[79] A. Yamada, A. Biere, C. Artho, T. Kitamura, and E.-H. Choi, "Greedy combinatorial test case generation using unsatisfiable cores," in *Proc. 31st IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, Sep. 2016, pp. 614–624.

[80] E.-H. Choi, S. Kawabata, O. Mizuno, C. Artho, and T. Kitamura, "Test effectiveness evaluation of prioritized combinatorial testing: A case study," in *Proc. IEEE Int. Conf. Softw. Quality, Rel. Secur. (QRS)*, Aug. 2016, pp. 61–68.

[81] A. Hervieu, D. Marijan, A. Gotlieb, and B. Baudry, "Practical minimization of pairwise-covering test configurations using constraint programming," *Inf. Softw. Technol.*, vol. 71, pp. 129–146, Mar. 2016.

[82] L. Yu, F. Duan, Y. Lei, R. N. Kacker, and D. R. Kuhn, "Constraint handling in combinatorial test generation using forbidden tuples," in *Proc. IEEE 8th Int. Conf. Softw. Test., Verification Validation Workshops (ICSTW)*, Apr. 2015, pp. 1–9.

[83] P. M. Kruse, "Test oracles and test script generation in combinatorial testing," in *Proc. IEEE 9th Int. Conf. Softw. Test., Verification Validation Workshops (ICSTW)*, Apr. 2016, pp. 75–82.

[84] S. Nakornburi and T. Suwannasart, "A tool for constrained pairwise test case generation using statistical user profile based prioritization," in *Proc. 13th Int. Joint Conf. Comput. Sci. Softw. Eng. (JCSSE)*, Jul. 2016, pp. 1–6.

[85] J. Yuan, C. Jiang, and Z. Jiang, "Improved extremal optimization for constrained pairwise testing," in *Proc. Int. Conf. Res. Challenges Comput. Sci.*, 2009, pp. 108–111.

[86] E. Salecker, R. Reicherdt, and S. Glesner, "Calculating prioritized interaction test sets with constraints using binary decision diagrams," in *Proc. IEEE 4th Int. Conf. Softw. Test., Verification Validation Workshops (ICSTW)*, Mar. 2011, pp. 278–285.

[87] B. P. Lamancha, M. Polo, and M. Piattini, "PROW: A pairwise algorithm with constraints, order and weight," *J. Syst. Softw.*, vol. 99, pp. 1–19, Jan. 2015.

[88] S. Hallé, E. La Chance, and S. Gaboury, *Graph Methods for Generating Test Cases With Universal and Existential Constraints*. Cham, Switzerland: Springer, 2015, pp. 55–70.

[89] P. Danziger, E. Mendelsohn, L. Moura, and B. Stevens, "Covering arrays avoiding forbidden edges," in *Proc. Int. Conf. Combinat. Optim. Appl.*, 2009, pp. 298–308.

[90] Y. Sheng, C. Wei, G. Wang, S. Jiang, and Y. Chen, "Constraint test cases generation based on particle swarm optimization," in *Proc. 22nd ISSAT Int. Conf. Rel. Quality Design*, 2016, pp. 329–333.

[91] R. N. Kacker, D. R. Kuhn, Y. Lei, and J. F. Lawrence, "Combinatorial testing for software: An adaptation of design of experiments," *Measurement*, vol. 46, no. 9, pp. 3745–3752, 2013.

[92] G. B. Sherwood, "Embedded functions in combinatorial test designs," in *Proc. IEEE 8th Int. Conf. Softw. Test., Verification Validation Workshops (ICSTW)*, Apr. 2015, pp. 1–10.

[93] J. Bozic, B. Garn, D. E. Simos, and F. Wotawa, "Evaluation of the IPO-family algorithms for test case generation in Web security testing," in *Proc. IEEE 8th Int. Conf. Softw. Test., Verification Validation Workshops (ICSTW)*, Apr. 2015, pp. 1–10.

[94] N. Li, Y. Lei, H. R. Khan, J. Liu, and Y. Guo, "Applying combinatorial test data generation to big data applications," in *Proc. 31st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, Aug. 2016, pp. 637–647.

[95] G. Fraser and A. Gargantini, "Generating minimal fault detecting test suites for Boolean expressions," in *Proc. 3rd Int. Conf. Softw. Test., Verification, Validation Workshops*, 2010, pp. 37–45.

[96] P. M. Kruse, J. Bauer, and J. Wegener, "Numerical constraints for combinatorial interaction testing," in *Proc. IEEE 5th Int. Conf. Softw. Test., Verification Validation*, Apr. 2012, pp. 758–763.

[97] C. Yilmaz, "Test case-aware combinatorial interaction testing," *IEEE Trans. Softw. Eng.*, vol. 39, no. 5, pp. 684–706, May 2013.

[98] M. Palacios, J. García-Fanjul, J. Tuya, and G. Spanoudakis, "Automatic test case generation for ws-agreements using combinatorial testing," *Comput. Standards Inter.*, vol. 38, pp. 84–100, Feb. 2015.

[99] J. A. Galindo, H. Turner, D. Benavides, and J. White, "Testing variability-intensive systems using automated analysis: An application to Android," *Softw. Quality J.*, vol. 24, no. 2, pp. 365–405, 2016.

[100] A. Arrieta, G. Sagardui, L. Etxeberria, and J. Zander, "Automatic generation of test system instances for configurable cyber-physical systems," *Softw. Quality J.*, vol. 25, no. 3, pp. 1041–1083, 2016.

[101] R. A. M. Filho and S. R. Vergilio, "A multi-objective test data generation approach for mutation testing of feature models," *J. Softw. Eng. Res. Develop.*, vol. 4, no. 1, p. 4, 2016.

[102] F. Bouquet, F. Peureux, and F. Ambert, *Model-Based Testing for Functional and Security Test Generation*. Cham, Switzerland: Springer, 2014, pp. 1–33.

[103] H. Zhong, L. Zhang, and S. Khurshid, *The comKorat Tool: Unified Combinatorial and Constraint-Based Generation of Structurally Complex Tests*. Cham, Germany: Springer, 2016, pp. 107–113.

[104] B. P. Lamancha and M. P. Usaola, *Testing Product Generation in Software Product Lines Using Pairwise for Features Coverage*. Berlin, Germany: Springer, 2010.

[105] Y. Li, Z.-A. Sun, and J.-Y. Fang, "Generating an automated test suite by variable strength combinatorial testing for Web services," *J. Comput. Inf. Technol.*, vol. 24, no. 3, pp. 271–282, 2016.

[106] K. Go, S. Kang, J. Baik, and M. Kim, "Pairwise testing for systems with data derived from real-valued variable inputs," *Softw.-Pract. Exper.*, vol. 46, no. 3, pp. 381–403, 2016.

[107] S. Nakornburi and T. Suwannasart, "Constrained pairwise test case generation approach based on statistical user profile," in *Proc. Lect. Notes Eng. Comput. Sci.*, vol. 1. 2016, pp. 445–448.

[108] H. Zhong, L. Zhang, and S. Khurshid, "Combinatorial generation of structurally complex test inputs for commercial software applications," in *Proc. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2016, pp. 981–986.

[109] C. H. P. Kim, D. Batory, and S. Khurshid, "Eliminating products to test in a software product line," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, 2010, pp. 139–142.

[110] C. H. P. Kim, D. S. Batory, and S. Khurshid, "Reducing combinatorics in testing product lines," in *Proc. 10th Int. Conf. Aspect-Oriented Softw. Develop. (AOSD)*, 2011, pp. 57–68.

[111] R. Gao, L. Hu, W. E. Wong, H. L. Lu, and S. K. Huang, "Effective test generation for combinatorial decision coverage," in *Proc. IEEE Int. Conf. Softw. Quality, Rel. Secur. Companion (QRS-C)*, Aug. 2016, pp. 47–54.

[112] E. Salecker and S. Glesner, "Combinatorial interaction testing for test selection in grammar-based testing," in *Proc. IEEE 5th Int. Conf. Softw. Test., Verification Validation (ICST)*, Apr. 2012, pp. 610–619.

[113] M. Banbara, H. Matsunaka, N. Tamura, and K. Inoue, *Generating Combinatorial Test Cases by Efficient SAT Encodings Suitable for CDCL SAT Solvers*. Berlin, Germany: Springer, 2010, pp. 112–126.

[114] S. Bauersfeld, S. Wappler, and J. Wegener, *A Metaheuristic Approach to Test Sequence Generation for Applications With a GUI*. Berlin, Germany: Springer, 2011, pp. 173–187.

[115] A. Kalaee and V. Rafe, "An optimal solution for test case generation using ROBDD graph and PSO algorithm," *Quality Rel. Eng. Int.*, vol. 32, no. 7, pp. 2263–2279, 2016.

[116] P. Arcaini, A. Gargantini, and P. Vavassori, "Validation of models and tests for constrained combinatorial interaction testing," in *Proc. IEEE 7th Int. Conf. Softw. Test., Verification Validation Workshops*, Mar. 2014, pp. 98–107.

[117] S. K. Khalsa and Y. Labiche, "An extension of category partition testing for highly constrained systems," in *Proc. IEEE 17th Int. Symp. High Assurance Syst. Eng. (HASE)*, Jan. 2016, pp. 47–54.

[118] I. Segall, R. Tzoref-Brill, and A. Zlotnick, "Simplified modeling of combinatorial test spaces," in *Proc. IEEE 5th Int. Conf. Softw. Test., Verification Validation*, Apr. 2012, pp. 573–579.

[119] R. Tzoref-Brill and S. Maoz, *Lattice-Based Semantics for Combinatorial Model Evolution*. Cham, Switzerland: Springer, 2015, pp. 276–292.

[120] M. Spichkova and A. Zamansky, "A human-centred framework for combinatorial test design," in *Proc. 11th Int. Conf. Eval. Novel Softw. Approaches Softw. Eng. (ENASE)*, Apr. 2016, pp. 228–233.

[121] K. C. Tai and Y. Lie, "In-parameter-order: A test generation strategy for pairwise testing," in *Proc. 3rd IEEE Int. Symp. High-Assurance Syst. Eng.*, Nov. 1998, pp. 254–261.

[122] J. Bozic, D. E. Simos, and F. Wotawa, "Attack pattern-based combinatorial testing," in *Proc. 9th Int. Workshop Autom. Softw. Test (AST)*, 2014, pp. 1–7.

[123] B. Garn, I. Kapsalis, D. E. Simos, and S. Winkler, "On the applicability of combinatorial testing to Web application security testing: A case study," in *Proc. Workshop Joining AcadeMiA Ind. Contrib. Test Autom. Model-Based Test. (JAMAICA)*, 2014, pp. 16–21.

[124] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG: A general strategy for T-way software testing," in *Proc. 4th Annu. IEEE Int. Conf. Workshops Eng. Comput.-Based Syst.*, Mar. 2007, pp. 549–556.

[125] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG-IPOG-D: Efficient test generation for multi-way combinatorial testing," *Softw. Test. Verification Rel.*, vol. 18, no. 3, pp. 125–148, Sep. 2008.

[126] M. Forbes, J. Lawrence, Y. Lei, R. N. Kacker, and D. R. Kuhn, "Refining the in-parameter-order strategy for constructing covering arrays," *J. Res. Nat. Inst. Standards Technol.*, vol. 113, no. 5, pp. 287–297, 2008.

[127] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, May 1997.

[128] J. Lee, S. Kang, and D. Lee, "A survey on software product line testing," in *Proc. 16th Int. Softw. Product Line Conf. (SPLC)*, vol. 1. 2012, pp. 31–40.

[129] C. Yilmaz, M. B. Cohen, and A. A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," *IEEE Trans. Softw. Eng.*, vol. 32, no. 1, pp. 20–34, Jan. 2006.

[130] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. Schmidt, and B. Natarajan, "Skoll: Distributed continuous quality assurance," in *Proc. 26th Int. Conf. Softw. Eng.*, May 2004, pp. 459–468.

[131] A. Deshpande and D. Riehle, *The Total Growth of Open Source*. Boston, MA, USA: Springer, 2008, pp. 197–209.

[132] D. R. Kuhn and M. J. Reilly, "An investigation of the applicability of design of experiments to software testing," in *Proc. 27th Annu. NASA Goddard Softw. Eng. Workshop (SEW)*, 2002, pp. 91–95.

[133] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, "Software fault interactions and implications for software testing," *IEEE Trans. Softw. Eng.*, vol. 30, no. 6, pp. 418–421, Jun. 2004.

[134] R. C. Bryce and C. J. Colbourn, "Test prioritization for pairwise interaction coverage," in *Proc. 1st Int. Workshop Adv. Model-Based Test. (A-MOST)*, 2005, pp. 1–7.

[135] E. K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, and J. R. Woodward, *A Classification of Hyper-Heuristic Approaches*. Boston, MA, USA: Springer, 2010, pp. 449–468.

[136] E. K. Burke, G. Kendall, and E. Soubeiga, "A tabu-search hyperheuristic for timetabling and rostering," *J. Heuristics*, vol. 9, no. 6, pp. 451–470, Dec. 2003.

[137] K. Z. Zamli, F. Din, G. Kendall, and B. S. Ahmed, "An experimental study of hyper-heuristic selection and acceptance mechanism for combinatorial T-way test suite generation," *Inf. Sci.*, vol. 399, pp. 121–153, Aug. 2017.

[138] S. K. Khalsa and Y. Labiche, "An orchestrated survey of available algorithms and tools for combinatorial testing," in *Proc. IEEE 25th Int. Symp. Softw. Rel. Eng. (ISSRE)*, Nov. 2014, pp. 323–334.

**BESTOUN S. AHMED** received the B.Sc. degree in electrical and electronic engineering from the University of Salahaddin-Erbil in 2004, the M.Sc. degree from University Putra Malaysia in 2009, and the Ph.D. degree in software engineering from University Sains Malaysia (USM) in 2012. He was a Research Fellow with the Software Engineering Research Group, USM. He was as a Senior Lecturer with Salahaddin University. He spent one year doing his post-doctoral research in the Swiss AI Laboratory, Istituto Dalle Molle di Studi sull'Intelligenza Artificiale, Switzerland. He is currently an Assistant Professor with the Department of Computer Science and the Co-Founder of the Software Testing Intelligent Laboratory, Czech Technical University in Prague. His primary research interest includes software testing, search-based software testing, and applied soft computing. He serves as a reviewer and an editorial member for many international journals and an organizing committee member of many international conferences.

**KAMAL Z. ZAMLI** (M'17) received the B.Sc. degree in electrical engineering from the Worcester Polytechnic Institute, USA, in 1992, the M.Sc. degree in real-time software engineering from Universiti Teknologi Malaysia in 2000, and the Ph.D. degree in software engineering from Newcastle University, Newcastle upon Tyne, U.K., in 2003. He is currently the Dean and a Professor with the Faculty of Computer Systems and Software Engineering, University Malaysia Pahang. His main research interests include search-based software engineering, combinatorial software testing, and computational intelligence. He is a member of MySEIG.

**WASIF AFZAL** received the Ph.D. degree in software engineering from the Blekinge Institute of Technology. He is currently a Senior Lecturer with the Software Testing Laboratory, Mälardalen University. His research interests include software testing, empirical software engineering, and decision-support tools for software verification and validation.

**MIROSLAV BURES** received the Ph.D. degree from the Faculty of Electrical Engineering, Czech Technical University in Prague. He is currently a Researcher and a Senior Lecturer in software testing and quality assurance with the Faculty of Electrical Engineering, Czech Technical University in Prague. His research interests include model-based testing (process and workflow testing and data consistency testing) efficiency of test automation (test automation architectures, assessment of automated testability, and economic aspects) and quality assurance methods for the Internet of Things solutions, reflecting specifics of this technology. In these areas, he also leads several research and development and experimental projects. He is a member of the Czech Chapter of the ACM, CaSTB, and ISTQB Academia Workgroup and participates in broad activities in the professional testing community.

# 11 Appendix E: Identification of Potential Reusable Subroutines in Recorded Automated Test Scripts

[A.5] Miroslav Bures, Martin Filipsky, and Ivan Jelinek. Identification of Potential Reusable Subroutines in Recorded Automated Test Scripts. *International Journal of Software Engineering and Knowledge Engineering*, 28(01), pages 3-36. 2018. (Q4, IF 0.3)

International Journal of Software Engineering
and Knowledge Engineering
Vol. 28, No. 1 (2018) 3–36
© World Scientific Publishing Company
DOI: 10.1142/S0218194018500018

# Identification of Potential Reusable Subroutines in Recorded Automated Test Scripts

Miroslav Bures*, Martin Filipsky† and Ivan Jelinek‡

*Department of Computer Science*
*Czech Technical University in Prague, Karlovo namesti 13*
*Prague 121 35, Czech Republic*
**miroslav.bures@fel.cvut.cz*
*†filipma2@fel.cvut.cz*
*‡jelinek@fel.cvut.cz*

Received 10 October 2016
Revised 22 November 2016
Accepted 23 June 2017

In the automated testing based on actions in user interface of the tested application, one of the key challenges is maintenance of these tests. The maintenance overhead can be decreased by suitably structuring the test scripts, typically by employing reusable objects. To aid in the development, maintenance and refactoring of these test scripts, potentially reusable objects can be identified by a semi-automated process. In this paper, we propose a solution that identifies the potentially reusable objects in a set of automated test scripts and then provides developers with suggestions about these objects. During this process, we analyze the semantics of specific test steps using a system of abstract signatures. The solution can be used to identify the potentially reusable objects in both recorded automated test sets and tests programmed in an unstructured style. Moreover, compared to approaches that are based solely on searching for repetitive source code fragments, the proposed system identifies potentially reusable objects that are more relevant for test automation.

*Keywords*: Software testing; test automation; automated test recording; test set optimization; test code refactoring.

## 1. Introduction

Test automation represents one possible strategy for making the testing process more efficient. When implemented well, automated tests provide several benefits compared to manual testing: they can be automatically executed repeatedly on multiple platforms, which lowers test execution costs; they are more precise in comparison to tests performed by manual testers; and they can run in non-productive time slots in the software development cycle (typically at night). Nevertheless, automated testing also has its tradeoffs. The effort required to prepare automated tests is usually higher

[*] Corresponding author.

than that required to prepare manual test cases. Precisely defined sequences of steps and expected results tend to be brittle and are affected whenever the System Under Test (SUT) changes. In comparison to human testers, who use their intelligence to overcome inconsistencies between test cases and the SUT, such inconsistencies usually lead to interruptions in automated test flow situations (and can also lead to false error reports). Thus, when automated test scripts are designed sub-optimally or if the SUT changes frequently, they need relatively costly maintenance. From our observations [1] and those of others [2–5], costly test script maintenance is considered as one of the biggest challenges in this domain. This issue persists regardless of the test organization model used, which generally influences the test automation process [6]. Moreover, the maintenance problem is significant — especially for automated testing based on simulating a user's action in the Front-end (FE) user interface of the SUT, which is the scope of this paper.

Currently, two major strategies are used for creating automated test scripts that exercise SUT FE elements:

(i) Record and Replay, in which the test automation tool saves the steps the test designer performs in the SUT FE, and
(ii) Descriptive Programming, in which the test designer writes the code for the automated test script.

In practice, a combination of both variants can be used. Initially, we can record the flow of a user's actions in the SUT FE. Then, to finalize the test scripts, we can extend their code by adding more logic, for example, by adding assertions of the expected test results. Generally, a test recording approach is viewed as a fast automation style, but one that is inefficient from a test maintenance viewpoint [7–9].

From the maintenance viewpoint, it is difficult to assess what an optimum between the record and replay and the descriptive programming approaches should be because many factors influence the result: the structure of the automated test scripts, the structure and testability of the SUT, the scope of the tests and many others. The common generalization is that recording produces brittle automated tests but requires relatively low investment, whereas descriptive programming reduces script brittleness through various techniques (e.g. employing reusable objects, adding more intelligence to SUT FE element locators, using extra configurable layer for elements identification, etc.), but at the cost of a higher initial effort. However, descriptive programming can also result in higher script stability and, thus, lower maintenance costs [7, 10]. We have confirmed this effect through practical observations of many test automation projects.

From a technical point of view, record and replay or conducting descriptive programming in a naive, unstructured manner usually leads to scripts that contain a potentially high ratio of repeating fragments (an example is given in Fig. 1). These code-fragment duplications are potentially problematic and can lead to increased maintenance overhead and script brittleness. Reducing the duplicated code in automated

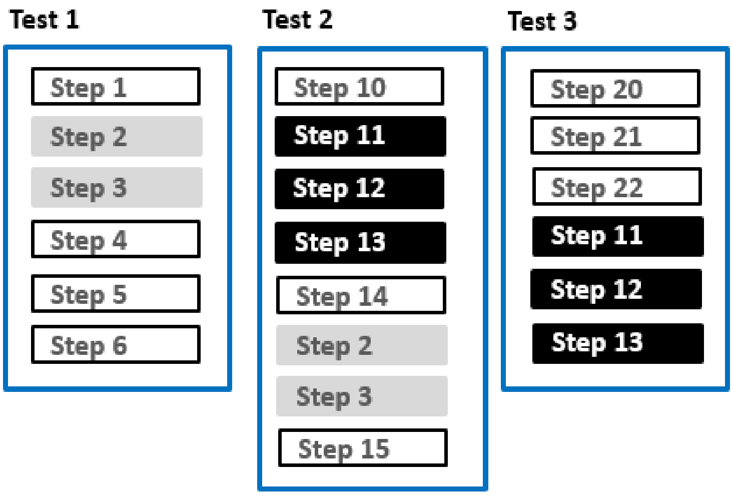


Fig. 1. An example of test scripts with repeating fragments (potentially reusable code).

test scripts has been considered as an approach that can result in improved maintenance economics for almost two decades [11].

In our work, we try to reap the advantages of both recording and descriptive programming. The relatively low cost of creating automated tests through recording can be combined with decreased brittleness and more efficient maintenance of the test scripts by refactoring the repetitive parts of these scripts. We aid this refactoring process using an automated method based on post-processing the recorded test scripts and identifying the potentially reusable parts. The proposed solution can be also applied to descriptively programmed — but poorly structured — scripts in which few or no reusable objects are used.

In Front-End-based automated test scripts, duplication can occur at several places. In simple form, the common locations can be categorized as follows:

(i) Setup, tear-down and other technical code locations that are common in the scripts (e.g. SUT configuration, parameter initializations, etc.)
(ii) Localization of the SUT FE elements and of particular actions performed on those elements. These sequences of common steps will be short and will likely occur in numerous tests where a particular element in the SUT FE is accessed and controlled by various tests covering various processes. An example might be: “Localize the account number text field in the payment form and enter a value.”
(iii) Sequences of steps that represent a business action from test logic viewpoint. Here, we expect longer common subsequences that are likely to occur in fewer test scripts. An example: “Enter the payment details, submit the payment and authorize it by typing of a sent SMS code”.

In each of these categories, we can identify reusable objects. In the first area (setup and tear-down), the potentially reusable objects are relatively easy to identify.

Therefore, we focus primarily on the second and third areas. In these areas, recorded and poorly structured automated test scripts usually contain considerable amounts of non-optimized duplicated code.

This paper is organized as follows. We present the proposed solution in Sec. 2. In Sec. 3, we describe the performed experiments and discuss the results. Section 4 presents the related works, and Sec. 5 lists the contributions of this study and concludes the paper.

## 2. Proposed Solution

First, we give a brief overview of the principles behind the TestOptimizer — the solution we propose to automatically identify potentially reusable subroutines in automated test scripts. After TestOptimizer has identified repeating code fragments in automated test scripts, it suggests the potentially reusable subroutines to the test script developer. The final decision concerning whether a subroutine is reusable is the responsibility of the developer. During this process, we report the positions in the test source code where a reusable object can be defined and where test automation code can be refactored. To identify repeating fragments, we do not consider only identical source code fragments because that approach would detect only trivial cases of code redundancy. Instead, we analyze the source code of automated test scripts to identify fragments that have the same semantics regarding the actions that the script performs in the SUT FE. At this stage, our semantic analysis is still approximate and can detect fewer refactoring opportunities than a human can; nevertheless, this approach is already more efficient than simply comparing source code fragments (the details are provided in the Sec. 3). We do not distinguish between completely reusable routines and partially reusable routines during the automated analysis: developers can subsequently tailor the subroutines manually to fit their needs. For instance, they can exclude steps from the subroutine or add new steps (e.g. test result assertions).

The TestOptimizer solution is flexibly configurable to accommodate various testing tools, languages and development styles. It analyzes the code of automated test scripts. This code can be written in a programming language (e.g. Java and Selenium WebDriver [12]), or the code can even be an internal representation of test cases in a particular tool (e.g. Selenese [13]).

The principles of this solution are outlined in Fig. 2. When submitted to the TestOptimizer server (step 1), the automated test source code is converted into abstract layer that reflects the semantics of the test steps it contains (step 2). In the abstract layer, analysis of repetitive parts is performed (steps 3 and 4). Then, based on this analysis, a set of potential common subroutines is prepared and provided to the user (step 5). The provided information can be then used to refactor the automated test scripts. The analysis rules parameters are stored in configuration file.

In the current phase, we do not refactor the test scripts automatically; therefore, the user maintains control of the source code during the process. The features of the

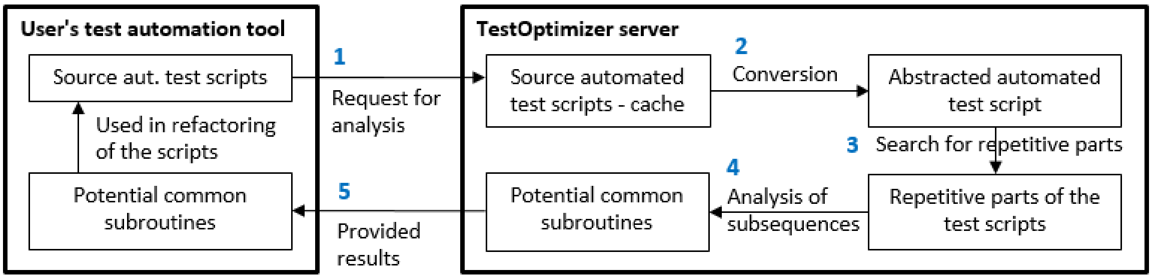

User's test automation tool
Source aut. test scripts
Used in refactoring of the scripts
Potential common subroutines
1
Request for analysis
5
Provided results
TestOptimizer server
Source automated test scripts - cache
2
Conversion
Abstracted automated test script
3
Search for repetitive parts
Repetitive parts of the test scripts
4
Analysis of subsequences
Potential common subroutines

Fig. 2. Conceptual schema of the proposed solution.

solution increase its flexibility and also differentiate our solution from previous work in this area [14], as discussed in Sec. 4.

In the following subsections, we describe the entire process in detail. Following the process steps outlined in Fig. 2, Sec. 2.1 describes step 2, Secs. 2.2 and 2.3 describe steps 3 and 4, respectively, and Sec. 2.4 describes the TestOptimizer server interface, corresponding to steps 1 and 5.

## 2.1. *Automated test script abstraction*

The test script analysis process converts the source test scripts to a universal abstraction before proceeding with further analyses for several reasons:

(i) To ensure platform independence of the proposed solution,
(ii) To conduct automated test script analyses on a semantic level, not only at the source code text level,
(iii) To analyze test scripts that differing in notation yet are coded in one programming language (e.g. different developer programming styles or when parts of the scripts are created in a framework, etc.), and
(iv) When fulfilling the requirements (i)–(iii), to still be able to reuse existing algorithms in searching for the longest common subsequences (LCS), which are text based. TestOptimizer uses a genetic algorithm, the use of which has previously been explored for the for LCS problem [15].

To identify the semantics of the steps in the test cases, we introduce “step signatures”. Each step of the analyzed test script is converted to a signature. Signatures hide the specifics of the particular programming language notations and allow the test case steps to be compared from a semantic point of view. Groups of identical signatures indicate potential common subsequences in the analyzed test scripts. Subsequently, we use this property to find the common subsequences (repetitive parts) in the test scripts.

Before describing the signature formally, we give an example. Test cases are sequences of steps that are executed through the testing tool; they are encoded as commands in a scripting language. Because high-level programming languages such as Java offer programmers a variety of options on how to implement any particular step, we must consider this issue when automatically analyzing the test source code. Listing 1 presents two such implementations of one identical test step written in Java and Selenium WebDriver represented in two different implementations.

```
driver.findElement(By.id("login")).click();

Element e = driver.findElement(By.id("login"));
e.click();
```

Listing 1. Example of syntactic differences in two test steps that have the same semantics (API objects appear in bold, methods in italics, and user-defined parameters are underscored).

In the source code, each test step consists of test automation application programming interface (API) objects, their methods and parameters. Practically, it is easy to imagine two test steps that perform the same action in SUT from a test semantics point of view but that differ in their specific notation (even in the same programming language and using the same test automation API).

Therefore, we translate these test steps to their signatures using a specialized converter. A signature $s$ is defined as follows:

$$s = \langle o, X_o, a, X_a \rangle,$$

where $o$ is an object, $a$ is an action, $X_o$ is a set of additional attribute parameters for identifying object $o$ in the SUT FE and $X_a$ is a set of parameters that specify the testing data with which the action $a$ will be performed.

**Object** $o = \langle c, n \rangle$ represents an abstraction of a SUT FE control element (for example, a text field or button link), independent from its physical implementation in HTML.

An object $o$ consists of a pairing of a **class**, $c$, and a **name**, $n$. The class defines the type of object (e.g. button or input box), while the name serves as a unique identifier for a particular instance of an object and also links the object to its physical representation in the SUT FE.

Practically, the object $o$ represents an atomic element of the SUT FE that can be located and is controlled by the test automation API. If the object name $n$ is not sufficient to clearly identify the FE element, other **attribute parameters** from $X_o$ can be used for its identification.

An attribute parameter, $x \in X_o$, is a pairing of an attribute name and its value.

An **action**, $a$, is performed on object $o$ in the SUT FE (for example submitting the form or clicking on a link). $A$ is the list of possible actions derived from the capabilities of a particular test automation API, and $a \in A$.

S is the set of all signatures created from the set of all analyzed test scripts.

In its textual representation (which is needed for further processing — the search for potential common subsequences in the test scripts), the signature is defined as the following regular expression:

```
obj:<class>{.<object_name>{+<attribute_parameter>=<value>}}&act:
<action>{+ <parameter>:<value>}
```

Table 1. Elements of a test step signature.

| Element | Description |
|---|---|
| **obj** | Keyword for object |
| `class` | Class $c$ of the object $o$ |
| `object_name` | Name $n$ of the object $o$ |
| `attribute_parameter` | Name of attribute parameter $x$: The attribute parameters can be used to specify the physical representation of object $o$, if the object_name is not sufficient to localize the element in the SUT front-end exactly. |
| **act** | Keyword for action |
| `action` | Identifier of action $a$ |
| `parameter` | Parameter specifying, with which data the action $a$ will be performed |
| `value` | General parameter value (string) |

Table 2. Levels of test step signature.

| Signature level | Signature |
|---|---|
| 0 | **obj**:`<class>`**&act**:`<action>` |
| 1 | **obj**:`<class>.<object_name>`{+`<attribute_parameter>` =`<value>`} **&act**:`<action>` |
| 2 | **obj**:`<class>.<object_name>`{+`<attribute_parameter>` =`<value>`} **&act**:`<action>` {+`<parameter>:<value>`} |

The individual elements of a signature are described in Table 1. We propose three types of signatures with different levels of detail, which are listed in Table 2.

The employment of signatures brings two main benefits — despite the fact that they are similar to real code. First, signatures allows us to isolate the TestOptimizer core algorithms from the specifics and complexity of different test automation notations (i.e. Python, Java, C# etc.), making TestOptimizer applicable to multiple languages and notations. This isolation also allows modular extensions of the TestOptimizer framework without requiring any changes to the core algorithms. Second, it allows us to flexibly configure which parts of the automated test scripts will be analyzed:

The Level 0 signatures consist of descriptions of objects and actions. Using Level 0, we search for groups of potentially similar actions performed on the same classes (which practically means types) of objects and independently on particular elements in the SUT FE. Level 0 is suitable for high-level analysis — identifying potential candidates for common reusable object allocation and controlling procedures.

Level 1 signatures address particular physical elements and actions. Nevertheless, at this level we do not analyze the parameter values with which the action will be performed. Instead, this level is used to identify repeating common subroutines in the analyzed test scripts that perform a certain action on a certain element of the SUT FE.

Level 2 signature analysis is the most detailed: at this level, we analyze the parameter values used in the actions. To optimize the comparison operations when searching for repeated common subsequences, long data strings in the signature are replaced by hashed values. For specific language test automation frameworks, we apply different sets of levels. For example, the MicroFocus Unified Functional

Testing (UFT) framework uses an automated test scripting language based on Visual Basic. This syntax allows using all three levels, 0–2. In contrast, for Selenium WebDriver automated test scripts, which are written in Java, only levels 1–2 are relevant.

The main difference between Level 1 and Level 2 is in their recognition of action parameters. In some cases, it also worth distinguishing between identical steps but with different parameters because they may represent different test semantics. For example, in role based tests, the steps might be identical but a test developer might want to keep them separate to add different verification steps after creating the tests. In this case, the user uses Level 2 signatures. In contrast, when action parameters do not matter, Level 1 signatures can be used.

Table 3 lists examples of these levels for commands taken from automated test scripts created by UFT and Selenium WebDriver. For UFT, the Visual Basic Script command is

`WebField("field1").Set "value"`

whereas a similar Java command for Selenium WebDriver is:

`driver.findElement(By.linkText("Automotive")).set("value");`

Information extracted from the command for a particular signature level is highlighted with underlined bold text in Table 3.

Table 3. Examples of information extracted from automated test commands for signature levels 0–2.

| Signature level | Information extracted from HPE UFT command | Information extracted from Selenium WebDriver command |
|---|---|---|
| 0 | **WebField**("field1").**Set** "value" | not relevant |
| 1 | **WebField**("**field1**").**Set** "value" | driver.findElement(By.linkText ("**Automotive**")).set("value"); |
| 2 | **WebField**("**field1**").**Set** "**value**" | driver.findElement(By.linkText ("**Automotive**")).set("**value**"); |

Now that we have described the signature, we can describe the overall conversion process. The process of converting the source code scripts to their abstractions is outlined in Fig. 3.

To transform the original automated test script to an abstracted test script $t = (s_1, s_2, \ldots, s_n)$, we designed a special converter based on a lexical and semantic analysis of the source code. In the lexical analysis phase, we tokenize the code and assign meanings to every token (keywords, numbers, special characters, etc.), which we use as an input for the semantic analysis. To understand Java semantics and, thus, test steps written in Java, we implemented a semantic analyzer based on the reference grammar defined by Oracle [16]. The semantic analyzer parses tokens and creates a syntax tree. We use the created syntax tree to translate source code statements to defined test step signatures. Currently, we have implemented and tested the parser for Java with Selenium WebDriver and the JUnit framework.

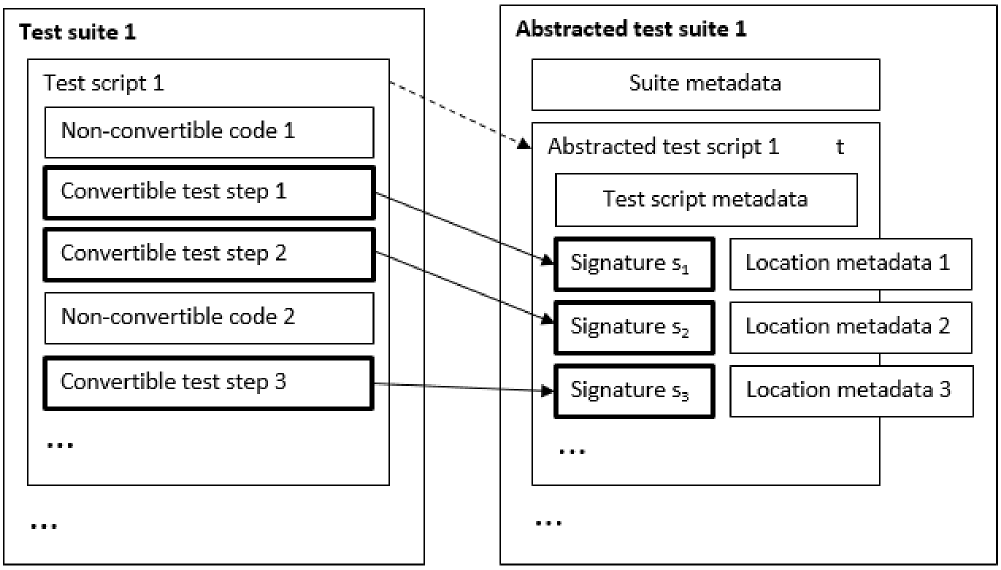


Fig. 3. Conversion of source automated test scripts to their abstractions.

In the transformation process, we distinguish between

(i) convertible test steps, which represent test actions in the SUT FE and are translated to test step signatures of the abstracted test script, and
(ii) non-convertible code, which is not translated because it is not relevant from a test semantics viewpoint.

The convertible steps involve actions in the SUT FE that change the state of the application (such as the example given in Fig. 2) and assertions of the expected results. Examples of convertible and non-convertible steps are listed in Table 4. The first two rows show convertible steps because WebDriver navigates to a particular page of the SUT FE in the first row, and then, the user checks for the presence of a

Table 4. Example of convertible and non-convertible steps in an automated test implemented in Selenium WebDriver and the JUnit framework.

| Type of step | Example of source code |
|---|---|
| Convertible | An action in SUT front-end<br>`driver.get(baseUrl + "/web/en-US/default.aspx?root=1");` |
| Convertible | Assertion of results<br>`assertTrue(isElementPresent(By.id("ContentInfo_sp")));` |
| Not convertible | JUnit framework code except @Test<br>`@After`<br>`public void tearDown() throws Exception {`<br>`    driver.quit();`<br>`    ...`<br>`}` |

given element in the SUT FE. In comparison, the command in the third row causes WebDriver to shut down; it does not involve the SUT FE.

JUnit/TestNG annotations other than @Test such as @BeforeSuite, @AfterClass and @BeforeMethod usually do not involve operations on the SUT FE because these commands set up the test environment, prepare test data, implement common actions or perform script cleanup. Moreover, commands in those actions have already been identified as reusable, otherwise the user would not have put them into annotated methods (for instance, the Selenium IDE saves recorded test steps methods other than those annotated by @Test when the test is exported into a JUnit format). Therefore, these other methods are not the targets of our investigation. However, we plan to add configurable support for determining which annotated methods should be analyzed in a future version of the TestOptimizer framework.

Analysis — determining which parts of the code are convertible and which are not — is based on the parsing rules stored in the converter configuration.

To maintain traceability between the signatures and the original test scripts, we use location metadata (more details follow in Sec. 2.4). This location metadata contains the respective code line-number range of the analyzed test script and the original code fragment. These metadata are not included in the search for potentially reusable parts (the search for the LCS, which is explained in the following subsection).

Moreover, each of the abstracted test scripts is accompanied by a set of metadata (see the test script metadata in Fig. 3) that includes the source filename and signature count. The analyzed test suite (see the suite metadata in Fig. 3) contains both the code of the test script and a count of the analyzed test scripts.

### 2.2. *Identification of potential common subroutines*

**The input to the analysis is a set of abstracted test scripts**, $T_A = \{t_1, t_2, \ldots, t_n\}$, each of which is composed of an ordered sequence of step signatures: $t_x = (s_1, s_2, \ldots, s_n)$, $s_1, s_2, \ldots, s_n \in S$, $t_x \in T_A$.

During the analysis, we search for potential subroutines. We denote a potential subroutine as $p = (s_1, s_2, \ldots, s_m)$, $s_1, s_2, \ldots, s_m \in S$, where $p$ must be present in two or more abstracted test scripts from $T_A$.

Then, $T_F$ is a set of analyzed abstracted test scripts in which $p$ is present, $T_F \subseteq T_A$.

**The output of the analysis is a set of potentially repeated subroutines**, $P$ (generally, more potentially repeated subroutines exist in $T_A$).

After the analyzed test scripts are converted to their abstractions, the next step is to identify the LCS in $T_A$. By translating the source test scripts to their abstractions, we are able to analyze the test steps from a semantic viewpoint. Nevertheless, using the signatures, we still process the text strings; therefore, we can reuse already defined algorithms for the problem of finding LCS, which generally represents an NP-complete problem. In TestOptimizer, we used the genetic algorithm, whose usability for the LCS problem has been explored previously [15, 17].

The metadata of an abstracted test script are not considered in the analysis. The LCS search is performed by the Solver component.

Before we run the genetic algorithm on $T_A$ to find common subsequences, we reduce $T_A$ by excluding those test scripts that do not have at least one common signature with the other test scripts in $T_A$. This reduction is performed by the function SELECT_PROSPECTIVE_TESTS, where $T_P$ denotes a set of prospective test scripts to analyze, $T_P \subseteq T_A$, and $T_P$:= SELECT_PROSPECTIVE_TESTS($T_A$).

For the SELECT_PROSPECTIVE_TESTS function, we adopted the Karp-Rabin string matching algorithm [18]. The SELECT_PROSPECTIVE_TESTS function searches for simple duplications in signatures among the set of analyzed abstracted test scripts, $T_A$. An abstracted test script test, $t_n \in T_A$, is excluded from $T_P$ when

$$\forall \quad s_n \in t_n : \forall \quad t_{\mathrm{o}} \in T_A / \{t_n\} : s_n \notin t_o$$

After reducing $T_A$ to $T_P$, we perform LCS on $T_P$. The genetic algorithm uses a search chromosome denoted as $t_c \in T_P$. The LCS search runs in a defined number of iterations. At the end of the run, the potential variants of results are evaluated by the fitness function to select the best result (set of candidate sequences), which is then processed further.

In the TestOptimizer project we experimented with several strategies to optimally configuring the LCS search in this specific area. We have tried to design the fitness function to prefer the longest possible sequences and the sequences that occur most often. The results, besides the settings used for the LCS algorithm, depend on the selection of the chromosome $t_c$. This selection is discussed later.

The question is: What would be more interesting for the test code developer from a refactoring viewpoint? A few long common subsequences or a larger number of shorter common subsequences? It is difficult to generalize which approach would be more efficient and useful in a particular practical case because the style in which the scripts are structured and the specific code being analyzed differs. A user might be interested in suggestions at different levels. Therefore, we decided to let the user wield the flexibility of making this decision. We defined metrics to select the best result for the potential subsequences in the set of analyzed scripts: Subroutine Quality (SQ), defined by Eq. (1), and Analysis Variant Quality (AVQ), defined by Eq. (2).

$$\mathrm{SQ}(p, T_A) = |p| \cdot |T_F|. \tag{1}$$

$T_F$ is a set of analyzed abstracted test scripts in which $p$ is present, and $T_F \subseteq T_A$.

$$\mathrm{AVQ}(P, T_A) = \sum_{p \in P} \mathrm{SQ}(p, T_A). \tag{2}$$

A user can adjust the preferences by setting the parameters LengthWeight and TestCountWeight, which are real number constants whose values are set by the user before the analysis of the automated test set begins. A user can determine the best values for LengthWeight and TestCountWeight experimentally by running several analysis with different values.

By increasing LengthWeight, the user prefers the identification of longer common subsequences, which tend to occur less often in analyzed test cases, and by increasing TestCountWeight, the user prefers the identification of possibly shorter common subsequences that occur more often in analyzed test cases.

Therefore, $\text{LengthWeight} + \text{TestCountWeight} = 1$, $\text{LengthWeight} > 0$ and $\text{TestCountWeight} > 0$.

The LengthWeight and TestCountWeight parameters are used in several parts of the analysis, as described later. We first use these parameters in the configuration of the fitness function for LCS search. This configuration is presented in Listing 2.

```
fitness = |p| * LengthWeight + |T_F| * TestCountWeight;
if ( |p| == |t_c| ) then {
       fitness = fitness * 10;
}
if ( |T_F| == |T_P| ) then {
       fitness = fitness * 5;
}
```

Listing 2. Configuration of fitness function for Longest Common Subsequences (LCS) search.

If the system does not find any reusable routines it may mean that either (i) there are no common steps in the analyzed scripts, or (ii) the user set inappropriate values for the configuration of one or more input parameters. Consequently, the solver inclines to a local optimum found in a single test. In such cases, it is recommended that the user restart the analysis with a different configuration.

In the following subsection, we present the details of the algorithms the TestOptimizer uses to identify potential common subroutines in a set of abstracted automated test scripts.

### 2.3. *Used algorithms*

As introduced above, the output of the analysis is a candidate sequence created for chromosome $t_c$. The sequence is encoded as a binary string. If a signature (representing a test step) from chromosome $t_c$ is present in one or more abstracted test scripts (we denoted them already as $T_F, T_F \subseteq T_P, t_c \notin T_F$), it is encoded by a one in the binary string; otherwise, it is encoded by a zero. Because [15] demonstrated that the genetic algorithm achieves better results when the population is empty (i.e. the population is represented by zeros), we do not use any technique to generate the population.

Figure 4 shows an example in which $T_P = \{t_1, \ldots, t_5\}$ and $t_1 = t_c$ is the chromosome. $T_F$ is not yet known. The $t_c$, potential common steps (which will be analyzed later to find potential subroutines) are depicted by black rectangles; other steps are depicted by white rectangles.

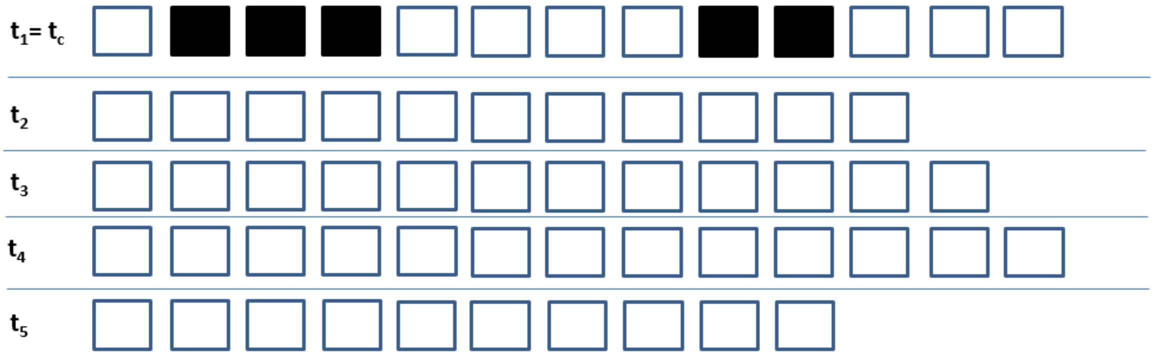


Fig. 4. Example of an output of the LCS algorithm: the potential common steps for chromosome $t_c$.

When the potential common steps are identified in chromosome $t_c$, we need to localize them in $T_P \backslash \{t_c\}$. The details are presented in Algorithm 1. The asymptotic complexity of the algorithm in the worst case is given by the count of found signatures in the chromosome, the size of the analyzed set and by the number of signatures in a test from the analyzed set. Summarized, this complexity is $O(|T_p||t_c||t_{\max}|)$, where $t_{\max}$ is an abstracted test script with the highest number of signatures, $t_{\max} \in T_P$.

In our example, Fig. 5 presents the result of Algorithm 1: a set of analyzed abstracted test scripts, in which common steps have been identified. In this example, there is a chance that the test script set $\{t_1, t_2, t_4, t_5\}$ can contain potential common subroutines. Tests $t_1, t_2, t_4$ and $t_5$ are likely to contain one three-step subroutine and one two-step subroutine while test $t_4$ is likely to contain only the three-step subroutine. The test script $t_1$ is the chromosome and does not have to be analyzed but we need to find the remaining tests.

Thus, in the next step, we find the potential common subroutines in $T_P$. This is done by Algorithm 2. The output of Algorithm 2 is a set of detected potential common subroutines, denoted previously as $P$. The parameter RequiredMinLength

**Algorithm 1.** Localization of the potential common steps in the abstracted test scripts.

```
FIND_COMMON_STEPS($t_c, T_P$)
foreach $s_c \in t_c$                         for each signature from the chromosome
    foreach $t \in (T_P \backslash \{t_c\})$      for each abstracted test script in the
                                              analyzed set, excluding the chromosome
        foreach $s \in t$                     for each signature from the analyzed
                                              abstracted test script
            if $s_c = s$ THEN s.common($s_c$) mark $s$ as being common with $s_c$
return $T_P$
```

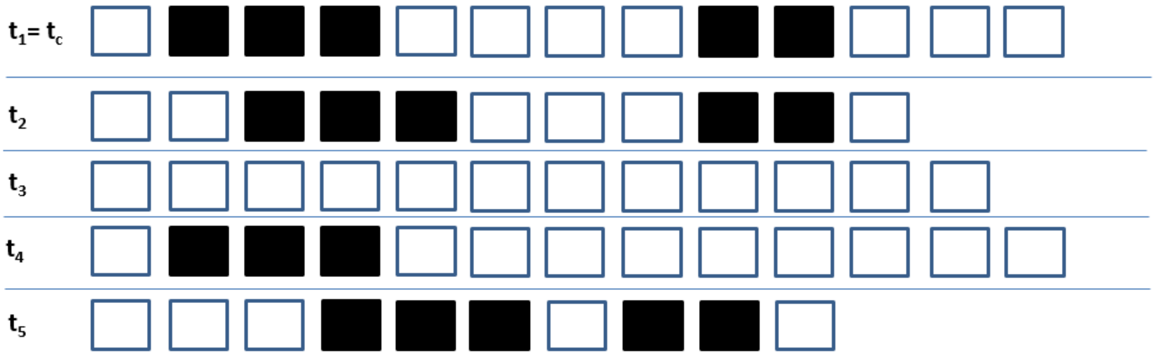


Fig. 5. The potential common steps localized in abstracted test scripts.

specifies minimal length (the number of test-step signatures) of potential common subroutines that will be identified. This parameter is specified by the user.

In Algorithm 2, the array "occurrence" contains information concerning the number of tests from $T_P$ in which a particular signature occurs. Then, the "max_occurence" variable contains the largest number of tests in which some of the signatures have occurred thus far during the processing. The asymptotic complexity of this algorithm is given by the size of the analyzed set and the number of signatures in a processed test for the first phase. The computation of the second phase also depends on the number of detected potential subroutines. Thus, the complexity is $O(|T_p|(|t_{\max}| + |t_{\max}||R_{\max}|))$, where $t_{\max}$ is the abstracted test script with the highest number of signatures, $t_{\max} \in T_P$, and $R_{\max}$ is the largest of the sets of detected potential common subroutines created for each abstracted test script in the analyzed $T_P$.

In the physical implementation of the algorithm, we store the information concerning where in the analyzed scripts the signature is positioned. This is ensured by the following additional metadata of the signature object:

(i) Where the signature s is positioned in the abstracted test script, $t \in T_P$; and
(ii) Where the signature is positioned in the original test script: the location metadata presented above (refer to Fig. 3).

Figure 6 illustrates the situation for our example. Common steps in the tests are depicted by black rectangles and other steps by white rectangles. In our example, there are two potential common subroutines, depicted as Subroutines A and B. It holds that $\exists \quad T_1 \subseteq T_P : |T_1| > 1, \forall \quad t_x \in T_1 : \mathrm{A} \subset t_x$ and $\exists \quad T_2 \subseteq T_P : |T_2| > 1, \forall \quad t_x \in T_2 : \mathrm{B} \subset t_x$.

In Fig. 6, potential common subroutine B follows subroutine A in each of the test scripts in which they occur. However, this is only an example; generally, the identified potential subroutines detected by TestOptimizer can occur in any order in the analyzed test scripts. For example, in some of the analyzed test scripts, subroutine B could occur before subroutine A.

**Algorithm 2.** Finding common subroutines in the abstracted test scripts.

```
FIND_SUBROUTINES($t_c, T_P$,RequiredMinLength)
    SET occurrence[] to 0    set all elements of occurrence array to 0
    max_occurence := 0       set max_occurence to 0
    occurrence[] :=
        FIND_CANDIDATES($t_c, T_P$,RequiredMinLength, occurrence[], max_occurence)
                             build potential candidate subroutines from common steps in tests
    $C := \emptyset$                  empty the sequence of candidate steps
    $P := \emptyset$                  empty the set of final detected potential common subroutines
    $P :=$ FILTER_CANDIDATES($t_c, T_P$, RequiredMinLength, occurrence[], max_occurence, $C$, $P$)
                             exclude low quality candidate subroutines
    return $P$
FIND_CANDIDATES($t_c, T_P$, RequiredMinLength, occurrence[], max_occurence)
    $\forall t \in (T_P \backslash \{t_c\})$          for each abstracted test script in the analyzed set
        $C := \emptyset$              empty the potential sequence of common steps
        $R := \emptyset$              empty the set of detected potential common subroutines
        foreach $s \in t$         for each signature from the analyzed abstracted test script
            if $s$ being marked as common with any other signature (see the
            LOCALIZE_COMMON_STEPS, Algorithm 1) then
                $C := (C, (s))$       append the potential sequence of common steps by signature s
            else
                if $|C| >=$ RequiredMinLength then
                    $R := R \cup \{C\}$          add the potential sequence of common steps
                                        C to the set of detected
                                        potential common subroutines
                end if
                $C := \emptyset$ empty the potential sequence of common steps
            end if
            if $|C| >=$ RequiredMinLength then
                $R := R \cup \{C\}$       add the potential sequence of common steps C to the set of
                                detected potential common subroutines
                $C := \emptyset$             empty the potential sequence of common steps
            else
                $C := \emptyset$             empty the potential sequence of common steps
            end if
        foreach $r \in R$             for each of detected potential common subroutines
            foreach $s \in r$         for each signature in a detected potential subroutine
                occurrence[$s$] = occurrence[$s$] + 1
                if max_occurence < occurrence[$s$] THEN max_occurence = occurrence[$s$]
                        end if
    return occurrence[]
```

**Algorithm 2. (*Continued*)**

```
FILTER_CANDIDATES($t_c, T_P$, RequiredMinLength, occurrence[], max_occurence, $C, P$)
  $\forall s_c \in t_c$                          *for each signature from the chromosome*
    if occurrence[$s_c$] = max_occurence then
      $C := (C, (s_c))$                   *append the potential sequence of common steps by*
                                          *signature $s_c$*
    else
      if $|C|$ >= RequiredMinLength then
        $P := P \cup \{C\}$              *add the potential sequence of common steps C to the set of*
                                          *detected potential common subroutines*
      end if
      $C := \emptyset$                        *empty the potential sequence of common steps*
    end if
  return $P$
```

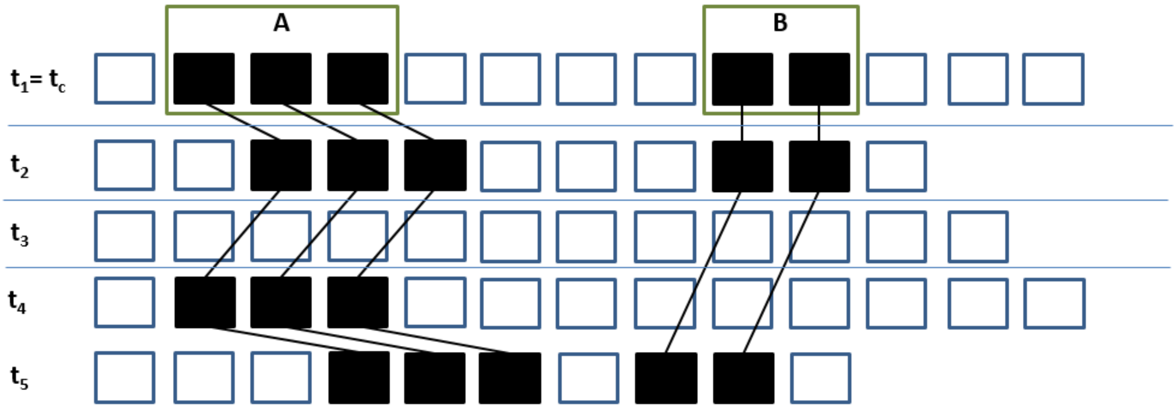


Fig. 6. The potential common subroutines identified in abstracted test scripts.

In addition to the configuration of the genetic algorithm for the LCS search, the results also depend on the selection of chromosome $t_c$. We allow two principal modes of analysis.

(i) The user wants to know whether a particular subroutine (for instance an already defined reusable subroutine) occurs in a set of analyzed test scripts. In this case, a test that contains this subroutine will be set as chromosome $t_c$.

(ii) The user simply wants to know if there are potential common subroutines in a set of automated test scripts. In this case, chromosome $t_c$ is unknown in advance.

Table 5. Methods of chromosome selection.

| Method | Selection of $t_c$ | Description |
|---|---|---|
| CHROMOSOME_MANUAL | by the user | Principle: Full user control: The user selects the chromosome and a set of test scripts to analyze.<br>Use case: The user wants to know whether a particular reusable routine occurs in a set of analyzed scripts.<br>Algorithm: CHROMOSOME_MANUAL_SEARCH, (Algorithm 3 is defined later). |
| CHROMOSOME_AUTO | Automated, for selected suitable $t_c \in T_p$ | Principle: The user selects only a set of test scripts to analyze; the chromosome is selected by the solver automatically.<br>Use case: The user simply wants to know whether there are common subroutines in the set of scripts.<br>Algorithm: CHROMOSOME_AUTO_SEARCH, Algorithm 4 defined further on. |

**Algorithm 3.** Processing in CHROMOSOME_MANUAL mode.

```
CHROMOSOME_MANUAL_SEARCH($T_A$, RequiredMinLength, LengthWeight,
                                   TestCountWeight)
$T_P$ :=SELECT_PROSPECTIVE_TESTS($T_A$)                select prospective tests to analyze
LCS($t_c, T_P$, LengthWeight,TestCountWeight)          longest common subsequences search
FIND_COMMON_STEPS($t_c, T_P$)                          post-processing, see Algorithm 1
$P$ := FIND_SUBROUTINES($t_c, T_P$, RequiredMinLength)   post-processing, see Algorithm 2
return P
```

Table 5 lists the details.

Algorithm 3 specifies the processing in CHROMOSOME_MANUAL mode. $T_A$ is reduced to $T_P$ by SELECT_PROSPECTIVE_TESTS, and then, LCS followed by the algorithms FIND_COMMON_STEPS and FIND_SUBROUTINES are performed for the defined $t_c$.

Algorithm 4 specifies processing in CHROMOSOME_AUTO mode. First, $T_A$ is reduced to $T_P$ by SELECT_PROSPECTIVE_TESTS; then, prospective chromosomes are selected from $T_P$. For each of these chromosomes, we perform the LCS search followed by the algorithms FIND_COMMON_STEPS and FIND_SUBROUTINES. The results are collected and finally, the best results are selected by the AVQ metric (therefore, the LengthWeight and TestCountWeight parameters plays a role in the selection). The asymptotic complexity of Algorithm 4 is determined by the complexity of the LCS search algorithm, which is $O(n^{|Tp|})$, where $n$ is the sum of $|t|$ for all $t \in T_P$ [19].

$T_C$ denotes a set of prospective chromosomes, $T_C \subseteq T_P$.

The parameter IterationsCount specifies how many iterations for various chromosomes should be executed in total. This limit is defined for performance reasons. The maximum value of iterationsCount is $|T_C|$.

**Algorithm 4.** Processing in CHROMOSOME_AUTO mode.

```
CHROMOSOME_AUTO_SEARCH($T_A$, RequiredMinLength, LengthWeight,
                       TestCountWeight, IterationsCount)
$X := \emptyset$                                      empty the set of results
$T_P :=$ SELECT_PROSPECTIVE_TESTS($T_A$)              select prospective tests to analyze
$T_C :=$ SELECT_PROSPECTIVE_CHROMOSOMES($T_P$)        select prospective
                                                      chromosomes (see further)
if IterationsCount $> |T_C|$ then IterationsCount $= |T_C|$   adjust the number of iterations
foreach $t_c \in T_C$                                 for all prospective chromosomes
    if IterationsCount $> 0$ then                     still to iterate?
        LCS($t_c, T_P$,LengthWeight,TestCountWeight)  longest common subsequence search
        FIND_COMMON_STEPS($t_c, T_P$)                 post-processing, see algorithm 1
        $F :=$ FIND_SUBROUTINES($t_c, T_P$,RequiredMinLength)   post-processing,
                                                      see algorithm 2
        $X := X \cup \{F\}$                           collect the potential results
        IterationsCount = IterationsCount $- 1$       decrease iteration counter
    end if
$P = x, x \in X$, $x$ is having the highest AVQ($x, T_P$)
return $P$                    select the best results in accord to user's preferences
```

The SELECT_PROSPECTIVE_CHROMOSOMES function searches for tests that are suitable for selection as chromosomes. We employ the Karp–Rabin [17] algorithm to count duplications in the abstracted test scripts from $T_P$. Based on the number of duplicates found, we select the abstracted test scripts whose steps are as different as possible. Those tests are then selected as chromosomes.

### 2.4. *Parameterization and request and response structure*

In this subsection, we summarize the parameters that influence the analysis and describe the output of the analysis — the set of potential common subroutines that the end user will use to refactor the test scripts.

The request for analysis consists of the parameters summarized in Table 6. The result of the analysis is summarized in Table 7. The resulting potential common subroutines indicate potentially reusable objects in the scripts. During the refactoring process, it is the user's responsibility to decide, how to best employ the provided facts during code structuring. The structure of the analysis results is summarized in Fig. 7. Technical metadata are not depicted in the figure.

To determine the parameters scriptName, lineNumberFrom and lineNumberTo, we use the location metadata paired with the signatures as discussed above. The values of codeFragment are then determined from source scripts stored in the cache, using the parameters scriptName, lineNumberFrom and lineNumberTo.

Table 6. Request parameters influencing the analysis.

| Parameter | Description |
|---|---|
| requestID | ID of the request specified by user. The ID allows to track the submitted request during processing on the TestOptimizer server |
| converterType | The type of converter to use (which translates original test scripts to their abstractions). TestOptimizer provides different converters for different languages and test automation APIs. |
| startPoint | User can select one of following options:<br>NEW_UPLOAD - Set of automated test scripts for analysis are uploaded to the server. The scripts are then converted to abstracted test scripts $T_A$ and analyzed for potential common subroutines by specified parameters.<br>SCRITPS_CACHED - Uploaded scripts are taken from cache and converted to abstracted test scripts $T_A$, then analyzed for potential common subroutines by specified parameters.<br>ABSTRACTION_CACHED – Created $T_A$ is analyzed again with different parameters |
| scriptCacheID | In case of SCRITPS_CACHED: ID of set of already uploaded test scripts for analysis. |
| scripts | In case of NEW_UPLOAD: Stream of automated test scripts for analysis to upload. |
| signatureLevel | Level of detail, which is captured by signatures during the conversion to abstracted test script. For overview of levels, refer to Table 2. Expected in case of NEW_UPLOAD and SCRITPS_CACHED options. |
| requiredMinLength | minimal length of potential common subroutines, which will be identified (refer to RequiredMinLength parameter in Algorithm 2) |
| lengthWeight | Preference of longer potential subsequences in less test scripts by LengthWeight parameter (refer to fitness function in LCS and Algorithm 3) |
| testCountWeight | Preference of shorter potential subsequences in more test scripts by TestCountWeight parameter (refer to fitness function in LCS and Algorithm 3) |
| chromosomeMode | CHROMOSOME_MANUAL (user's preference to manually select a chromosome) or CHROMOSOME_AUTO mode, refer to Table 5. |
| chromosome | In case of CHROMOSOME_MANUAL mode: explicit selection of chromosome $t_c \in T_A$ |
| iterationsCount | In case of CHROMOSOME_AUTO mode: IterationsCount parameter, specifying how many iterations for various chromosomes will be executed (refer to Algorithm 3) |

Table 7. Structure of the response, including parameters that influenced the analysis.

| Parameter | Description |
|---|---|
| **TECHNICAL METADATA** | |
| requestID | requestID (copied from the request call) |
| scriptCacheID | ID of the server cache record in which the analyzed original test scripts are saved |
| Log | Access to the log, which lists details from the conversion and analysis process (a link to the log file stored on the server) |
| **SUBROUTINE SET**: For the set of potential common subroutines $P$: | |
| totalNumberOfScripts | Total number of analyzed test scripts, $\|T_A\|$ |
| lengthWeight | LengthWeight parameter specified in the request |
| testCountWeight | TestCountWeight parameter specified in the request |

Table 7. (*Continued*)

| Parameter | Description |
|---|---|
| AVQ | AVQ($P, T_A$, LengthWeight, TestCountWeight) value to help the user to decide, if analysis result is good enough for particular $T_A$, or if to try to analyze $T_A$ again with different parameters |
| **SUBROUTINE**: For each potential common subroutine $p \in P$: | |
| subroutineID | ID of potential common subroutine $p$ |
| SQ | SQ($p, T_A$) value to help the user to decide, which of the potential subroutines to prefer in identification of reusable objects |
| numberOfScripts | Number of test scripts, where the potential common subroutine $p$ occurs |
| scriptNames | List of names of test scripts, where the potential common subroutine $p$ occurs |
| **SCRIPT**: For each of test scripts, where the potential common subroutine $p$ occurs: | |
| scriptName | Name of test script, where potential common subroutine $p$ occurs |
| lineNumberFrom | Number of line in the original test script code, where potential common subroutine $p$ starts. |
| lineNumberTo | Number of line in the original test script code, where potential common subroutine $p$ ends. |
| codeFragment | Aggregated fragment of analyzed original test script code. |
| **SIGNATURE**: For each occurrence of signature $s \in p$ in an original test script: | |
| subroutineIDref | Reference to ID of potential common subroutine $p$ |
| scriptName | Names of test script, where s occurs |
| lineNumberFrom | Number of line in the original test script code, where signature $s$ starts. |
| lineNumberTo | Number of line in the original test script code, where signature $s$ ends. |
| codeFragment | Fragment of analyzed original test script code. Can be different for individual signatures from $p$. |

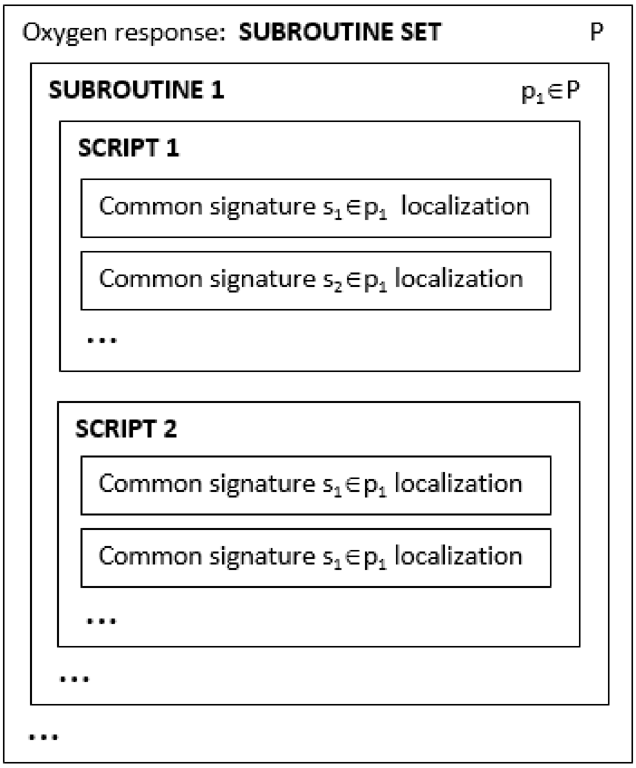

Oxygen response: SUBROUTINE SET
P
SUBROUTINE 1
$p_1 \in P$
SCRIPT 1
Common signature $s_1 \in p_1$ localization
Common signature $s_2 \in p_1$ localization
...
SCRIPT 2
Common signature $s_1 \in p_1$ localization
Common signature $s_1 \in p_1$ localization
...
...
...

Fig. 7. Structure of analysis results provided by the TestOptimizer server.

We start this process for particular signatures (SIGNATURE level, see Table 5). Then, we compose the allocation data to the blocks in the order in which the signatures occur in the potential common subroutine in the original test script (SCRIPT level, see Table 5).

The codeFragment can be considered as a duplicate at first glance; nevertheless, there are several reasons why we include the original code fragment in the response:

(i) We are abstracting the automated test scripts to capture the semantics of their steps; therefore, the specific text of a codeFragment could be different even though their signatures are identical in $p$.
(ii) Using codeFragment simplifies the implementation of the plugin for the user's test automation tool.
(iii) For testing purposes, the returned codeFragment can be quickly compared with a fragment of the original source code in the user's test automation tool, which is specified by scriptName, lineNumberFrom and lineNumberTo.

## 3. Experimental Verification

In this section, we describe a physical implementation of the proposed solution and some performed experiments, during which we both adjusted the configuration details of the proposed method and verified its practical functionality.

### 3.1. *Prototype implementation*

The TestOptimizer server is implemented using the Java 2 Enterprise Edition server application and a relational database combined with the file system as data storage. The system exposes its functionality via an API. Through this API, a user can upload automated test scripts for analysis and retrieve the results (refer to the description of the request and response structure in Sec. 2.4). To make working with the TestOptimizer server user friendly, plugins can be created for individual test automation Integrated Development Environments (IDEs).

Using the API interface, we can also connect to a web console through which administrative tasks can be performed for the TestOptimizer server Another option for performing administrative tasks is to use the command-line interface on the TestOptimizer server. The high-level architecture of the system from a conceptual point of view is depicted in Fig. 8. Storage components are depicted by the grey rectangles (physically, a common relational database is used). The arrows indicate the main data flows in the process.

Processing in the TestOptimizer server is designed as a modular pipeline; the individual parts can be adjusted flexibly. The pipeline starts with a user's request to optimize scripts. The request is handled by the *runAnalysis* function of the API.

After upload to the server via the API, analyzed automated scripts are stored in the Script cache to save network traffic and reduce processing time (which is

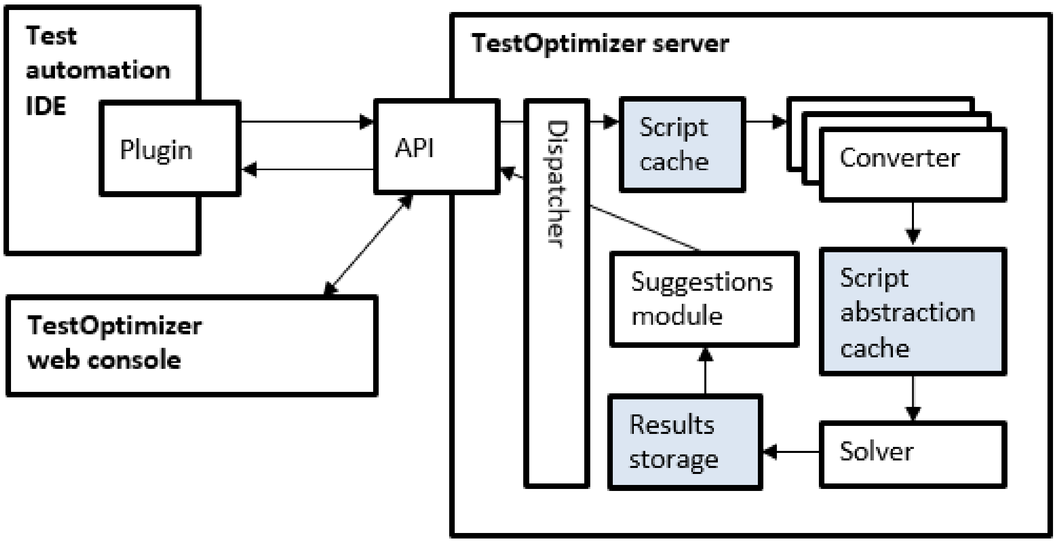


Fig. 8. Architecture overview of TestOptimizer prototype.

particularly important when a user reruns the analysis with different parameters on the same set of scripts). This is determined by the *startPoint* parameter in the *runAnalysis* request.

In the Converter, the original test scripts are converted to the abstracted test script form (refer to Sec. 2.1). For each combination of particular programming language and test automation API (for instance Java and Selenium WebDriver API or Visual Basic for HPE UFT), a variant of the Converter must be prepared. The implemented prototype includes a converter for Java and Selenium WebDriver API. The specific converter used for the analysis is specified by the *converterType* parameter in the *runAnalysis* request.

Abstracted test scripts are stored in the Script Abstraction cache. Records in this cache are linked with the original scripts stored in the Scripts cache. When the user specifies the *startPoint* as ABSTRACTION_CACHED, the system checks whether the Script Abstraction cache contains previously created abstractions for the specified *scriptCacheID* and *signatureLevel.* If yes, this cache is used, otherwise the Converter is executed again for this new configuration.

The analysis process continues in the Solver component, which performs the main computations. The key goal for the Solver is to find the longest common subsequences in the chromosome using the core LCS algorithm. Subsequent tasks are (i) to identify common steps in the subsequences (the FIND_COMMON_STEPS algorithm) and (ii) to compute the subroutines from the common steps (the FIND_SUBROUTINES algorithm). The Solver optionally also performs the CHROMOSOME_AUTO_SEARCH. The results are stored in the Results storage and processed by the Suggestion module to provide the response from the system interface.

Due to the processing times, the request and response API calls are asynchronous. By making a *runAnalysis* request, a user starts the processing on the server. To track the submitted request during further processing, the *requestID* assigned by user is used.

The whole process is orchestrated by the Dispatcher component. The Dispatcher also manages the pool of possible concurrent requests to the TestOptimizer sever.

### 3.2. *Experiments*

We verified the proposed solution through a set of experiments, in which we used automated testing scripts from several test automation projects. For these experiments, we defined two research questions:

(i) **Functionality of the solution**: Does the proposed algorithm work on various real automated test scripts and does it provide better results than typical established solutions for identifying duplicate code?

(ii) **Usability of the solution in the test automation process**: does the analysis save the developer time during test script refactoring?

Table 8 describes the sets of automated test scripts (taken from real test automation projects) used in testing the proposed solution. All the test scripts were recorded in Selenium IDE, then exported to Selenium WebDriver (we selected this setup because semi-automated optimization of recorded scripts is the major use case for TestOptimizer).

Regarding the first research question, we want to determine how efficiently the solution detects potential common subroutines from real automated test scripts. To be more specific: **Does the solution work properly and does it provide better results than common approaches based on analysis of source code fragments which do not reflect the specific context and structure of automated test code?** To answer this question, we compared our approach, based on the semantics of test steps, with a common approach based on a direct analysis of the source code fragments. For this comparison, we chose the established PMD tool [20] for the Eclipse IDE, which uses the Rabin–Karp string search algorithm to find duplicated code. We analyzed the sets of test scripts (see Table 8) using both the PMD and TestOptimizer server and compared the results, which are presented in Tables 9 and 10. In this comparison, in addition to the analysis variant quality

Table 8. Sets of automated test scripts used in experiments.

| Script set ID | Language | Number of scripts | Number of code lines |
|---|---|---|---|
| 1 | Java, Selenium WebDriver | 70 | 6728 |
| 2 | Java, Selenium WebDriver | 99 | 9345 |
| 3 | Java, Selenium WebDriver | 50 | 4507 |
| 4 | Java, Selenium WebDriver | 150 | 16347 |
| 5 | Java, Selenium WebDriver | 120 | 11704 |

(AVQ) measure defined above, we also used an averaged value of sequence quality (ASQ):

$$\mathrm{ASQ}(P, T_A) = \sum_{p \in P} \mathrm{SQ}(p, T_A)/|P|. \tag{3}$$

For the results provided by both the PMD and TestOptimizer server, we also performed a manual analysis to determine which of the found potential subsequences is truly suitable for refactoring in the form of a common subroutine in the context of the automated test scripts. Manual analysis of the true relevance of the identified potential subroutines was performed by two expert test-automation-code developers, who were asked to (i) exclude all identified subroutines already included in any longer subroutine; (ii) exlude all subroutines in which object descriptors were mistaken for action parameters; and (iii) exclude all subroutines in which instances of classes and static members are mistakenly identified (for example, a driver.findElement command may be mistaken for an Assert.AssertTrue command). Each expert developer carried out the manual analysis; subsequently, we performed a cross check to verify the results.

In the first part of Table 9 (*Raw data*), all the identified potential common subroutines produced are evaluated, using the AVQ metric. Then, in the second part, *Truly relevant potential subroutines after manual analysis*, which is described in the previous paragraph, we provide the numbers after the manual analysis and correction that identified those potential common subroutines that are truly relevant candidates for a common subroutine in an automated test. The third part of Table 9

Table 9. Comparison of TestOptimizer with PMD using AVQ values.

| Script set ID | AVQ for PMD | AVQ for TestOptimizer, signatureLevel = 0 | AVQ for TestOptimizer, signatureLevel = 1 | AVQ for TestOptimizer, signatureLevel = 2 |
|---|---|---|---|---|
| Raw data | | | | |
| 1 | 320 | 363 | 176 | 98 |
| 2 | 502 | 1421 | 798 | 181 |
| 3 | 324 | 197 | 183 | 66 |
| 4 | 700 | 652 | 312 | 120 |
| 5 | 592 | 438 | 288 | 158 |
| Truly relevant potential subroutines after manual analysis | | | | |
| 1 | 215 | 233 | 172 | 96 |
| 2 | 411 | 671 | 798 | 162 |
| 3 | 179 | 154 | 179 | 63 |
| 4 | 312 | 425 | 312 | 112 |
| 5 | 282 | 259 | 284 | 155 |
| Relative ratio of relevant potential subroutines | | | | |
| 1 | 67,2% | 64,2% | 97,7% | 98,0% |
| 2 | 81,9% | 47,2% | 100,0% | 89,5% |
| 3 | 55,2% | 78,2% | 97,8% | 95,5% |
| 4 | 44,6% | 65,2% | 100,0% | 93,3% |
| 5 | 47,6% | 59,1% | 98,6% | 98,1% |

gives the *Relative ratio of relevant potential subroutines*. The same organization applies to Table 10, where the respective ASQ values are presented.

Regarding the configuration of TestOptimizer server, for script sets 1, 3, 4 and 5, lengthWeight was set to 0.9 and testCountWeight to 0.1. For script set 2, lengthWeight was set to 0.95 and testCountWeight to 0.05. In addition, chromosomeMode was set to CHROMOSOME_AUTO and iterationsCount to 5. The collected data considered only those potential subroutines where $|p| >= 3$. Regarding the configuration of PMD, we used the standard settings from Eclipse (Kepler edition 2016). The minimal length of common code blocks was set to 3.

From evaluating the data in Table 9 and the feedback from the manual analysis and corrections, several conclusions can be made:

(i) Without further analysis of the relevance of the potential subroutines in the context of the automated tests (see the Raw data section of Table 10), the ASQ data show that for analyses at signatureLevel = 0, TestOptimizer performed better, but for more detailed analysis (signatureLevels 1 and 2) PMD returned better sets of potentially reusable subroutines.
Nevertheless, after the expert manual analysis of which potentially reusable subroutines identified by both solutions are truly relevant for automated test optimization (see the "*Truly relevant potential subroutines after manual analysis*" sections of Tables 9 and 10), several conclusions can be made:

(ii) The setting signatureLevel = 2 was too specific to be efficient. At this level we also analyze the specific testing data used in the actions. This data typically

Table 10. Comparison of TestOptimizer with PMD using ASQ values.

| Script set ID | ASQ for PMD | ASQ for TestOptimizer, signatureLevel = 0 | ASQ for TestOptimizer, signatureLevel = 1 | ASQ for TestOptimizer, signatureLevel = 2 |
|---|---|---|---|---|
| Raw data | | | | |
| 1 | 40 | 60,5 | 35,3 | 18,9 |
| 2 | 167,3 | 355,3 | 199,5 | 90,5 |
| 3 | 36 | 49,3 | 43,5 | 24,8 |
| 4 | 140 | 163 | 122 | 60 |
| 5 | 74 | 109,5 | 48 | 39,5 |
| Truly relevant potential subroutines after manual analysis | | | | |
| 1 | 26,9 | 38,8 | 34,7 | 18,4 |
| 2 | 137 | 167,8 | 199,5 | 81 |
| 3 | 19,8 | 38,5 | 42,8 | 23,3 |
| 4 | 104 | 106,3 | 122 | 56 |
| 5 | 47 | 64,8 | 47,3 | 38,8 |
| Relative ratio of relevant potential subroutines | | | | |
| 1 | 67,2% | 64,2% | 98,1% | 97,4% |
| 2 | 81,9% | 47,2% | 100,0% | 89,5% |
| 3 | 54,9% | 78,2% | 98,5% | 93,9% |
| 4 | 74,3% | 65,2% | 100,0% | 93,3% |
| 5 | 63,5% | 59,1% | 98,6% | 98,1% |

differs in various potential subroutines in the analyzed abstracted test scripts. Thus, at this level, the system identifies fewer potential subroutines.

(iii) Based on the feedback from the expert manual analysis and correction, signatureLevel 0 was found to be too abstract to derive common subroutines in the code efficiently. At this level, the system identifies many potential subroutines, but significantly fewer of these are truly relevant for final common subroutines. As the experiment results showed, signatureLevel 0 is suitable for identifying potential common subroutines at a high level, but not for practical test code refactoring suggestions.

(iv) The setting signatureLevel = 1 seems to be the best level of abstraction at which to analyze the test scripts with regard to the AVQ and ASQ values. At this level, the TestOptimizer yields better results than the PMD.

(v) Concerning the specific context and structure of automated test code, at signatureLevels 1 and 2, the TestOptimizer solution provides more relevant results than does the general search for common subroutines by the PMD (see Tables 9 and 10, section "Relative ratio of relevant potential subroutines").

These results show that TestOptimizer, which is specifically tailored to the context of test automation code, identifies more relevant potential common subroutines. In contrast, the PMD simply reports on the number of sequences, which were less relevant from an automated test semantic viewpoint.

To test the practical usability of the solution, we compared independent manual analyses of potential common subroutines independently performed on three sets of the automated test scripts with the results provided by the TestOptimizer server. In this test, we were interested in two aspects:

(1) The relevance of the potential common subroutines found;
(2) The gain in time saved when using automated analysis via the TestOptimizer server.

The independent manual analysis of the test scripts was performed by a group of 5 senior and 25 junior Java developers, who all had equivalent training in Selenium WebDriver and test code refactoring principles. During this manual analysis, the participants were allowed to use any refactoring and code duplication search tools to aid the process, except TestOptimizer.

The results are presented in Table 11.

Table 11. Comparison of automated analysis by TestOptimizer server with independent manual analysis.

| Script set ID | ASQ for performed manual analysis | ASQ for TestOptimizer, signatureLevel = 1 | ASQ for TestOptimizer, signatureLevel = 1, after manual revision | Average time spent by developers performing the manual analysis [hours] | Time spent using Test Optimizer [hours] | Time saved by using Test Optimizer [percentage] |
|---|---|---|---|---|---|---|
| 1 | 38,4 | 35,3 | 34,7 | 3,2 | 0,9 | 71,88% |
| 2 | 218,1 | 199,5 | 199,5 | 4,1 | 1,5 | 63,41% |
| 3 | 30,1 | 43,5 | 42,8 | 1,7 | 0,6 | 64,71% |

In the Time spent using TestOptimizer column, the server computation time is not included. From this experiment, we can make the following conclusions:

(i) For the cases used in the experimental study, the relevance scores (expressed by ASQ) of both the manual analysis and the TestOptimizer results are comparable.
(ii) Using TestOptimizer server significantly reduced the time required (by 65% on average).

The measured times are in hours; nevertheless we need to consider that TestOptimizer is particularly useful when repeating test set refactoring tasks. The time savings accumulate during a project and finally result in savings sufficiently significant to justify employing the proposed solution.

### 3.3. *Discussion*

Because the TestOptimizer is designed to respect the specific context of FE automated test scripts, its results when searching for the potential common subroutines are better than approaches that use common universal tools to search for these subroutines. When we compared the TestOptimizer results with a manual analysis, significant time savings were achieved. Nevertheless, TestOptimizer has some limitations and there are some threats to the validity of the experimental results. We start with the limitations of the TestOptimizer solution.

The system's efficiency in detecting potential common subsequences can vary across different sets of automated test scripts. To obtain objective results, we used real automated test scripts from five selected projects in our experiments. Generally, TestOptimizer performs best for:

(i) Larger test automation projects (in which there are larger sets of automated test scripts);
(ii) Recorded automated tests or automated tests programmed in a non-structured, naive style;
(iii) Automated tests that have been planned to run repeatedly over long time spans and where (due to potential maintenance overhead) better structuring is needed; and
(iv) Whenever a need for repeated refactoring of the set of automated tests exists.

The test automation code that can be analyzed and converted to signatures depends on the configuration of the Converter module. This limits TestOptimizer to test automation frameworks and languages for which a respective Converter is implemented. On the other hand, this approach gives TestOptimizer the flexibility to be applied on various types of FE based automated tests by creating a suitable Converter.

The processing times limit the solution to batch processing: a set of automated test scripts are sent to TestOptimizer server, the analysis is performed as a batch job, and the user asks TestOptimizer server whether the results are ready to download. For the analyzed test sets in our experiments, the run times extend up to 30 min. For this reason, TestOptimizer is not suitable for real-time analysis of test automation code in the current prototype. Nevertheless, as we discussed with several test automation developers, batch processing is suitable for the intended use case. Real-time processing was discussed as "nice-to-have" feature, but not as a necessity.

### 3.4. *Threats to validity*

Several threats to the validity of the performed experiments can be identified:

The manual analysis of the true relevance of the identified potential subroutines was performed by two expert test-automation-code developers. Because we had a strong interest in finding the best configuration for TestOptimizer, we conducted this analysis in the most objective way. Despite that, subjective factors could have played a role. We estimate this factor maximally to 10% of the potential subroutines identified as non-relevant.

The real test automation code used in the experiments was been recorded in the Selenium IDE and exported to Java using the Selenium WebDriver. For different languages and recording styles, the efficiency of the solution could be different.

The group of 30 test automation developers used in the evaluation of the solution efficiency can be considered sufficiently extensive; nevertheless, 25 of these developers were junior test automation developers. The task of identifying potentially reusable subroutines in the test code was appropriate for their expertise level, but it can be assumed that senior test code developers would be faster at performing this activity. When we compared the performance of the senior developers with the junior developers in the group, the senior developers produced more relevant potential common subroutines in less time (on average by 30%). Nevertheless, the results were still significant.

These first results encourage us to continue with the experiments and evolution of the TestOptimizer solution. In future validations, we plan to use more sets and various types of automated test scripts. In addition, to test the efficiency of the TestOptimizer compared to a manual refactoring process, we plan to conduct another set of experiments with a group of more senior developers.

## 4. Related Work

Reduction of potential duplication in test automation code by creating reusable objects is area that has been investigated from various points of view. On structural and architectural levels, over the past two decades, several proposals such as [11, 21–23] have presented frameworks that address reusability issues. In the latest trends, an object character recognition (OCR) approach is being explored as an alternative approach to reduce the maintenance requirements of test scripts, for instance [24]. These proposals can be used as inspiration when creating a new test automation framework or scripting architecture. Generally, designing a framework or architecture that directly addresses reusability issues implies an initial effort to define the proper structure to support reusability. Our aim differs in this regard — we analyze simply structured automated test scripts, created mainly by recording or by descriptive programming in a naive style.

The reusability aspect has also been assessed and discussed for the most established test automation platforms such as Test Complete, HP QuickTest pro (now called MicroFocus Unified Functional Testing) and Selenium [25, 26]. Despite the fact that each of these platforms provides an individual level of support for reusable objects in test scripts, none have implemented a direct solution to reduce duplication in previously recorded test scripts.

A related relevant question is how to estimate the reusability of test components for automated testing. This question inspired Kaner [5] who proposed a method based on the "return on investment" model to estimate the minimum reusability of an automated test component and inspired Kan [27] to propose a method for estimating the potential reusability of test automation components. Finally, it inspired us [28] to propose a model of test automation architecture in which the potential reusability of code is estimated during the design phase. TestOptimizer can be also indirectly used to estimate the potential reusability ratio in analyzed test automation code, nevertheless that is not the primary use case of the solution.

Generally, identification of repetitive code is covered much more intensely at the unit test refactoring level [29–32] than on the automated test level based on SUT FEs. At the unit test refactoring level, we can find individual reports and case studies, for instance [33].

It is worth mentioning that on general level, redundancy of the test code can be also potentially decreased by keeping and maintaining of the SUT model and generation of up-to-date test cases, which is intensely investigated area. As a few examples, we can give [34–37]. Also these approaches do not address the use case of the presented TestOptimizer solution.

In the area of processing recorded automated test scripts, little work exists. Among recent works, the BlackHorse project [14] is definitively relevant to our proposal. The goal of the BlackHorse project is similar: to record tests and then reduce their brittleness. Nevertheless, there are several major differences in the BlackHorse project compared to the approach used in TestOptimizer regarding both implementation and in the general intended use case. BlackHorse uses a proprietary recording tool to capture test traces; then, the test traces are converted to Java code to minimize the effects of changes in the SUT FE and obtain more stable scripts. In contrast, TestOptimizer analyzes already created test scripts and identifies potential common subroutines, but only makes suggestions to the developers, and it is more platform independent. By developing a suitable Converter module, TestOptimizer can be used to analyze automated test scripts in a specific language or test automation APIs.

The test-trace concept used in BlackHorse is conceptually similar to the signature concept used in TestOptimizer; nevertheless, the aims and implementations of these two concepts differ. BlackHorse uses the test trace output from test script recordings as input to produce the final Java test script code, while TestOptimizer uses the signatures to identify potential common test steps from a business logic viewpoint, abstracting the results from any particular source code notation. Because of these significant use case differences, we decided not to compare TestOptimizer with BlackHorse.

An alternative approach to increase the reliability of automated test sets was presented in [38]. Based on an analysis of a SUT FE, the method determines whether the result of an automated test should be inspected manually by a human tester. In this method, a previous baseline version of the SUT FE is compared with a new version, and the differences found are used in the analysis.

Another area conceptually similar to TestOptimizer use case is source code refactoring. Here, a number of previous solutions exist; for example, [39–41]. Code refactoring tools are designed to work with general source code and they do not reflect the specifics of automated test scripts, where two different fragments of test script source code can perform practically the same action in a SUT FE. Nevertheless, this type of tool is comparable to the functionality of TestOptimizer. From the commonly available solutions, we used the copy-paste-detector (CPD) [42] for comparison with TestOptimizer in the presented experiments. CPD is part of the PMD Eclipse plugin project [20] distribution and uses the Karp–Rabin algorithm for string matching.

Regarding the algorithms used, for searching for common subsequences in two strings, the relevant initial algorithms were published some 40 years ago. For instance, an algorithm that solves the problem in quadratic time and linear space was presented by [43, 44]. In our problem, we search for common subroutines in larger sets of strings, which is principally an NP-complete problem [45] in which genetic algorithms obtain satisfactory results [15, 17]. Inspired by this report, we used the genetic algorithm for the LCS problem in the TestOptimizer Solver component.

## 5. Conclusion

Proposed approach addresses one of the key issues of the SUT front-end test automation, which is extensive maintenance of recorded or naively structured automated test scripts. From an economic viewpoint, letting the test engineers to record the tests decrease the test creation costs, whilst proposed automated identification of refactoring opportunities decrease the test optimization costs and as a result the maintenance costs of the test set.

A novelty of the proposed approach lay in a post-processing method based on abstraction of the analyzed tests that enables test engineers to identify truly relevant potential reusable subroutines with more accuracy than the common tools for code refactoring (PMD for instance). Together with this, the proposed approach does not imply any dependency of the created test automation code on an external library or framework.

As shown by the experiment results, the proposed TestOptimizer solution has potential for reducing the maintenance effort in projects where large sets of automated test scripts are recorded that need to be updated manually to improve their stability from a potential maintenance perspective. Automated suggestion of potential common subroutines can help test automation developers analyze the possible refactoring opportunities and can save time finding and assessing potentially reusable objects in the script code. This time savings is relevant despite the fact that TestOptimizer's suggestions must be implemented by the developer (100% relevance is, due to the nature of the problem, practically unreachable). We tracked the resources required to perform the given tasks during our experiments and concluded that using TestOptimizer (i) we spared the time required to create a proposal for reusable test objects and were able to gain time in test development by using script recording (rather than descriptive programming), (ii) the time required to create object descriptors remained the same compared to the pure test recording method, and (iii) we could reduce senior staff costs of a project because tests can be recorded by junior developers; the senior staff are required only to refactor using the TestOptimizer or to review the refactored code. Because the TestOptimizer system respects the specific context of FE automated test scripts, it yields more relevant results compared to PMD (or other common universal tools used to search code for common subroutines).

In our proposal, we use the signatureLevel concept, which specifies the detail of the performed search for the potential common subroutines. Here, signatureLevel 1, which specifies the actions and elements (but does not reflect particular testing data) seems to be the most efficient level of abstraction.

Regarding the applicability of the solution, in addition to semi-automated optimization of recorded test scripts, this solution can be also applied to scripts created by descriptive programming in a naive style, when parts of the code with similar functions are likely to be repeated in the scripts. For more advanced test automation architectures, the TestOptimizer can work as an auditing mechanism, assessing the level of redundancy in a particular test code set.

The proposed solution is built upon a scalable architecture organized as a processing pipeline; therefore, it flexibly supports extensions and adjustments of the analysis process. By creating plugins, the solution can be connected to various IDEs used for test automation (e.g. Eclipse, MicroFocus Unified Functional Testing, Selenium IDE, and others). By adding appropriate converters, the system can process a variety of scripting languages and test automation APIs whereas the principles of the analysis and suggestions remain the same.

TestOptimizer provides only suggestions about potential common subroutines for the submitted scripts. Therefore, it does not include a library or framework that the test automation code is dependent on to work.

Currently we are focusing on few areas for future versions of the solution. We are currently developing a further extension of the abstraction of test scripts to allow processing of more advanced Java features (such as the lambda functions in Java 8). Moreover, we have also focused on continuously improving the selection of prospective tests, which promises better results. Another point of improvement is to optimize the performance optimization of the algorithms used, which could possibly allow TestOptimizer server to response more promptly to submitted requests.

## Acknowledgments

This research is conducted as a part of the project TACR TH02010296 Quality Assurance System for Internet of Things Technology. The research was supported by internal grant of CTU in Prague SGS17/097/OHK3/1T/13.

## References

1. M. Bures, Automated testing in the Czech Republic: The current situation and issues, in *Proc. 15th Int. Conf. Computer Systems and Technologies*, ACM, 2014, pp. 294–301.
2. D. M. Rafi, K. R. K. Moses, K. Petersen and M. V. Mäntylä, Benefits and limitations of automated software testing: Systematic literature review and practitioner survey, in *7th Int. Workshop on Automation of Software Test*, Zurich, Switzerland, 2012, pp. 36–42.
3. S. Berner, R. Weber and R. K. Keller, Observations and lessons learned from automated testing, in *Proc. 27th Int. Conf. Software Engineering*, ACM, New York, 2005, pp. 571–579.
4. J. Kasurinen, O. Taipale and K. Smolander, Software test automation in practice: Empirical observations, in *Advances in Software Engineering*, 2010.
5. C. Kaner, J. Bach and B. Pettichord, *Lessons Learned in Software Testing: A Context-Driven Approach* (Wiley, 2002).
6. M. Dolezel, Images of enterprise test organizations: Factory, center of excellence, or community? in *Proc. 9th Int. Conf. SWQD 2017*, Lecture Notes in Business Information Processing, Vol. 269 (Springer, 2017), pp. 105–116.
7. M. Fewster and D. Graham, *Software Test Automation: Effective Use of Test Execution Tools* (Addison-Wesley Professional, ACM Press Books, 1999).
8. C. Persson and N. Yilmazturk, Establishment of automated regression testing at ABB: Industrial experience report on 'Avoiding the pitfalls', in *Proc. 19th IEEE Int. Conf. Automated Software Engineering*, Washington DC, USA, 2004, pp. 112–121.
9. E. Alégroth and R. Feldt, Industrial application of visual GUI testing: Lessons learned, in *Continuous Software Engineering* (Springer, 2014), pp. 127–140.
10. M. Fewster, Common Mistakes in Test Automation, White paper, Groove consultants, 2001. http://www.agileconnection.com/sites/default/files/article/file/2012/XDD2901 filelistfilename1_0.pdf.
11. D. D. Lonngren, Reducing the cost of test through reuse, in *Proc. IEEE Systems Readiness Technology Conf.*, Salt Lake City, USA (IEEE Press, 1998), pp. 48–53.
12. Selenium WebDriver, http://www.seleniumhq.org/projects/webdriver/.
13. Selenese (Selenium IDE project), http://www.seleniumhq.org/docs/02_selenium_ide.jsp.

14. S. Carino, J. H. Andrews, S. Goulding, P. Arunthavarajah and J. Hertyk, BlackHorse: Creating smart test cases from brittle recorded tests, *Software Quality Journal* **22**(2) (2014) 293–310.
15. B. A. Julstrom and B. Hinkemeyer, Starting from scratch: Growing longest common subsequences with evolution, in *Proc. 9th Int. Conf. Parallel Problem Solving from Nature*, LNCS Vol. 4193 (Springer, 2006), pp. 930–938.
16. Grammar for the Java programming language, Oracle, https://docs.oracle.com/javase/specs/jls/se7/html/jls-18.html.
17. B. Hinkemeyer and B. A. Julstrom, A genetic algorithm for the longest common subsequence problem, in *Proc. 8th Annual Conf. Genetic and Evolutionary Computation* (ACM, 2006), pp. 609–610.
18. P. E. Black, "Rabin-Karp", in *Dictionary of Algorithms and Data Structures*, V. Pieterse and P. E. Black (eds.), 16 November 2009. Available at http://www.nist.gov/dads/HTML/karpRabin.html.
19. R. A. Wagner and M. J. Fischer, The string-to-string correction problem, *J. ACM* **21** (1974) 168–173.
20. PMD plug-in for Eclipse project, http://pmd.sourceforge.net/.
21. L. Kawakami, A. Knabben, D. Rechia, D. Bastos, O. Pereira, R. Pereira e Silva and L. C. V. D. Santos, An object-oriented framework for improving software reuse on automated testing of mobile phones, in *Testing of Software and Communicating Systems*, LNCS Vol. 4581 (Springer, Heidelberg, 2007), pp. 199–211.
22. D. H. Nguyen, P. Strooper and J. G. Süß, Automated functionality testing through GUIs, in *Proc. Thirty-Third Australasian Conf. Computer Science*, Vol. 102, Australian Computer Society, 2010, pp. 153–162.
23. B. Mu, M. Zhan and L. Hu, Design and implementation of GUI automated testing framework based on XML, in *Proc. WRI World Congr. Software Engineering*, Vol. 4, IEEE, 2009, pp. 194–199.
24. E. Alegroth, M. Nass and H. Olsson, JAutomate: A tool for system-and acceptance-test automation, in *IEEE Sixth Int. Conf. Software Testing, Verification and Validation*, IEEE, 2013, pp. 439–446.
25. M. Kaur and R. Kumari, Comparative study of automated testing tools: TestComplete and QuickTest Pro, *Int. J. Comput. Appl.* **24** (2011) 1–7.
26. H. Kaur and G. Gupta, Comparative study of automated testing tools: Selenium, quick test professional and testcomplete, *Int. J. Eng. Res. Appl.* **3**(5) (2013) 1739–1743.
27. H. Kan *et al.*, A method of minimum reusability estimation for automated software testing, *J. Shanghai Jiaotong Univ.* **18** (2013) 360–365.
28. M. Bures, Model for evaluation and cost estimations of the automated testing architecture, in *New Contributions in Information Systems and Technologies*, Advances in Intelligent Systems and Computing, Vol. 353 (Springer, 2015), pp. 781–787.
29. G. Meszaros, *xUnit Test Patterns: Refactoring Test Code* (Pearson Education, 2007).
30. L. Moonen, A. V. D. Bergh and G. Kok, *Refactoring test code*, CWI, 2001, pp. 92–95.
31. E. M. Guerra and T. F. Clovis, Refactoring test code safely, in *Proc. Int. Conf. Software Engineering Advances*, IEEE, 2007, pp. 44.
32. D. Janzen and S. Hossein, Test-driven development: Concepts, taxonomy, and future direction, *Computer* **38** (2005) 43–50.
33. C. McMahon, History of a large test automation project using selenium, in *Proc. Agile Conference*, IEEE, 2009, pp. 363–368.
34. B. Kumar and K. Singh, Testing uml designs using class, sequence and activity diagrams, *Int. J. Innov. Res. Sci. Technol.* **2** (2015) 71–81.

35. T. Yue, S. Ali and L. Briand, Automated transition from use cases to UML state machines to support state-based testing, in *Modelling Foundations and Applications* (Springer, 2011), pp. 115–131.
36. R. Lipka, T. Potuzak, P. Brada, P. Hnetynka and J. Vinarek, A method for semi–automated generation of test scenarios based on use cases, in *41st Euromicro Conf. Software Engineering and Advanced Applications*, IEEE, 2015, pp. 241–244.
37. T. Potuzak and R. Lipka, Interface-based semi-automated generation of scenarios for simulation testing of software components, in *SIMUL*, IARIA, 2014, pp. 35–42.
38. K. Dobolyi, E. Soechting and W. Weimer, Automating regression testing using web-based application similarities, *Int. J. Softw. Tools Technol. Transf.* **13**(2) (2011) 111–129.
39. B. S. Baker, A program for identifying duplicated code, *Computing Science and Statistics* (Springer, 1993), pp. 49–49.
40. T. Pessoa, F. B. e Abreu, M. P. Monteiro and S. Bryton, An eclipse plugin to support code smells detection, in *Proc. Conf. INFORUM '2011*, Computing Research Repository, 2012.
41. A. Hamid, M. Ilyas, M. Hummayun and A. Navaz, A comparative study on code smell detection tools, *Int. J. Adv. Sci. Technol.* **60** (2013) 25–32.
42. CPD copy-paste-detector project, being the part of PMD distribution, http://pmd.sourceforge.net/pmd-4.3.0/cpd.html.
43. D. S. Hirschberg, A linear space algorithm for computing maximal common subsequences, *Commun. ACM* **18** (1975) 341–343.
44. A. V. Aho, J. D. Ullman and D. S. Hirschberg, Bounds on the complexity of the longest common subsequence problem, *J. ACM* **23** (1976) 1–12.
45. D. Maier, The complexity of some problems on subsequences and supersequences, *J. ACM* **25** (1978) 322–336.

# 12 Appendix F: Tapir: Automation Support of Exploratory Testing Using Model Reconstruction of the System Under Test

[A.6] Miroslav Bures, Karel Frajtak, and Bestoun S. Ahmed. Tapir: Automation Support of Exploratory Testing Using Model Reconstruction of the System Under Test. Accepted in *IEEE Transactions on Reliability*, January 2018. (Q1, IF 2.79). Pre-print published at arXiv.org, 1802.07983, https://arxiv.org/abs/1802.07983, 22 Feb 2018.

# Tapir: Automation Support of Exploratory Testing Using Model Reconstruction of the System Under Test

Miroslav Bures, Karel Frajtak, and Bestoun S. Ahmed

***Abstract*—For a considerable number of software projects, the creation of effective test cases is hindered by design documentation that is either lacking, incomplete or obsolete. The exploratory testing approach can serve as a sound method in such situations. However, the efficiency of this testing approach strongly depends on the method, the documentation of explored parts of a system, the organization and distribution of work among individual testers on a team, and the minimization of potential (very probable) duplicities in performed tests. In this paper, we present a framework for replacing and automating a portion of these tasks. A screen-flow-based model of the tested system is incrementally reconstructed during the exploratory testing process by tracking testers' activities. With additional metadata, the model serves for an automated navigation process for a tester. Compared with the exploratory testing approach, which is manually performed in two case studies, the proposed framework allows the testers to explore a greater extent of the tested system and enables greater detection of the defects present in the system. The results show that the time efficiency of the testing process improved with framework support. This efficiency can be increased by team-based navigational strategies that are implemented within the proposed framework, which is documented by another case study presented in this paper.**

***Index Terms*—Model-Based Testing, Web Applications Testing, Functional Testing, System Under Test Model, Generation of Test Cases from Model, Model Reengineering**

## I. Introduction

The contemporary software development market is characterized by the increasing complexity of implemented systems, a decrease in the time to market, and a demand for real-time operation of these systems on various mobile devices [1]. An adequate software system is needed to solve the quality-related problems that arise from this situation. Model-Based Testing (MBT) is a suitable method for generating efficient test cases [2]. For a considerable ratio of the cases, the direct applicability of the method is hindered by the limited availability and consistency of the test basis, which is used to create the model.

To overcome these limitations, we explore possible crossover between common MBT techniques and the exploratory testing approach. Exploratory testing is defined as the simultaneous testing, learning, documentation of the System Under Test (SUT) and creation of the test cases [3]. The exploratory testing approach is a logical choice for testing systems for which a suitable test basis is not available. Even when the test basis is available, and the test cases are created, they can be either obsolete or inconsistent and structured at an excessively high level [4]. Thus, testers employ the exploratory testing technique as a solution for overcoming these obstacles.

The key factors for the efficiency of exploratory testing are consistent documentation of the explored path and exercised test cases [3], [5]. This systematic documentation has the following features:

1) prevents the duplication of (informal) test scenarios that are executed by various testers, which prevents a waste of resources;
2) leads to an exploration of the parts of the SUT that have not been previously explored and tested;
3) improves the efficiency and accuracy of the defect reporting process; and
4) improves the transparency and documentation of the testing process, which is necessary for reporting and making managerial decisions related to a project.

Regarding the weaknesses of the exploratory testing, several issues have been observed in previous reports. Particularly, the low level of structuring of the testing process and a certain *ad hoc* factor can prevent efficient management of the exploratory testing process. These issues may cause problems with the prioritization of the tests, the selection of suitable tests in the actual state of the testing process and the repetition of the exercised tests [3], [6].

Particular SUT exploration strategies are considered important in the exploratory testing process [7]. To efficiently conduct an exploration strategy, the exercised tests must be manually recorded and documented, which can generate additional overhead for the testing team. This overhead can outweigh the benefits gained by a more efficient SUT exploration strategy. Thus, an automation of the tasks related to the documentation of the exercised tests is an option worth exploring.

Teamwork can have a significant role in exploratory testing. The team organization increases the efficiency of the exploratory testing technique in terms of identified defects [5], and it may also prevent repetitive tests and test data combinations. However, these possibilities have only been

M. Bures, Software Testing Intelligent Lab (STILL), Department of Computer Science, Faculty of Electrical Engineering Czech Technical University, Karlovo nam. 13, 121 35 Praha 2, Czech Republic, (email: buresm3@fel.cvut.cz)

K. Frajtak, Software Testing Intelligent Lab (STILL), Department of Computer Science, Faculty of Electrical Engineering Czech Technical University, Karlovo nam. 13, 121 35 Praha 2, Czech Republic, (email: frajtak@fel.cvut.cz)

B. Ahmed, Software Testing Intelligent Lab (STILL), Department of Computer Science, Faculty of Electrical Engineering Czech Technical University, Karlovo nam. 13, 121 35 Praha 2, Czech Republic, (email: albeybes@fel.cvut.cz)

explored in a manual version of the exploratory testing process [5].

These issues represent our motivation for exploring the possibilities of supporting exploratory testing by a suitable machine support method.

This paper summarizes a concept of such a machine support. The paper focuses on SUT exploration strategies and the generation of high-level test cases from the SUT model. The model is reengineered during the exploratory testing activities. The paper presents the Test Analysis SUT Process Information Reengineering (**Tapir**) framework, which guides exploratory testers during their exploration of a SUT. The first objective of the framework is to enable the testing team members to explore the SUT more systematically and efficiently than the manual approach. The second objective of the framework is to automatically document the explored parts of the SUT and create high-level test cases, which guide the testers in the SUT. In this process, the framework continuously builds a SUT model front-end UI. This model is enriched by numerous elements that enhance the testing process.

This paper is organized as follows. Section II presents the principle of the Tapir framework and introduces the SUT model and its real-time construction process. Section III explains the process of guiding the exploratory tester in the SUT based on this model. Section IV describes the setup of the performed case studies. Section V presents the results of these case studies. Section VI discusses the presented results. Section VII discusses threats to the validity of the results, and Section VIII presents related studies. Finally, Section IX provides the concluding remarks of the paper.

## II. Real-time Construction of the SUT Model

This section summarizes the functionality of the Tapir framework and the underlying SUT model.

### *A. Principles of the Tapir Framework*

The aim of the Tapir framework is to improve the efficiency of exploratory testing by automating the activities related to the following:

1) records of previous test actions in the SUT,
2) decisions regarding the parts of the SUT that will be explored in the subsequent test steps, and
3) organization of work for a group of exploratory testers.

The framework tracks a tester's activity in the browser and incrementally builds the SUT model based on its User Interface (UI). Based on this model, which can be extended by the tester's inputs, navigational test cases are generated. The explored paths in the SUT are recorded for the individual exploratory testers. The navigational test cases help the testers explore the SUT more efficiently and systematically, especially when considering the teamwork of a more extensive testing group (typically larger than five testers).

Technically, the Tapir framework consists of three principle parts.

1) **Tapir Browser Extension**: this extension tracks a tester's activity in the SUT and sends the required information to the Tapir HQ component, and it also highlights the UI elements of the SUT in a selected mode (i.e., the elements already analyzed by the Tapir framework or elements suggested for exploration in the tester's next step). The extension also analyzes the SUT pages during the building of the SUT model. Currently, implemented for the Chrome browser.
2) **Tapir HQ**: this part is implemented as a standalone web application that guides the tester through the SUT, provides navigational test cases, and enables a Test Lead to prioritize the pages and links and enter suggested Equivalence Classes (ECs) for the SUT inputs and related functionality. This part constructs and maintains the SUT model. Tapir HQ runs in a separate browser window or tab, thus serving as a test management tool that displays the test cases for the SUT.
3) **Tapir Analytics**: this component enables the visualization of the current state of the SUT model and a particular state of SUT exploration. This part is also implemented as a module of Tapir HQ that shares the SUT model with the Tapir HQ application.

The overall system architecture is depicted in Figure 1.

The Tapir framework defines two principal user roles.

1) **Tester**: a team member who is guided by the Tapir framework to explore the parts of the SUT that have not been previously explored. For each of the testers, a set of navigational strategies and a test data strategy can be set by the Test Lead. The navigational strategy determines a sequence of the SUT functions to be explored during the tests, which is suggested to the tester by the navigational test cases. The test data strategy determines the test data to enter on the SUT forms and similar inputs during the tests. The test data are also suggested to the testers by the navigational test cases.
2) **Test Lead**: senior team member who explores the SUT before asking the testers to perform detailed tests. In addition to the tester's functionalities, the Test Lead has the following principal functionalities.
    a) Prioritization of the pages, links, and UI action elements of the SUT. During the first exploration, the Test Lead can determine the priority of the particular screens and related elements. This priority is saved as a part of the SUT model and subsequently employed in the navigational strategies (refer to Section III-B).
    b) Definition of suitable input test data. During the first exploration, the Test Lead can define ECs for the individual input fields that are detected on a particular page. The ECs are saved to the SUT model and subsequently employed in the process when generating navigational test cases. After the ECs are defined for all inputs of the form on the particular page, the Test Lead can let the Tapir framework generate the test data combinations using an external Combinational Interaction Testing (CIT) module. This module generates a smaller number of efficient test data combinations based

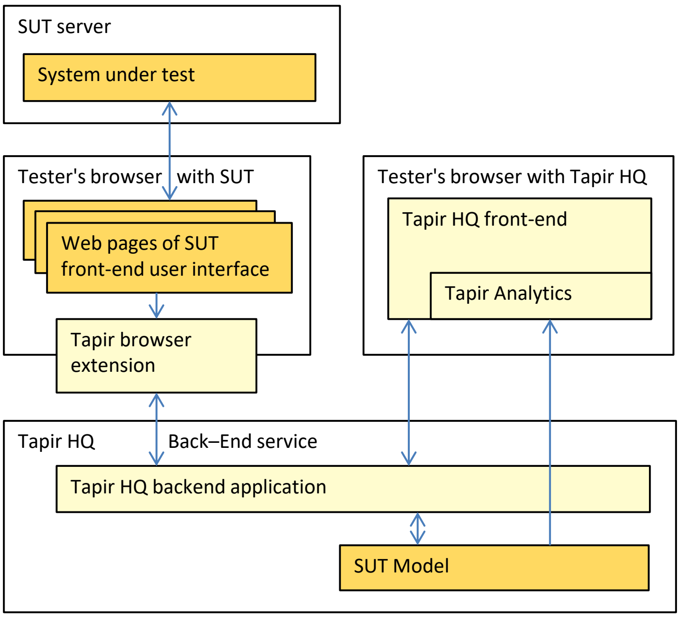

SUT server
System under test
Tester's browser with SUT
Web pages of SUT front-end user interface
Tapir browser extension
Tester's browser with Tapir HQ
Tapir HQ front-end
Tapir Analytics
Tapir HQ
Back–End service
Tapir HQ backend application
SUT Model

Fig. 1. Overall architecture of the Tapir framework

on a set of inputs on the page and the ECs defined for these inputs. In this process, a Particle Swarm Test Generator (PSTG) algorithm [8] is applied. Additional details are provided in Section IV-A.

The Test Lead can dynamically change the navigational and test data strategy of particular testers during the testing to reflect the current state and priorities in the testing process. These strategies are explained in Sections III-B and III-C. The role of the Test Lead is not mandatory in the process. The Tapir framework can be employed by a team of exploratory testers without defining this role. In this case, functions related to prioritization and test data definitions are not available to the team. The framework administrator sets navigational strategies for team members.

### B. SUT Model

For the purpose of systematic navigation by exploratory testers in the SUT, we evolved the following model during our work and experiments with the Tapir framework. $T$ denotes all exploratory testers who are testing the SUT. The set $T$ includes testers $t_1 ... t_n$. A tester can be given the role of Test Lead. The SUT is defined as a tuple $(W, I, A, L)$, where $W$ is a set of SUT pages, $I$ is a set of input elements displayed to the user on the web pages of the SUT user interface, $A$ is a set of action elements (typically `<form>` element submits), and $L$ is a set of link elements displayed to the user.

A Web Page $w \in W$ is a tuple $(I_w, A_w, L_w, \Theta_w, \phi_w, M_w)$, where $I_w \subseteq I$ is a set of input elements, $A_w \subseteq A$ is a set of action elements and $L_w \subseteq L$ is a set of link elements located on page $w$. A Web Page $w$ can contain action elements that can perform actions with more than one form displayed on the page. In our notation, $I_a \subseteq I_w$ contains a set of input elements that are connected to the action elements $a \in A_w$.

On the SUT page $w$, an input element is a data entry field (text field, drop-down item, checkbox or another type of input element defined by the HTML for the forms). An action element triggers the submission of the form, invokes data processing in the SUT and transition to the next page $w_{next}$. The action elements are HTML buttons or elements invoking submit event on the page. The link elements are HTML links, or elements that invoke a transition to the next page $w_{next}$. Typically, the SUT header or footer menu is captured by the link elements.

Then, $range(i)$ denotes the particular data range that can be entered in an input element $i \in I_w$. The $range(i)$ can be either an interval or a set of discrete values (items of a list of values for instance). Then, $range(I_w)$ contains these ranges for the input elements of $I_w$.

$\Theta_w$ is a set of action transition rules $\theta : w \times range(I_w) \times A_w \rightarrow w_{next}$, where $w_{next} \in W$ is a SUT web page that is displayed to a user as a result of exercising the action element $a \in A_w$ with particular input data entered in the input elements $I_a \subseteq I_w$.

$\varPhi_w$ is a set of action transition rules $\phi : w \times L_w \rightarrow w_{next}$, where $w_{next} \in W$ is a SUT web page that is displayed to the user as a result of exercising a link element $l \in L_w$.

Web Pages that are accessible from Web Page $w$ by the defined transition rules $\Theta_w$ and $\varPhi_w$ are denoted as $next(w)$.

$M \subseteq W$ is a set of UI master pages. The Master Page models repetitive components of the SUT user interface, such as a page header with a menu or a page footer with a set of links. The definition of the Master Page is the same as the definition of a Web Page, and the Master Pages can be nested (refer to the Web Page definition). $M_w \in M$ is a set of Master Pages of page $w$, and this set can be empty. Additionally, $w_0 \in W$ represents the home page (defined page, from which the exploratory testers start exploring the SUT) and $w_e \in W$ represents the standard error page displayed during fatal system malfunctions (e.g., the exception page in J2EE applications).

The model of the Web Page and related concepts are depicted in Figure 2. The blocks with a white background depict parts of the model that are automatically reengineered by the Tapir framework during the exploratory testing process. Of these parts, the elements specifically related to the interaction of the tester with the SUT are depicted by the blocks with a dotted background. The blocks with a blue-gray background depict metadata entered by the Test Lead during the exploratory testing process.

As explained in Section II-A, the model is continuously built during the exploratory testing process. The team of testers $T$ contribute to this process, and $W_T$ denotes the SUT pages explored by the whole team while $W_t$ denotes SUT pages explored by the tester $t \in T$. By analogy, $L_T$ ($A_T$) denotes the SUT link (action) elements explored by the whole team and $L_t$ ($A_t$) denotes link (action) elements explored by the tester $t \in T$.

By principle, a link or action element can be exercised additional times during the test exploration process because a page can be repeatedly visited. To capture this fact, $visits(w)_t$, $visits(l)_t$ and $visits(a)_t$ denotes the number of visits of page $w$, link element $l$ and action element $a$ by tester $t$, respectively. Additionally, $visits(w)_T$, $visits(l)_T$ and $visits(a)_T$ denote the number of visits of the page $w$, link element $l$ and action element $a$ by all testers in the testing team $T$, respectively.

For each input element $i \in I$, the Test Lead can define a set of ECs $EC(i)$, which determine the input test data that shall be entered by the exploratory testers during the tests. When these ECs are not defined, $EC(i)$ is empty. For each $ec(i) \in EC(i)$, if $range(i)$ is an interval, then $ec(i)$ is a sub-interval of $range(i)$; and if $range(i)$ is a set of discrete values, then $ec(i) \in range(i)$.

ECs can be dynamically defined during the exploratory testing process, and certain classes can be removed from the model while other classes can be added to the model, with $\bigcap_{ec(i) \in EC(i)} ec(i) = \emptyset$ for each $i \in I$.

In addition, $data(i)_t$ ($data(i)_T$) denotes a set of test data values that are entered to input element $i$ by tester $t$ (by all testers in testing team $T$) during the testing process.

A set of test data combinations that are entered by tester $t$ for the input elements $I_a \subseteq I_w$ connected to the action element $a$ is denoted as $data(I_a)_t = \{(d_1, \ldots, d_n) \mid d_1 \in data(i_1)_t, \ldots, d_n \in data(i_n)_t, \ i_1 \ldots i_n \in I_a\}$. A set of test data combinations that are entered by all testers in testing team $T$ in the input elements $I_a \subseteq I_w$ connected to the action element $a$ is denoted as $data(I_a)_T = \{(d_1, \ldots, d_n) \mid d_1 \in data(i_1)_T, \ldots, d_n \in data(i_n)_T, \ i_1 \ldots i_n \in I_a\}$.

The Test Lead can also set a priority for selected elements of the SUT model. This priority is denoted as $prio(\mathscr{X})$, $prio(\mathscr{X}) \in \{1 \ldots 5\}$, where five is the highest priority. $\mathscr{X}$ denotes particular web page $w \in W$, link element $l \in L$, or action element $a \in A$.

The presented model is inspired by a web application model proposed by Deutsch *et al.* [9]. In our SUT model, significant changes have been made. We kept and modified the definitions of SUT and its web page $w$. In the main tuple that defines the SUT model, we removed the database and states, and then we distinguish the system inputs as input $I$, action $A$ and link $L$ elements. In the definition of the SUT web page $w$, we distinguish the action $A_w$ and link $L_w$ elements. Consequently, we defined the transitions to the following page according to the action transition rule sets $\Theta_w$ and $\varPhi_w$. The home page $w_0$ is taken from the original model. We also applied a concept of the error page $w_e$; however, we define it differently.

The remaining elements of the model (master pages, team of testers, visits of particular SUT pages and their elements, ECs, test data combinations and element priorities) are completely different and were defined by new elements, and they capture specific features of the exploratory testing problem.

## III. Generation of Navigational Test Cases from the Model

As previously explained, the Tapir framework generates high-level navigational test cases that are aimed at guiding a group of exploratory testers through the SUT. The primary purpose of these test cases is to guide the tester in the SUT. The test cases are dynamically created from the SUT model during the exploratory testing process.

### A. Structure of the Navigational Test Case

The navigational test cases are dynamically constructed from the SUT model for an individual tester $t \in T$ during the exploratory testing process. The navigational test case is constructed for the actual page $w \in W$ visited in the SUT and helps the tester determine the next step in the exploratory testing process. The structure of the navigational test case is described as follows.

1) Actual page $w \in W$ visited in the SUT.
2) $L_w$ (list of all link elements that lead to other SUT pages that are accessible from the actual page). In this list, the following information is given:
   a) $L_w \cap L_t$ (links elements that lead to other SUT pages that are accessible from the actual page previously visited by the particular tester $t$),
   b) $visits(l)_t$ for each $l \in L_w \cap L_t$,
   c) $L_w \cap L_T$ (links elements that lead to other SUT pages that are accessible from the actual page previously visited by all testers in team $T$),
   d) $visits(l)_T$ for each $l \in L_w \cap L_T$, and
   e) $prio(l)$ and $prio(w_l)$for each $l \in L_w$. Link $l$ leads from the actual page $w$ to the page $w_l$.
3) $A_w$ (list of all action elements that lead to other SUT pages that are accessible from the actual page). In this list, the following information is given:
   a) $A_w \cap A_t$ (action elements that lead to other SUT pages that are accessible from the actual page previously visited by the particular tester $t$),
   b) $visits(a)_t$ for each $a \in A_w \cap A_t$,
   c) $A_w \cap A_T$ (action elements that lead to other SUT pages that are accessible from the actual page previously visited by all testers in the team $T$),

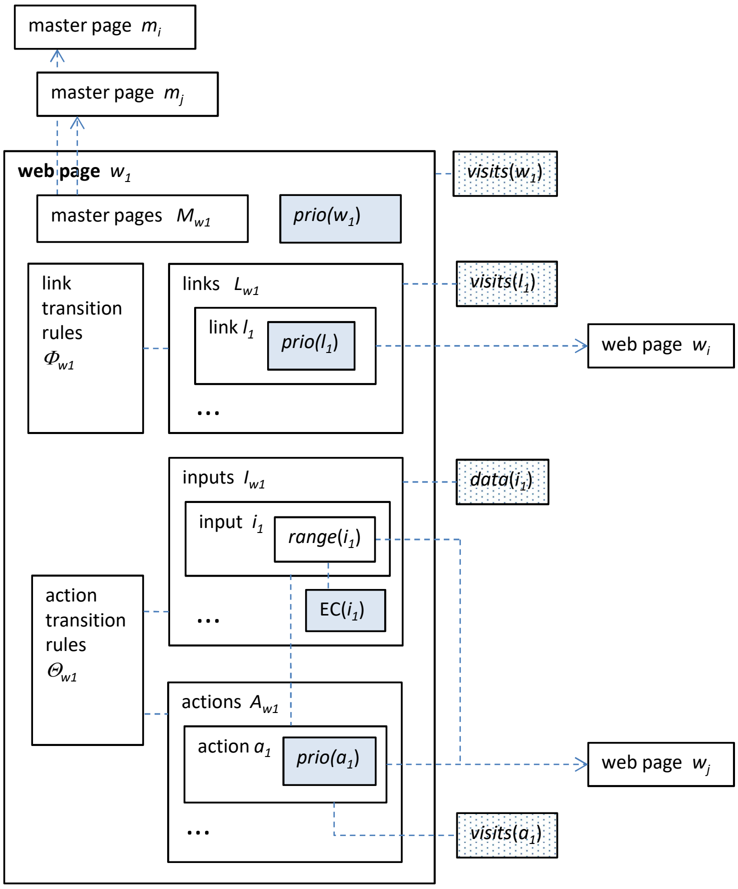


Fig. 2. Model of SUT Web Page and related concepts

   d) $visits(a)_T$ for each $a \in A_w \cap A_T$, and
   e) $prio(a)$ for each $a \in A_w$.

4) Five elements are suggested for exploration in subsequent test steps for each of the navigational strategies assigned to tester $t$ by the Test Lead. These elements are sorted by their ranking, which is calculated by the respective navigational strategies. This ranking gives the tester the flexibility to choose an alternative element if he considers the suggested option to be unsuitable. Because a particular tester can have additional navigational strategies available, these suggestions are displayed for each of the assigned navigational strategies in a separate list, and the tester can choose the optimal strategy according to his personal testing strategy within the navigational strategies set by the Test Lead. The elements that are suggested for exploration are described as follows:
   a) The link $l_{next} \in L_w$ and page $next(w)$ suggested for exploration in the next test step, or
   b) The action element $a_{next} \in A_w$ suggested for exploration in the next test step.
5) For each $a \in A_w$, if $I_a \neq \emptyset$:
   a) $data(I_a)_t$,
   b) $data(I_a)_T$,
   c) for each $i \in I_a$:
      i) $ec(i) \in EC(i)$ is suggested based on the test data strategy (refer to section III-C) set by the Test Lead, and based on this suggestion, tester $t$ can select a particular data value from $ec(i)$ to enter it into the input element $i$ for the actual test;
      ii) $data(i)_t$ (all test data previously entered by the particular tester $i$ to the input element $i$); and
      iii) $data(i)_T$ (all test data previously entered by all testers in the testing team $T$ to the input element $i$).
6) Previous test data combinations entered in $I_a$, which resulted in the error page $w_e$ (e.g., a J2EE exception page ) or a standardized error message (typically, a PHP parsing error message or application specific error message formatted in a unified standard manner) that can be recognized by the Tapir framework.
7) Notes for testers, which can be entered by the Test Lead onto page $w$, all link elements from $L_w$ and all action elements $A_w$. The Test Lead can enter these notes as simple textfields (the notes are not defined in the model in Section II-B).

### B. Navigational Strategies

To create navigational test cases during the exploratory testing process, several navigational strategies can be employed. These strategies are specified in Table I. A navigational strategy determines a principal method that can be applied by a tester to explore the SUT. Most of the navigational strategies can be adjusted using a particular ranking function as specified in Table II. The navigational strategies address the guided exploration of new SUT functions for all testers individually or as a collaborative work by the testing team. The same process is enhanced by navigation driven by priorities of the SUT pages, links and action elements or by regression testing for a defined historical period. This latter strategy is also applicable to retests of defect fixes after a new SUT release.

The navigational strategy determines the SUT user interface elements that are suggested for the SUT page $w$ in the navigational test case (refer III-A). The input of this process is the application context (tester $t$ and related metadata) and actual state of the SUT model that has been specified in sub-section II-B. By the rules specified in Table I and the ranking functions specified in Table II, a list of $l \in L_w$ and $a \in A_w$ , which are sorted by these rules and functions, is created.

The ranking function *ElementTypeRank* is used for both link elements $l \in L_w$ and action elements $a \in A_w$. The *PageComplexityRank* is only used for link elements $l \in L_w$. In the case of action elements, we are not able to determine the exact SUT page after the process triggered by the action element $a$ because test data that have been entered can have a role in determining the page that will be displayed in the next step (refer to the SUT model in sub-section II-B).

In the *ElementTypeRank*, action elements are preferred to link elements because action elements can be expected to contain additional business logic and data operations of the SUT to be explored in the tests (typically, submitting the data by forms on SUT pages).

In the *PageComplexityRank* ranking function, the constants $actionElementsWeight$, $inputElementsWeight$, and $linkElementsWeight$ determine how strongly the individual page action elements, input elements and link elements are preferred for determining the page $w_n \in next(w)$. These elements are suggested for exploration in the tester's next step via the use of a link element that leads to $w_n$. An increase of a particular constant will cause the pages with a higher number of particular elements to be preferred. These constants can be dynamically set by the Test Lead during the exploration testing process.

The constants $actionElementsWeight$ and $linkElementsWeight$ can be set in the range of 1 to 512. The constant $inputElementsWeight$ can be set in the range of 0 to 512. The default value of these constants is 256. Without any change, the pages with a higher number of forms, a higher number of input fields, a higher number of action elements, and a higher number of link elements are considered more complex for the testing purposes, and they are suggested for initial exploration.

In the *PriorityAndComplexityRank*, the priorities of the SUT pages set by the Test Lead have the strongest role in the determination of the suggested next page for exploration. The constant $pagePriorityWeight$ determines the extent of the role of this prioritization. Then, the decision is influenced by the number of input fields, the number of action elements, and the number of link elements. The constants $actionElementsWeight$, $inputElementsWeight$, and $linkElementsWeight$ have the same meaning and function as in the *PageComplexityRank*. The constant $pagePriorityWeight$ can be set in the range of 0 to 512 and its default value is 256. In Tapir HQ component, the Test Lead is provided with a guideline for setting the constants $actionElementsWeight$, $inputElementsWeight$, $linkElementsWeight$ and $pagePriorityWeight$ and the influence of a particular setting.

### C. Test Data Strategies

During the construction of the navigational test cases, test data are suggested for the input elements $I_a \subseteq I_w$ connected to the action elements $a \in A_w$ of the particular page$w \in W$. For this suggestion, (1) test data previously entered by the testers ($data(i)_t$ and $data(i)_T$ for each $i \in I_a$) and (2) ECs defined by the Test Lead ( $EC(i)$ for each $i \in I_a$) are employed.

For this process, the test data strategies described in Table III are available. These test data strategies are specifically designed for different cases in the testing process, such as retesting defect fixes, testing regressions or exploring new test data combinations.

Similar to the case of the navigational strategies, the test data strategies for independent exploration of the SUT by individual testers or team collaboration are available, and they are marked by the postfix "_TEAM" in the name of the test data strategy.

The test data strategy DATA_NEW_RANDOM_TEAM aims to minimize the particular duplicated test data variants entered by multiple testers either by chance or by an improper work organization during the exploration of new test data variants. Another case is intended for testing defect fixes or regression testing, where DATA_REPEAT_LAST, DATA_REPEAT_RANDOM and DATA_REPEAT_RANDOM_TEAM strategies are available to reduce a tester's overhead by remembering the last entered test data. The team strategy DATA_REPEAT_RANDOM_TEAM can improve the efficiency of the process by minimizing particular duplicated test data variants entered by multiple testers during regression testing.

The ECs entered by the Test Lead during his pioneering exploration of the SUT prevent the input of test data that belong to one EC, which exercises the same SUT behavior according to the SUT specification.

The possibility of connecting the Tapir framework to a CIT module (DATA_NEW_GENERATED and DATA_NEW_GENERATED_TEAM strategies) makes the process more controlled and systematic. Only the efficient set of test data combinations are used by the testers to exercise the SUT functions.

## IV. Setup of the Case Studies

To evaluate the proposed framework, compare its efficiency with the manually performed exploratory testing process, and assess the proposed navigational strategies, we conducted three

TABLE I
NAVIGATIONAL STRATEGIES

| Navigational strategy | Rules for element suggestion for page $w$ and tester $t$.<br>Element$\varepsilon$ can be link element $l \in L_w$ or action element $a \in A_w$. | Ranking functions (see Table II) used | Use case |
|---|---|---|---|
| RANK_NEW | $\varepsilon$ satisfying the following conditions:<br>(1) $visits(\varepsilon)_t = 0$, AND<br>(2) ($\varepsilon$ has the highest *ElementTypeRank*($\varepsilon$) OR a page $w_n \in next(w)$ to which $\varepsilon$ leads has the highest *PageComplexityRank*($w_n$)), AND<br>(3) $\varepsilon \in w \in W \setminus M$ are preferred to $\varepsilon \in w \in M$ | *ElementTypeRank*<br>*PageComplexityRank* | Exploration of new SUT functions |
| RANK_NEW_TEAM | As RANK_NEW, (1) modified to:<br>$visits(\varepsilon)_T$ has the minimal value among all $\varepsilon \in L_w$ and $\varepsilon \in A_w$. | *ElementTypeRank*<br>*PageComplexityRank* | Exploration of new SUT functions |
| RT_TIME | $\varepsilon$ satisfying the following conditions:<br>(1) $visits(\varepsilon)_t > 0$, AND<br>(2) time elapsed from the last exploration of $\varepsilon$ by tester $t >$ $LastTime$ constant, AND<br>(3) $\varepsilon \in w \in W \setminus M$ are preferred to $\varepsilon \in w \in M$ | - | (1) Retesting of defect fixes, (2) Regression testing |
| PRIO_NEW | $\varepsilon$ satisfying the following conditions:<br>(1) $visits(\varepsilon)_t = 0$, AND<br>(2) $prio(\varepsilon)$ has the maximal value among all $\varepsilon \in L_w$ and $\varepsilon \in A_w$, AND<br>(3) if $\varepsilon$ is a link element $l \in L_w$, page $w_n \in next(w)$ has the highest *PriorityAndComplexityRank*($w_n$), AND<br>(4) $\varepsilon \in w \in W \setminus M$ are preferred to $\varepsilon \in w \in M$ | *PriorityAndComplexityRank* | Exploration of new SUT functions by priorities set by the Test Lead |
| PRIO_NEW_TEAM | As RANK_NEW, (1) modified to:<br>$visits(\varepsilon)_T$ has the minimal value among all $\varepsilon \in L_w$ and $\varepsilon \in A_w$. | *PriorityAndComplexityRank* | Exploration of new SUT functions by priorities set by the Test Lead |

TABLE II
RANKS USED IN NAVIGATIONAL STRATEGIES

| Rank | Definition |
|---|---|
| *ElementTypeRank*($\varepsilon$) | IF $\varepsilon$ is link THEN *ElementTypeRank*($\varepsilon$) = 1<br>IF $\varepsilon$ is action element THEN *ElementTypeRank*($\varepsilon$) = 2 |
| *PageComplexityRank*($w_n$) | *PageComplexityRank*($w_n$) = ((($\vert\ I_w \ \vert\cdot inputElementsWeight$<br>+$\vert\ A_w \ \vert$)$\cdot actionElementsWeight$ +$\vert\ L_w \ \vert$) $\cdot linkElementsWeight$<br>$I_w \in w_n,\ A_w \in w_n,\ L_w \in w_n$ |
| *PriorityAndComplexityRank*($w_n$) | *PriorityAndComplexityRank*($w_n$) = (((($prio(w_n)\cdot pagePriorityWeight$<br>+$\vert\ I_w \ \vert$)$\cdot inputElementsWeight$ +$\vert\ A_w \ \vert$)$\cdot actionElementsWeight$<br>+$\vert\ L_w \ \vert$) $\cdot linkElementsWeight$<br>$I_w \in w_n,\ A_w \in w_n,\ L_w \in w_n$ |

TABLE III
TEST DATA STRATEGIES

| Test data strategy | Description | Use case |
|---|---|---|
| DATA_REPEAT_LAST | For each $i \in I_a$, suggest the value of $data(i)_t$ used in the last test made by tester $t$ on page $w$.<br>If $data(i)_t = \emptyset$, no suggestion is made. | (1) Retesting of defect fixes, (2) Regression testing |
| DATA_REPEAT_RANDOM | Suggest a randomly selected test data combination from $data(I_a)_t$.<br>If $data(I_a)_t = \emptyset$, no suggestion is made. | Regression testing |
| DATA_REPEAT_RANDOM_TEAM | Suggest a randomly selected test data combination from $data(I_a)_T$.<br>If $data(I_a)_T = \emptyset$, no suggestion is made. | Regression testing |
| DATA_NEW_RANDOM | For each $i \in I_a$:<br>if $EC(i) \neq \emptyset$, suggest a $ec(i) \in EC(i)$, such that $d \notin ec(i)$ for any $d \in data(i)_t$<br>if $EC(i) = \emptyset$, suggest a value $d \in range(i)$, such that $d \notin data(i)_t$ | Exploration of new test data combinations |
| DATA_NEW_RANDOM_TEAM | For each $i \in I_a$:<br>if $EC(i) \neq \emptyset$, suggest a $ec(i) \in EC(i)$, such that $d \notin ec(i)$ for any $d \in data(i)_T$<br>if $EC(i) = \emptyset$, suggest a value $d \in range(i)$, such that $d \notin data(i)_T$ | Exploration of new test data combinations |
| DATA_NEW_GENERATED | The Tapir engine suggests combination, which was not used previously by individual tester $t$. Combination of test data is taken from a pipeline of test data combinations created by a Combination Interaction Testing (CIT) module, connected to the framework by the defined interface (details follow in section IV-A). | Exploration of new test data combinations |
| DATA_NEW_GENERATED_TEAM | As DATA_NEW_GENERATED_TEAM, modified to: combination, which was not used previously by any tester of the testing team $T$ | Exploration of new test data combinations |

case studies. These case studies primarily focus on the process efficiency of the exploration of new SUT functions, and their objective is to answer the following research questions.

***RQ1***: In which aspects the efficiency of the exploratory testing process is increased by the Tapir framework when compared with its manual performance? What is this difference?

***RQ2***: Are there any aspects for which the Tapir framework decreases the efficiency of the exploratory testing process compared with its manual execution?

***RQ3***: Which of the proposed navigational strategies and ranking functions designed for exploration of new parts of the SUT are the most efficient strategies and functions?

***RQ4***: How the previous tester's experience influences defects found in the exploratory testing process supported by the Tapir framework when compared with its manual performance?

Details of the case studies are presented in the following subsections.

### *A. Tapir Architecture and Implementation*

As mentioned in Section II-A, the Tapir framework consists of three principal parts: Tapir Browser Extension, Tapir HQ component, and Tapir Analytics module. A tester interacts with the SUT in a browser window with an installed extension. In the second window, the tester interacts with a Tapir HQ front-end application, which serves as a test management tool. Here, suggestions for the navigational test cases are presented to the tester. The Analytics module can be accessed by the testers, Test Leads, and administrator as a separate application and enables the visualization of the state of the SUT model.

In this section, we provide additional implementation details of the functionality of these framework modules.

**The Tapir Browser Extension** The Tapir Browser Extension is implemented as a web browser plugin. The extension analyzes the current page, intercepts the internal browser events (e.g., page was loaded or redirected, user navigated back, or authentication is required), and it registers event handlers for all links and buttons on the page. All events relevant to the Tapir framework functionality are intercepted and tracked. The browser extension has a functionality to highlight SUT page elements in a mode selected by the Test Lead (elements already analyzed by the Tapir framework or elements suggested for exploration in the tester's next step). Currently, the browser extension is developed for the Chrome browser to cover the highest market share. The extension is implemented in JavaScript. Currently, we are working on a Firefox version of the browser extension, which can be simplified by browser extension portability [1].

**The Tapir HQ** represents the core of the framework functionality. This module receives the events from the browser extensions, constructs the model, constructs the navigational test cases, and presents them to the tester. Tapir HQ is a client application that is implemented as a JavaScript single page application using the ReactJS framework. The server back-end is implemented in .Net C#. When a tester starts testing the SUT, a socket is opened between the Tapir HQ back-end service and Tapir HQ front-end application in the browser to synchronize the data in real time. For this communication, the SocketIO library is employed. Tapir HQ also contains an open interface to a CIT module to import preferred test data combinations (when test data strategies DATA_NEW_GENERATED and DATA_NEW_GENERATED_TEAM are employed; refer to Table III). The interface is based on uploading CSV files of a defined structure or a defined JSON format. The test data combinations are determined for the input elements $I_a \subseteq I_w$ connected to the action elements $a \in A_w$ of the particular page$w \in W$. For the action element $a \in A_w$, the inputs to the CIT module are $I_a$ and $EC(i)$ for each $i \in I_a$. The output from the CIT module is a set of test data combinations. The data combination $(d_{i_1}, ..., d_{i_n})$ is a set of values to be entered in $i_1...i_n \in I_a$. The test data combinations are stored in a pipeline and suggested to the testers to be entered in $I_a$ during the tests. In cases in which $EC(i)$ is not defined by the Test Lead, a random value from $range(i)$ is employed. In this case, the test data combinations are marked by a special flag. For the generation of the test data combinations, PSTG (Particle Swarm Test Generator) algorithm [8] is applied. The process of generating test data combinations can be run by the Test Lead or system administrator for $I_a$ when $EC(i)$ for $i \in I_s \subseteq I_a$. are defined. If previous test data combinations are available, they are overridden by the new set.

To store the SUT model, the MongoDB NoSQL database is employed. The document-oriented NoSQL database was selected because of its efficiency for JSON document processing. Documents can be directly stored in this database. The database with the SUT model is shared by both the Tapir HQ module and Tapir Analytics module. The Tapir HQ back-end service exposes the API to access the database by the individual modules. In the current version of the framework, user authorization is implemented via Google authentication, which is supported by the Chrome browser.

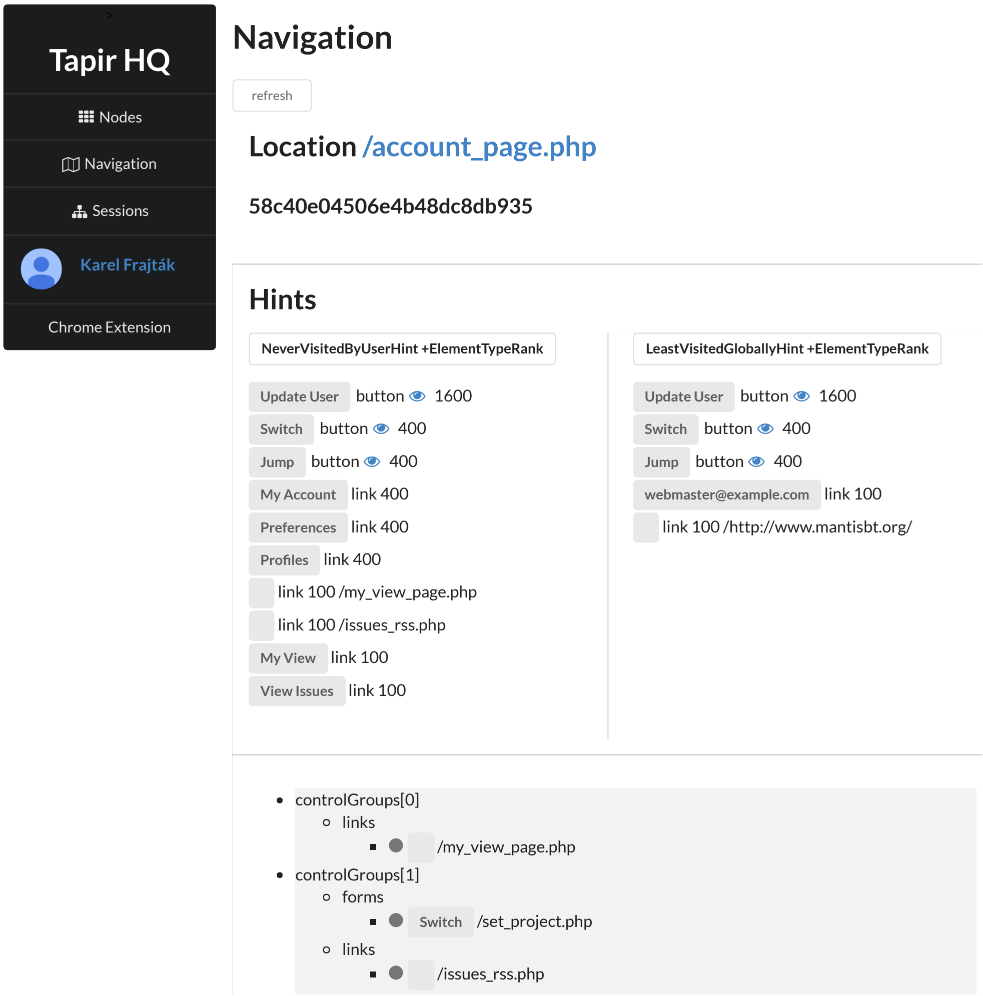
**The Tapir Analytics** module enables users to visualize the current state of the SUT model. The visualization is in the form of a textual representation or a directed graph and a particular state of SUT exploration. This part is implemented as a module of Tapir HQ that shares the SUT model with the Tapir HQ application. The framework administrator grants access rights to this module. This part is implemented in .Net C#. Visualization of the SUT model is implemented in the ReactJS framework.

Figure 3 depicts a tester's navigation support in the SUT via Tapir HQ. The system displays suggested actions to be explored (sorted by ranking functions) by several navigational strategies available to the user. Regarding the format of this article, the view is simplified: test data suggestions and other details are not visible in this sample.

In Figure 4, a corresponding screenshot from the SUT is presented. The Tapir Browser Extension highlights the elements to be explored in the next step and displays the value of the ranking function for these elements.

Figure 5 depicts a sample of the Test Lead's administration of particular SUT page. Prioritization of the SUT pages and elements can be performed in this function.

[1] https://developer.mozilla.org/en-US/Add-ons/WebExtensions/Porting_a_Google_Chrome_extension



Tapir HQ

Nodes

Navigation

Sessions

Karel Frajták

Chrome Extension

Navigation

refresh

Location /account_page.php

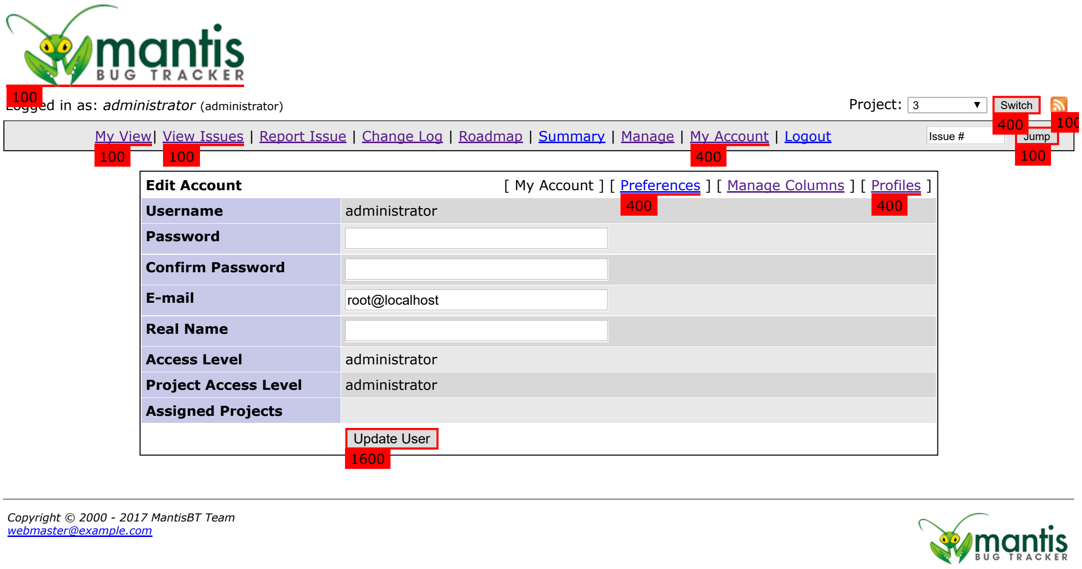
58c40e04506e4b48dc8db935

Hints

| NeverVisitedByUserHint +ElementTypeRank | LeastVisitedGloballyHint +ElementTypeRank |
|---|---|
| Update User button 1600 | Update User button 1600 |
| Switch button 400 | Switch button 400 |
| Jump button 400 | Jump button 400 |
| My Account link 400 | webmaster@example.com link 100 |
| Preferences link 400 | link 100 /http://www.mantisbt.org/ |
| Profiles link 400 | |
| link 100 /my_view_page.php | |
| link 100 /issues_rss.php | |
| My View link 100 | |
| View Issues link 100 | |

- controlGroups[0]
  - links
    - ● /my_view_page.php
- controlGroups[1]
  - forms
    - ● Switch /set_project.php
  - links
    - ● /issues_rss.php

Fig. 3. A sample of testers' navigational support (simplified)



Logged in as: *administrator* (administrator)

Project: 3 Switch

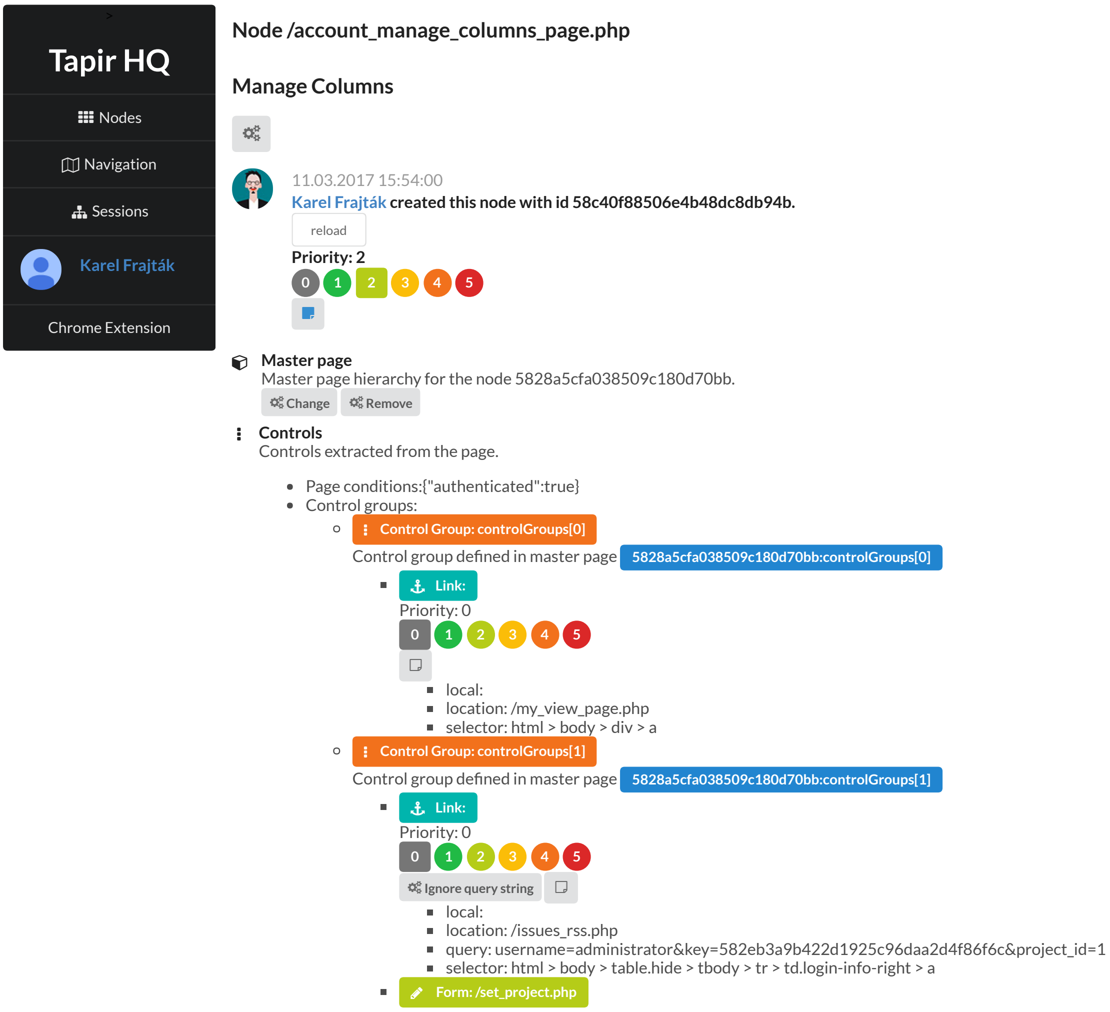
My View | View Issues | Report Issue | Change Log | Roadmap | Summary | Manage | My Account | Logout

Issue # Jump

| Edit Account | [ My Account ] [ Preferences ] [ Manage Columns ] [ Profiles ] |
|---|---|
| Username | administrator |
| Password | |
| Confirm Password | |
| E-mail | root@localhost |
| Real Name | |
| Access Level | administrator |
| Project Access Level | administrator |
| Assigned Projects | |
| | Update User |

*Copyright © 2000 - 2017 MantisBT Team*
*webmaster@example.com*

Fig. 4. SUT with the Tapir Browser Extension highlights



Tapir HQ

Nodes

Navigation

Sessions

Karel Frajták

Chrome Extension

Node /account_manage_columns_page.php

Manage Columns

11.03.2017 15:54:00

Karel Frajták created this node with id 58c40f88506e4b48dc8db94b.

reload

Priority: 2

0 1 2 3 4 5

Master page

Master page hierarchy for the node 5828a5cfa038509c180d70bb.

Change Remove

Controls

Controls extracted from the page.

- Page conditions:{"authenticated":true}
- Control groups:
  - Control Group: controlGroups[0]

    Control group defined in master page 5828a5cfa038509c180d70bb:controlGroups[0]
    - Link:

      Priority: 0

      0 1 2 3 4 5
      - local:
      - location: /my_view_page.php
      - selector: html > body > div > a
  - Control Group: controlGroups[1]

    Control group defined in master page 5828a5cfa038509c180d70bb:controlGroups[1]
    - Link:

      Priority: 0

      0 1 2 3 4 5

      Ignore query string
      - local:
      - location: /issues_rss.php
      - query: username=administrator&key=582eb3a9b422d1925c96daa2d4f86f6c&project_id=1
      - selector: html > body > table.hide > tbody > tr > td.login-info-right > a
    - Form: /set_project.php

Fig. 5. A sample of Test Lead's administration of particular SUT page

Figure 6 depicts a sample from the Analytics module, which consists of the visualization of SUT pages and possible transitions between the pages. Because the graph is usually extensive, it can be arranged by several layouts to obtain additional user comfort. In the sample, compact visualization is depicted.

### *B. Systems Under Test with Injected Defects*

In Case Studies 1 and 2, we employed an open-source MantisBT[2] issue tracker as a SUT for the experiments. The MantisBT is written in PHP and uses a relational database. We modified the source code of the SUT by inserting 19 artificial defects. We accompanied the defective code lines by a logging mechanism for reporting each activation of the defective line code. The details of the injected defects are illustrated in Table IV.

In Case Study 3, we used the healthcare information system Pluto that was developed in a recent software project[3]. The Pluto system is employed by hospital departments for a complete supply workflow of pharmaceutical products and medical equipment. The system is one part of a complex hospital information system. The front-end of the Pluto system is created using HTML, CSS and the JavaScript framework Knockout. The back-end of the system is developed in C# and runs on the .NET platform. In this case, we obtained a history of real defects that were detected and fixed in the software from December 2015 to September 2017. As a SUT for the experiments, we used a code baseline of the system from August 2015. We analyzed 118 defects that were reported from December 2015 to September 2017. During the bug fixing and maintenance phase of the SUT, 72 of the defects were found to be caused by the regression as a result of implemented change requests and other bug fixes. During the initial analysis, we excluded these defects; thus, the final number of defects in the experiment was 48.

We obtained information about the presence of the defects in the SUT code. We accompanied the SUT with a .NET Aspect-based logging mechanism to log the flows calls of the SUT methods during the tests. Using this mechanism, we were able to determine and analyze the defects in the code that were activated by the exercised tests. We automated this analysis

[2] https://www.mantisbt.org/

[3] http://www.lekis.cz/Stranky/Reseni.aspx

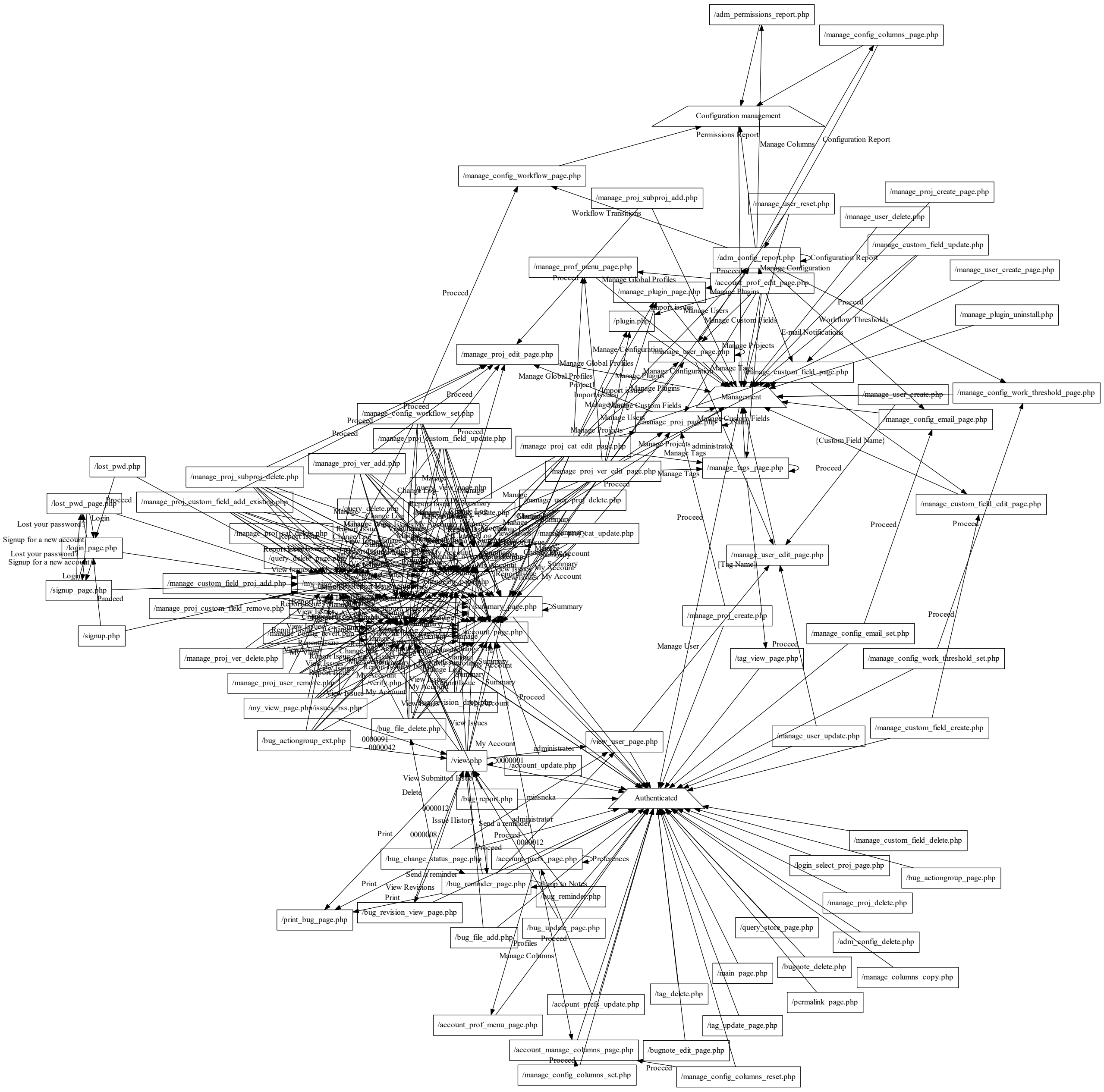


Fig. 6. A sample from Analytics module

by a set of scripts that compare the data of the SUT model recorded by the Tapir framework with these defect activation logs.

### C. Setup of Case Studies

In Case Study 1, we compared the exploratory testing process that was manually performed by individual testers with the exploratory testing process supported by the Tapir framework presented in this paper. The aim of this case study is to answer research questions *RQ1* and *RQ2*. In this case study, we employed the following method.

A group of 54 testers performed exploratory testing in the SUT. The MantisBT issue tracker and inserted artificial defects are employed (refer to Table IV). Each of the testers were allowed to individually act, and they were instructed to perform an exploratory smoke test and explore the maximal extent of the SUT. Exit criteria were left for an individual tester's consideration.

To evaluate the results of this case study, we applied data that were available in the SUT model created by the Tapir framework during the exploratory testing process (for details, refer to Section V-A). A subjective report by individual testers was not used in the evaluation. The testers were divided into

TABLE IV
DEFECTS INJECTED TO THE SYSTEM UNDER TEST

| Injected defect ID | Type | SUT function |
|---|---|---|
| synt_1 | Syntax error | Plugin installation function broken |
| synt_2 | Syntax error | Plugin uninstallation function broken |
| synt_3 | Syntax error | Import issues from XML function broken |
| synt_4 | Syntax error | Adding empty set of users to a project causes system defect of the SUT |
| synt_5 | Syntax error | Setting configuration option with empty value causes system defect of the SUT |
| synt_6 | Syntax error | Config option of float type cannot be created |
| synt_7 | Syntax error | Config option with complex type cannot be created |
| mc_1 | Missing code | Export to CSV is not implemented |
| mc_2 | Missing code | The action “set sticky” in search issues screen is not implemented |
| mc_3 | Missing code | Printing of the issue details is not implemented |
| mc_4 | Missing code | User cannot be deleted |
| mc_5 | Missing code | Bug note cannot be deleted |
| cc_1 | Change in condition | Issue configuration option value cannot be set in database |
| cc_2 | Change in condition | Issue configuration option value in not loaded properly from database |
| cc_3 | Change in condition | Tag with the name ”Tapir” (predefined in the SUT) cannot be deleted |
| var_1 | Wrong set of variable | Language in user preferences is always ”English” and cannot be changed |
| var_2 | Wrong set of variable | User defined columns in issue list cannot be copied between projects |
| var_3 | Wrong set of variable | When adding new bug note, its status cannot be ”private” |
| var_4 | Wrong set of variable | Bug note view status cannot be changed |

two groups.

1) A group of 23 testers manually performed the exploratory testing process. The activity of these testers was recorded by the Tapir framework tracking extension and Tapir HQ Back-End service. The Tapir HQ Front-End application was not available to this group. Thus, navigational support was not provided to its members.
2) A group of 31 testers disjunctive to the previous group performed the exploratory testing process with support provided by the Tapir framework. This group employed the RANK_NEW navigational strategy. Within this strategy, half of this group is randomly selected to use *PageComplexityRank* and the other half of the group is selected to use *ElementTypeRank*. Although DATA_NEW_RANDOM was employed as the test data strategy, the Team Lead did not define any ECs. The testers in this group were explicitly instructed to not use the test data suggestions made by the framework, and their task was to determine which test data to actively enter. The purpose of this task was to equalize the conditions of both groups (the group that manually performs the exploratory testing has no support regarding the test data). No priorities were set for the SUT pages and its elements. The Test Lead did not change any setup during the experiments. The values of the $actionElementsWeight$, $inputElementsWeight$, and $linkElementsWeight$ constants were set to the default value 256.

The participants were differing in praxis in software testing from 0.5 to 4 years. The participants were randomly distributed in the groups. In this case study, we have not employed the team variants of the provided navigational strategies (RANK_NEW_TEAM). In an objective experiment, equivalent team support must be provided for cases in which exploratory testing is manually performed. We have attempted to perform this initial experiment, and an equivalent simulation of the Tapir framework functionality by a human team leader was difficult to achieve. Thus, we evaluate the team versions of the navigational strategies in Case Study 2.

In Case Study 2, we focused on answering the research question *RQ3*.With an independent group of testers, we compared the proposed navigational strategies that primarily focused on exploring new SUT functions. In this study, a group of 48 testers performed exploratory testing in the MantisBT issue tracker with inserted artificial defects (refer to Table IV). All testers used the support of the Tapir framework. The testers were instructed to explore the maximal extent of the SUT. This group was split into four subgroups as specified in Table V.

The participants were differing in praxis in software testing from 0.5 to 4 years. The participants were randomly distributed in the groups. In this case study, ECs were not defined by the Team Lead. In addition, the testers in this group were explicitly instructed to ignore the test data suggestions made by the framework, and their task was to determine which test data to actively enter. No priorities were set for the SUT pages and its elements. The Test Lead did not change any of the setup parameters during the experiment. The values of the $actionElementsWeight$, $inputElementsWeight$, and $linkElementsWeight$ constants were set to the default value 256.

Regarding the strategies that employ prioritization of the page elements and pages (particularly, PRIO_NEW and PRIO_NEW_TEAM), this concept adds extra opportunities to improve the efficiency of the exploratory testing process. A comparable alternative for the proposed navigational strategies is not available at this developmental stage of the Tapir framework. When equivalent prioritization is performed in the manual exploratory testing process, we expect the same increase in testing process efficiency. For these reasons, we have decided to exclude the evaluation of the navigational strategies

TABLE V
PARTICIPANT GROUPS PERFORMING THE CASE STUDY 2

| Group ID | Number of participants | Navigational strategy | Ranking function | Test data strategy |
|---|---|---|---|---|
| 1 | 13 | RANK_NEW_TEAM | *ElementTypeRank* | DATA_NEW_RANDOM_TEAM |
| 2 | 11 | RANK_NEW | *ElementTypeRank* | DATA_NEW_RANDOM |
| 3 | 12 | RANK_NEW_TEAM | *PageComplexityRank* | DATA_NEW_RANDOM_TEAM |
| 4 | 12 | RANK_NEW | *PageComplexityRank* | DATA_NEW_RANDOM |

PRIO_NEW and PRIO_NEW_TEAM from the described case study.

Regarding the test data strategies, each of the individual strategies is practically designed for different use cases (refer to Table III). A comparison can be performed between the strategies designed for an individual tester's guidance and the strategies designed for team exploratory testing, e.g., DATA_NEW_RANDOM versus DATA_NEW_RANDOM_TEAM. Because the presented case study primarily focuses on the efficiency of process exploration for the new SUT functions, a comparison of the strategies DATA_NEW_RANDOM and DATA_NEW_RANDOM_TEAM was included in Case Study 2.

In Case Study 3, we compared the exploratory testing process that was manually performed by individual testers with the exploratory testing process supported by the Tapir framework. In this case study, another SUT with real software defects was employed (refer to SectionIV-B). This case study aims to answer research questions *RQ1*, *RQ2 and RQ4*. The organization of the experiment is described as follows.

A group of 20 testers performed exploratory testing in the Pluto system, with each tester acting individually. The instructions to perform an exploratory smoke test and explore the maximal extent of the SUT were the same instructions provided in Case Study 1. Exit criteria were left for an individual tester's consideration.

A group of ten testers performed the manual exploratory testing process, and their activities were recorded by the Tapir framework. Navigational support of the Tapir HQ was not provided to these testers. Another group of ten testers employed the Tapir framework support. RANK_NEW was used as a navigational strategy. In this group, one randomly selected half of the testers used *PageComplexityRank*, whereas the other half of the testers used *ElementTypeRank*. Regarding the testing data, we selected the same setup as in Case Study 1 to ensure that the conditions of both groups were as equal as possible. Although DATA_NEW_RANDOM was utilized, ECs were not defined by the Test Lead, and the testers were instructed to define their individual testing data (test data suggestions of the Tapir framework were not employed). Element priorities were not applied, and the values of the $actionElementsWeight$, $inputElementsWeight$, and $linkElementsWeight$ constants were set to the default value 256. For objectivity reasons, no team variant of a navigational strategy was utilized (this experiment was the subject of Case Study 2). Experience of the testers varied from 0.5 years to 3 years. We mixed both groups to have the average experience of the testers in each of the groups 1.5.

Regarding the experimental groups, we ensured that all participants had received the equivalent initial training regarding the following software testing techniques: (1) principle of exploratory testing, (2) identification of boundary values, (3) equivalence partitioning, (4) testing data combinations to input in the SUT (condition, decision and condition/decision coverage, pairwise testing and basics of constraint interaction testing) and (5) techniques to explore a SUT workflow (process cycle test).

To evaluate the case studies, we used a set of metrics that are based on the SUT model and activated defects in the SUT code. This set of metrics is defined in Table VI.

In the definitions, $\tau$ represents the total time spent by exploratory testing activity, and it was averaged for all testers in the group and given in seconds. The average time spent on a page is measured using the Tapir framework logging mechanism. The average time to activate a defect (*time_defect*) is calculated as the average total time spent by the exploratory testing process divided by the number of activated defects. When a defect is activated, it occurs during the exploratory testing process. Thus, a tester can notice and report this defect.

## V. Case Study Results

In this section, we present and discuss the results of the performed case studies.

### A. The Results of Case Study 1

Table VII summarizes the comparison between the manual exploratory testing approach and the Tapir framework. The results are based on the data that we were able to automatically collect from the recorded SUT model. In this comparison, the average of the results from the navigational strategy RANK_NEW and ranking functions *ElementTypeRank* and *PageComplexityRank* are provided for the Tapir framework. DATA_NEW_RANDOM was utilized as the test data strategy. $DIFF = (AUT - MAN)/AUT$ is presented as a percentage, where $AUT$ represents the value measured in the case of the Tapir framework, and $MAN$ denotes the value measured in the case of the manual approach. In Table VII, we use the metrics previously defined in Table VI. In the statistics, we excluded excessive lengthy steps (tester spent more than 15 minutes on a particular page) caused by leaving the session open and not testing. In the case of the manual exploratory testing, these excluded steps represented 1.19% of the total recorded steps; and in the case of Tapir framework support, this ratio was 0.72%.

To measure the activated defects, we accompanied the defective code lines by a logging mechanism and reported each activation of the defective line code.

| Metric definition | Explanation | Unit |
|---|---|---|
| $\mid T \mid$ | Number of participants | - |
| *pages* = $\sum_{w \in W} visits(w)_T$ | Total number of explored pages, pages can repeat | - |
| *u_pages* = $\mid W \mid$ | Total number of explored unique pages | - |
| *r_pages* = $\frac{\mid W \mid}{\sum_{w \in W} visits(w)_T} \cdot 100\%$ | Ratio of explored unique pages | % |
| *links* = $\sum_{l \in L} visits(l)_T$ | Total number of link elements explored, elements can repeat | - |
| *u_links* = $\mid L \mid$ | Total number of explored unique link elements | - |
| *r_links* = $\frac{\mid L \mid}{\sum_{l \in L} visits(l)_T} \cdot 100\%$ | Ratio of explored unique link elements | % |
| *actions* = $\sum_{a \in A} visits(a)_T$ | Total number of action elements explored, elements can repeat | - |
| *u_actions* = $\mid A \mid$ | Total number of explored unique action elements | - |
| *r_actions* = $\frac{\mid A \mid}{\sum_{a \in A} visits(a)_T} \cdot 100\%$ | Ratio of explored unique action elements | % |
| *time_page* = $\frac{\tau}{\sum_{w \in W} visits(w)_T}$ | Average time spent on page | seconds |
| *time_u_page* = $\frac{\tau}{\mid W \mid}$ | Average time spent on unique page | seconds |
| *time_link* = $\frac{\tau}{\sum_{l \in L} visits(l)_T}$ | Average time spent on link element | seconds |
| *time_u_link* = $\frac{\tau}{\mid L \mid}$ | Average time spent on unique link element | seconds |
| *time_action* = $\frac{\tau}{\sum_{a \in A} visits(a)_T}$ | Average time spent on action element | seconds |
| *time_u_action* = $\frac{\tau}{\mid A \mid}$ | Average time spent on unique action element | seconds |
| *defects* | Average activated defects logged, activated defects can repeat | - |
| *u_defects* | Average unique activated defects logged | - |
| *time_defect* | Average time to activate one defect, activated defects can repeat | seconds |
| *time_u_defect* | Average time to activate one unique defect | seconds |

TABLE VI
METRICS USED TO EVALUATE CASE STUDIES 1-3

TABLE VII
COMPARISON OF MANUAL EXPLORATORY TESTING APPROACH WITH TAPIR FRAMEWORK: DATA FROM SUT MODEL FOR CASE STUDY 1

| Metric | Manual approach | Tapir framework used | *DIFF* |
|---|---|---|---|
| $\mid T \mid$ | 23 | 31 | - |
| *pages* | 151.8 | 197.9 | 23.3% |
| *u_pages* | 22.2 | 37.7 | 41.0% |
| *r_pages* | 14.6% | 19.0% | 23.1% |
| *links* | 64.7 | 113.2 | 42.9% |
| *u_links* | 21.4 | 44.0 | 51.3% |
| *r_links* | 33.1% | 38.9% | 14.8% |
| *actions* | 24.5 | 59.0 | 58.5% |
| *u_actions* | 9.6 | 28.3 | 66.2% |
| *r_actions* | 39.1% | 47.9% | 18.5% |
| *time_page* | 21.5 | 20.1 | -6.6% |
| *time_u_page* | 146.7 | 105.7 | -38.7% |
| *time_link* | 50.4 | 35.2 | -43.2% |
| *time_u_link* | 152.3 | 90.6 | -68.1% |
| *time_action* | 133.1 | 67.5 | -97.3% |
| *time_u_action* | 340.7 | 140.8 | -141.9% |
| *defects* | 11.1 | 16.6 | 32.8% |
| *u_defects* | 4.6 | 6.6 | 31.1% |
| *time_defect* | 292.8 | 240.5 | -21.7% |
| *time_u_defect* | 713.8 | 601.3 | -18.7% |

Of the 19 inserted artificial defects, three defects, *synt_6*, *synt_7* and *mc_2*, were not activated by any of the testers in the group supported by the Tapir framework, which yields a 15.8% ratio. In the case of the manually performed exploratory testing, the defect *mc_2* was activated by one tester from the group.

A comparison was performed between the manual exploratory testing approach and the exploratory testing approach supported by the Tapir framework to determine the efficiency of the potential to detect injected artificial defects in the SUT, and it is depicted in Figure 7. For particular injected defects, an average value for the number of times one tester activated the defect is presented. Value 1 indicates that all the testers in the group have activated the defect once. For example, value 0.5 indicates that 50% of the testers in the group have activated the defect once. The injected artificial defects are introduced in Table IV.

Figure 8 provides details on the average time spent on a SUT page by testers using the manual exploratory testing approach and testers using the Tapir framework support.

Figure 9 provides details on another comparison of the unique inserted defects that were activated during the activity of individual testers in both groups.

### *B. The Results of Case Study 2*

Table VIII presents a comparison of different Tapir framework navigational strategies (refer to Table I) based on the data that were automatically collected from the SUT model. In

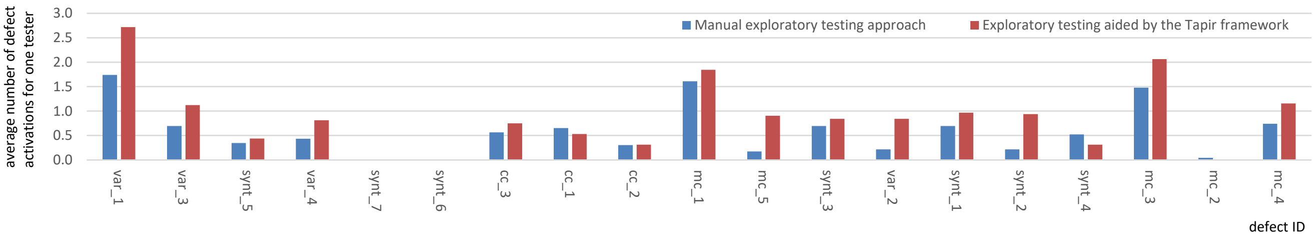


Fig. 7. Potential of manual exploratory testing and Tapir framework approach to detect injected defects in the SUT in Case Study 1

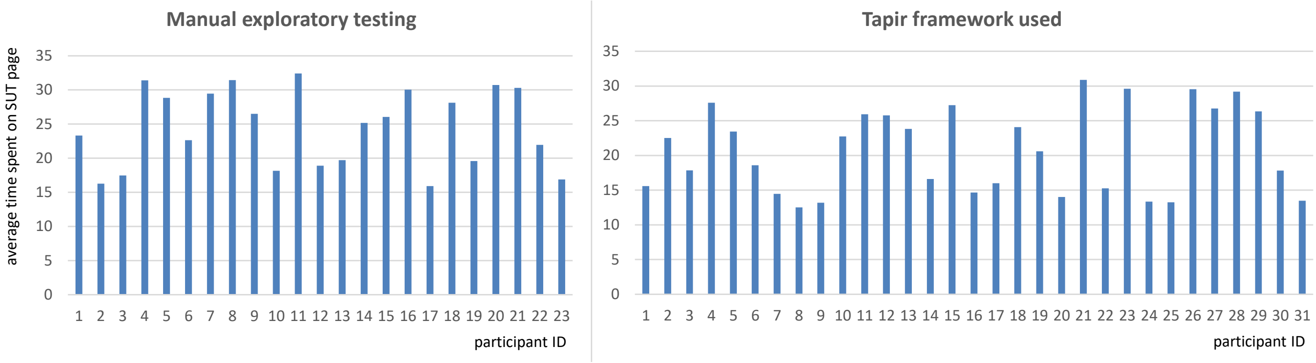


Fig. 8. Average times spent on SUT pages by testers using manual approach and Tapir framework

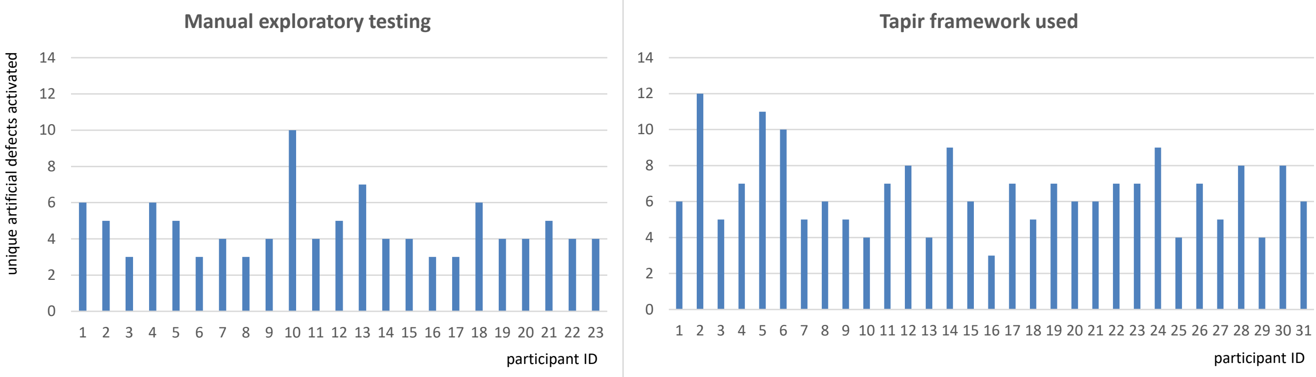


Fig. 9. Unique inserted defects activated by testers using manual approach and Tapir framework

Table VIII, we use the metrics that were previously defined in Table VI. In this case study, we excluded steps longer than 15 minutes and assumed that such length was caused by leaving the session open and not testing.

In Group 1, the excluded steps represented 0.54%, 0.87%, 0.49% and 0.76% of the total recorded steps in Groups 2, 3, and 4, respectively.

The relative differences between the results of the Case Study 2 groups are presented in Table IX.

### C. The Results of Case Study 3

Table X presents a comparison of the manual exploratory testing approach with the Tapir framework for the experiment with the Pluto system. Compared with Case Study 1, the real defects were present in the SUT code in this case study (refer to Section IV-B). In the Tapir framework, the RANK_NEW navigational strategy with ranking functions *ElementTypeRank* and *PageComplexityRank* were employed. DATA_NEW_RANDOM was utilized as the test data strategy.

The data collection method was the same as the data collection method in Case Study 1, including the meaning of *DIFF* in Table X. To evaluate the experiment, the metrics defined in Table VI are employed. In the statistics, we excluded test steps longer than 15 minutes on a particular page. We considered the possibility that the session was opened but testing was not performed. In the case of the manual exploratory testing, these excluded steps represented 1.26% of the total recorded steps, and in the case of Tapir framework support, this ratio was 0.64%.

To measure the activated defects, we logged the flow calls

TABLE VIII
COMPARISON OF TAPIR NAVIGATIONAL STRATEGIES BASED DATA FROM SUT MODEL

| Metric | Group 1 | Group 2 | Group 3 | Group 4 |
|---|---|---|---|---|
| Navigational strategy | RANK_NEW_ TEAM | RANK_NEW | RANK_NEW _TEAM | RANK_NEW |
| Ranking function | *ElementTypeRank* | *ElementTypeRank* | *PageComplexityRank* | *PageComplexityRank* |
| Test data strategy | DATA_NEW_ RANDOM_TEAM | DATA_NEW_ RANDOM | DATA_NEW_ RANDOM_TEAM | DATA_NEW_ RANDOM |
| $\| T \|$ | 13 | 11 | 12 | 12 |
| *pages* | 224.0 | 211.7 | 233.6 | 206.2 |
| *u_pages* | 47.1 | 39.0 | 51.4 | 42.2 |
| *r_pages* | 21.0% | 18.4% | 22.0% | 20.5% |
| *links* | 131.8 | 104.2 | 142.5 | 118.0 |
| *u_links* | 54.4 | 37.9 | 58.1 | 41.1 |
| *r_links* | 41.3% | 36.4% | 40.8% | 34.8% |
| *actions* | 69.3 | 57.5 | 75.1 | 62.7 |
| *u_actions* | 34.8 | 24.6 | 38.3 | 29.6 |
| *r_actions* | 50.2% | 42.8% | 51.0% | 47.2% |
| *time_page* | 17.8 | 19.1 | 19.0 | 21.4 |
| *time_u_page* | 84.6 | 103.6 | 86.3 | 104.7 |
| *time_link* | 30.2 | 38.8 | 31.1 | 37.4 |
| *time_u_link* | 73.2 | 106.6 | 76.3 | 107.5 |
| *time_action* | 57.5 | 70.3 | 59.0 | 70.4 |
| *time_u_action* | 114.5 | 164.3 | 115.8 | 149.2 |
| *defects* | 19.7 | 16.8 | 20.3 | 17.3 |
| *u_defects* | 8.6 | 6.9 | 9.1 | 7.4 |
| *time_defect* | 202.2 | 240.6 | 218.4 | 255.3 |
| *time_u_defect* | 463.3 | 585.8 | 487.2 | 596.9 |

of the SUT method during the tests by the added logging mechanism. Then, we automatically compared these logs with the recorded SUT model to determine which defects were activated during certain test steps. In this case study, all defects were activated by both groups. The average number of defects activated by both groups are depicted in Figure 10.

In this figure, we depict the average number of times one tester activated the defect (e.g., the value 0.5 indicates that 50% of the testers in the group has activated the defect once, and the value 2 indicates that all testers in the group activated the defect twice).

In Figure 11 we present defect activation data related to the experience of the testers. In the graphs, individual groups of columns present the data for individual testers. On the x-axis, the praxis of the tester in years is captured. In the graph, we present *defects*, *u_defects*, *time_defec*t and *time_u_defect* values.

## VI. DISCUSSION

To evaluate Case Studies 1 and 3, we analyze the data in Table VII and X. In the case of the MantisBT system, which is the subject of Case Study 1 (Table VII), we note that using the Tapir framework causes the testers to explore larger extents of the SUT compared with the manually performed exploratory testing. This effect can be observed for the total and unique SUT pages (values *pages* and *u_pages*), where Tapir support leads the testers to explore 23.3% more pages and 41% additional unique SUT pages. For the total link elements and unique link elements (*links* and *u_links*, respectively) and the total and unique action elements of the pages (*actions* and *u_actions*, respectively), the differences in the values are even higher.

For the Pluto system, which is the subject of Case Study 3 (Table X), the Tapir framework support increased the value *pages* by 19.5% and the value *u_pages* by 26.6% as measured by the relative difference *DIFF*. The number of the total and unique link and action elements (values *links*, *u_links*, *actions* and *u_actions*) is higher. This case study confirmed the trend observed in Case Study 1.

However, individual times spent by the exploratory testing process differ; thus, the efficiency of the exploratory testing process aided by the framework must be examined in more detail to analyze the proper relationships of the data. Three key indicators are analyzed: (1) the ratio of repetition of the pages and page elements during the testing process, (2) the extent of the SUT explored per time unit, and (3) the defect detection potential.

### A. Repetition of the Pages and Page Elements

The ratio of the repetition of the pages and ratio of the repetition of the page elements during the testing process (RQ1, RQ2) indicate the extent of possible unnecessary action in the SUT during the exploratory testing process. In the collected data, we express this metric as the ratio of the unique pages or the elements exercised during the tests. We start analyzing the data of Case Study 1 (MantisBT, Table VII). For the Tapir framework, the ratio of unique pages explored in the exploratory testing process (value *r_pages*) is improved by 4.4% (23.1% in the relative difference *DIFF*), the ratio of unique link elements (value *r_links*) is improved by 5.8% (14.8% in the relative difference *DIFF*), and the ratio of unique action elements (value *r_actions*) presents the largest improvement of 8.8% (18.8% in the relative difference *DIFF*). These improvements are significant; however, a more detailed explanation is needed to discuss the relevance of these metrics.

TABLE IX

RELATIVE DIFFERENCES BETWEEN RESULTS OF CASE STUDY 2 GROUPS

| Relative difference formula ($\alpha$ stands for a metric from Table VIII) | $\frac{\alpha_{Group1}-\alpha_{Group2}}{\alpha_{Group1}} \cdot 100\%$ | $\frac{\alpha_{Group3}-\alpha_{Group4}}{\alpha_{Group4}} \cdot 100\%$ | $\frac{\alpha_{Group2}-\alpha_{Group4}}{\alpha_{Group2}} \cdot 100\%$ | $\frac{\alpha_{Group1}-\alpha_{Group3}}{\alpha_{Group1}} \cdot 100\%$ |
|---|---|---|---|---|
| Metric / Comment | RANK_NEW vs. RANK_NEW_TEAM for *ElementTypeRank* | RANK_NEW vs. RANK_NEW_TEAM for *PageComplexityRank* | *ElementTypeRank* vs. *PageComplexityRank* for RANK_NEW | *ElementTypeRank* vs. *PageComplexityRank* for RANK_NEW_TEAM |
| *pages* | 5.5% | 11.7% | 2.6% | -4.1% |
| *u_pages* | 17.2% | 17.9% | -8.2% | -8.4% |
| *r_pages* | 12.4% | 7.0% | -11.1% | -4.4% |
| *links* | 20.9% | 17.2% | -13.2% | -7.5% |
| *u_links* | 30.3% | 29.3% | -8.4% | -6.4% |
| *r_links* | 11.9% | 14.6% | 4.2% | 1.2% |
| *actions* | 17.0% | 16.5% | -9.0% | -7.7% |
| *u_actions* | 29.3% | 22.7% | -20.3% | -9.1% |
| *r_actions* | 14.8% | 7.4% | -10.3% | -1.5% |
| *time_page* | -7.4% | -12.9% | -12.2% | -6.3% |
| *time_u_page* | -22.5% | -21.3% | -1.0% | -1.9% |
| *time_link* | -28.3% | -20.3% | 3.5% | -2.8% |
| *time_u_link* | -45.6% | -40.8% | -0.8% | -4.0% |
| *time_action* | -22.3% | -19.3% | -0.2% | -2.6% |
| *time_u_action* | -43.5% | -28.9% | 9.2% | -1.1% |
| *defects* | 14.7% | 14.8% | -3.0% | -3.0% |
| *u_defects* | 19.8% | 18.7% | -7.2% | -5.5% |
| *time_defect* | -19.0% | -16.9% | -6.1% | -7.4% |
| *time_u_defect* | -26.5% | -22.5% | -1.9% | -4.9% |

TABLE X
COMPARISON OF MANUAL EXPLORATORY TESTING APPROACH WITH TAPIR FRAMEWORK: DATA FROM SUT MODEL FOR CASE STUDY 3

| Metric | Manual approach | Tapir framework used | *DIFF* |
|---|---|---|---|
| $\mid T \mid$ | 10 | 10 | - |
| *pages* | 98.6 | 122.5 | 19.5% |
| *u_pages* | 24.3 | 33.1 | 26.6% |
| *r_pages* | 24.6% | 27.0% | 8.8% |
| *links* | 39.3 | 61.5 | 36.1% |
| *u_links* | 16.1 | 26.9 | 40.1% |
| *r_links* | 41.0% | 43.7% | 6.3% |
| *actions* | 13.6 | 25.5 | 46.7% |
| *u_actions* | 8.2 | 17.7 | 53.7% |
| *r_actions* | 60.3% | 69.4% | 13.1% |
| *time_page* | 23.9 | 22.3 | -7.4% |
| *time_u_page* | 97.1 | 82.5 | -17.8% |
| *time_link* | 60.1 | 44.4 | -35.3% |
| *time_u_link* | 146.6 | 101.5 | -44.5% |
| *time_action* | 173.5 | 107.0 | -62.1% |
| *time_u_action* | 287.8 | 154.2 | -86.6% |
| *defects* | 34.8 | 44.1 | 21.1% |
| *u_defects* | 22,2 | 28.7 | 22.6% |
| *time_defect* | 67,8 | 61.9 | -9.6% |
| *time_u_defect* | 106.3 | 95.1 | -11.8% |

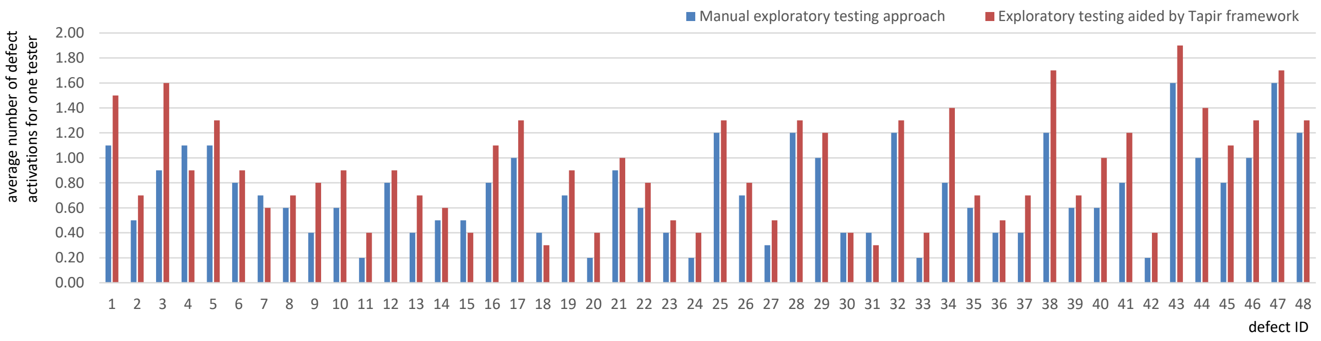


Fig. 10. Potential of manual exploratory testing and Tapir framework approach to detect injected defects in the SUT in Case Study 3

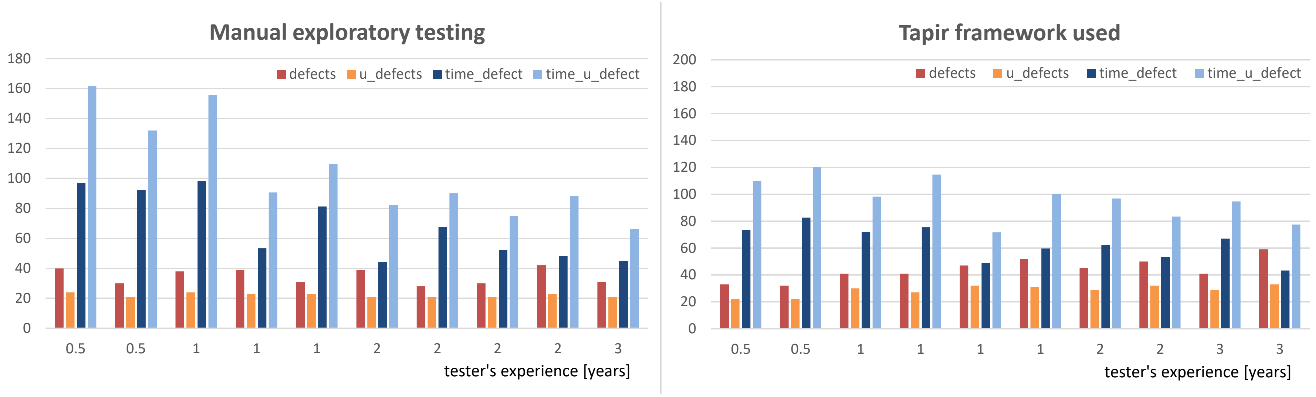


Fig. 11. Defect activations related to experience of the testers (Case Study 3)

The data indicate that the ratio of unique pages is relatively low. For example, when exploring the SUT in the manual exploratory testing process, each page was repeated an average of 6.83 times to achieve a new page in the SUT. In the case of the Tapir framework support, this number decreases to 5.25 because the visited pages in the SUT are repeated during the testing process. An interesting point is that the links are frequently repeated during the testing process. For the manually performed exploratory testing, participants exercised each link 3.02 times to explore one new unique link transition. In the case of the Tapir framework support, this ratio decreased to 2.57. When imagining navigation in the SUT and repetition of its particular functions with various test data, this finding is consistent with the total picture. The same case is the repetition of action elements, in which each action has been repeated 2.56 times in the case of the manual exploratory testing process and 2.08 times in the case of Tapir support.

In Case Study 3 (Pluto, Table X) , we observe a similar trend in which the Tapir framework improved the ratio of unique pages that were explored in the exploratory testing process (value *r_pages*) by 2.4% (8.8% in the relative difference $DIFF$) and the ratio of unique action elements (value *r_links*), the improvement was 2.8% (6.3% in relative difference $DIFF$) in case of Tapir framework support. In case of action elements (value *r_actions*) by 9.1% (13.1% in relative difference $DIFF$).

However, we concluded that the efficiency of the Tapir framework or exploratory testing process cannot be evaluated solely based on the ratio of unique elements because exercising the SUT with additional combinations of test data may decrease these numbers and impact the efficiency of the testing process. The use of additional test data combinations (with the SUT elements repaired additional times during the exploration) can lead to the detection of additional defects. This outcome strongly depends on the testing goals and principal types of defects that we want to detect. If the testing goal is a rapid smoke test of the SUT, the ratio of unique pages or page elements can be a suitable indicator of process efficiency. If the testing goal is to detect additional complex structural defects in the SUT, this metric is not a reliable indicator of the testing efficiency. Thus, other indicators must be analyzed and discussed.

### B. The Extent of the SUT Explored

The extent of the SUT explored per time unit (RQ1, RQ2) indicates total time efficiency when exercising the SUT with exploratory tests. In Case Study 1 (Mantis BT, Table VII), the average time spent on a page (*time_page*) improves by 6.6% in the case of Tapir support. This finding can be explained by the Tapir handling overhead connected to the exploratory testing process, including the documentation of the path, decision, and documentation of the test data. Because the SUT pages were frequently repeated, the significance of this result is not major. The detailed data in Figure 8 indicate differences among the individual times spent on a page by the testers. This factor is strongly influenced by an individual tester's attitude and work efficiency.

For the individual time spent on SUT pages, we need to distinguish two factors that contribute to the total testing time: (1) overhead related to the exploratory testing process, which is decreased by the Tapir framework; and (2) time required to analyze the SUT page and identify and report defects. The second part is equivalent in both the manual exploratory testing and aided exploratory testing. In the first factor, the machine support can reduce the time spent on overhead activities. In the provided data, both parts are mixed (because distinguishing these two parts is nearly impossible when collecting data based on monitoring the events in the SUT user front end).

Because we are interested in exploring the SUT functions available on pages, we analyze the amount of time is needed to explore the SUT action element (value *time_action*) or link (value *time_link*). In the case of action elements, the time significantly changes by 97.3% in the case of Tapir support. In the case of links, the difference is also significant (43.2%). The differences are even more striking in the case of unique pages (value *time_u_page*, difference 38.7% in favor of the Tapir framework), unique action elements (value *time_u_action*, difference 141.9%), and unique link elements (value *time_u_link*, difference 68.1%).

In Case Study 3 (Pluto system, Table X), the Tapir framework improved the time spent on a page (*time_page*) by 7.4%. However, an analysis of the data related to the link and action elements (functions available on SUT pages) provides more relevant results. With the framework support, the value *time_action* increased by 62.1% and the value *time_link* increased by 35.3%. Similar to Case Study 1, this improvement is greater in the case of unique action elements (value *time_u_action*, difference 86.6%) and unique link elements (value *time_u_link*, difference 44.5%).

From the presented figures, we can conclude that the Tapir framework leads to a more efficient exploration of the SUT functions in relation to the time spent testing. However, this optimism can diminish when we discuss the possible various goals of the testing process. For a rapid smoke or exploratory lightweight testing of the SUT, when the primary mission statement is to explore the new SUT parts rapidly and efficiently, Tapir can provide promising support. For more thorough testing, the validity of these metrics shall be revised as more thorough tests and more extensive variants of test data are employed. Thus, the results for this part will be analyzed based on the efficiency of the defect detection potential.

### C. Defect Detection

Defect detection potential (RQ1, RQ2) is an alternative to the defect detection rate. In this metric, we determine whether the artificial defect has been activated in the code (which was ensured by the Tapir framework logging mechanism). When a tester activates a defect, he is capable of subsequently identifying and reporting the defect. In Case Study 1 (Mantis BT, Table VII), we examine the influence of the Tapir framework on the defect detection potential in the case of inserted artificial defects.

In the case of Tapir framework support, testers activated a total of 32.8% more defects when we considered all activated defects, including repeating defects (a tester exercised the same functionality with an inserted defect more frequently) and 31.1% more defects when we only considered unique defects. Of the 19 inserted defects, an average of 4.6 unique defects were detected in the manual execution of exploratory testing, whereas 6.6 unique defects were detected with Tapir support. This amount is approximately one-third of all inserted defects, and this result is attributed to the difficult characteristics of the inserted artificial defects.

The group that uses the Tapir support exercised the SUT longer than the group that does not use the Tapir support; therefore, we are interested in determining the time needed to activate a defect. With Tapir support, the average time to activate a defect (regardless of whether the activated defects repeat) was reduced by 21.7%, whereas for unique defects, the time was reduced by 18.7%.

The statistics by individual inserted defects are presented in Figure 7, and the details on the efficiency of individual participants are provided in Figure 9. We observe differences among individual testers concerning their efficiency. When analyzing the data, we do not observe a direct correlation between time spent on a page and the number of defects that were detected by individual testers in both groups.

One of the defects, *mc_2*, was activated by one tester from the group that only performed the manual testing and by none of the testers from the group using the Tapir framework support. This situation deserves an analysis. The defect *mc_2* can be activated via the issue list form by the following sequence: (1) set a combo box value to determine which operation has to be performed with a selected list of issues to the “set/unset sticky” value, (2) select additional new issues in the list (the defect is only activated for issues in “new” state), and (3) submit the form by the “ok” button. In the combo box that determines whether an operation has to be performed with a selected list of issues, fourteen different operations need to be tested. Thus, the exploration sequence needed to detect the defect that was not directly captured in the SUT model by a link or action element. The sequence was a combination of particular data values entered in the form (input elements) and an action element. Because of the design of the navigational strategies and ranking functions in the current version of the Tapir framework, this defect has “escaped” the exploration path of the testers. Conversely, one of the manual testers has attempted the particular combination needed to activate this defect.

In Case Study 3 (Pluto, Table X), in which real defects occur in the SUT code, improvements were achieved when the Tapir framework was utilized by the testers in the experimental group. With the framework support, a total of 21.1% additional defects were activated by testers. This value shows the possible repetitive activation of the same defect during the exploratory testing process. When only considering the unique defects, the Tapir framework caused the testers to activate 22.6% additional defects. Because the group that uses the Tapir framework has spent a longer testing time, we analyzed the time needed to activate a defect. The Tapir support improved the average time to activate one defect by 9.6% (*time_defect*). When we only consider unique defect activation, this improvement is 11.8% (*time_u_defect*).

In this case study, the defect density was higher; thus, less time is needed to activate one defect (*time_defect*) and one unique defect (*time_u_defect*). Improvements gained by the Tapir framework are slightly lower than the improvements gained in Case Study 1; however, improvements have also been achieved in the case of the Pluto system.

### D. Evaluation of Individual Strategies

In Case Study 2, we compared the efficiency of individual strategies provided by the framework (RQ3). In our analysis of the data, we refer to Table IX. We start with a comparison of the individual testing strategy RANK_NEW with the team strategy RANK_NEW_TEAM (columns $\frac{\alpha_{Group1}-\alpha_{Group2}}{\alpha_{Group1}} \cdot 100\%$ and $\frac{\alpha_{Group3}-\alpha_{Group4}}{\alpha_{Group4}} \cdot 100\%$). The results differ by ranking function (*ElementTypeRank* vs. *PageComplexityRank*); however, general trends are observed in the data. The team navigational strategy RANK_NEW_TEAM increased the ratio of unique explored pages (*r_pages*) by 12.4% for *ElementTypeRank* and by 7.0% for *PageComplexityRank*. In the case of the ratio of unique link elements (*r_links*), the improvement is 11.9% for *ElementTypeRank* and 14.6% for *PageComplexityRank*. A similar trend is observed for unique action elements (*r_actions*), where the improvement is 14.8% for *ElementTypeRank* and 7.4% for *PageComplexityRank*, which is employed as a ranking function.

The statistics related to the time efficiency of the testing process provide more relevant data. The strategy RANK_NEW_TEAM performs well. The average time spent on a unique page (*time_u_page*) decreased by 22.5% for *ElementTypeRank* and 21.3% for *PageComplexityRank*. The average time spent on a unique link element (*time_u_link*) decreased by 45.6% for *ElementTypeRank* and 40.8% for *PageComplexityRank*. The average time spent on a unique action element (*time_u_action*) decreased by 43.5% for *ElementTypeRank* and 28.9% for *PageComplexityRank*. These numbers indicate the favorability of the team strategy.

Regarding the average unique activated defects that were logged, RANK_NEW_TEAM increases the result by 19.8% for *ElementTypeRank* and 18.7% for *PageComplexityRank*. The average time to detect one unique defect decreases by 26.5% in the case of *ElementTypeRank* and by 22.5% in the case of *PageComplexityRank*. These results correspond to the explored extent of the SUT functions, which is higher in the case of the RANK_NEW_TEAM navigational strategy.

From these relative differences, *ElementTypeRank* performs better; and in the case of the team version of the navigational strategy, greater improvements are observed. This conclusion is not accurate; thus, to assess the efficiency of *ElementTypeRank* and *PageComplexityRank*, we need to independently analyze the data.

*PageComplexityRank* leads to the exploration of a slightly larger extent of the SUT (Table IX). The ratio of unique pages explored is 11.1% higher for the RANK_NEW navigational strategy and 4.4% higher for the RANK_NEW_TEAM navigational strategy. The ratio of unique forms explored is 10.3% higher for the RANK_NEW navigational strategy.

Regarding the average times spent on a SUT page, link and action elements, the only significant difference is the average time spent on a page (*time_page*), which is improved by 12.2% for the *ElementTypeRank* in the case of the RANK_NEW navigational strategy and by 6.3% in the case of RANK_NEW_TEAM navigational strategy. *PageComplexityRank* leads to the exploration of more complex pages, which requires additional processing time and more time spent on an SUT page. Because the more complex pages usually aggregate more unexplored links and action elements, the extent of the explored SUT parts is higher than the case of *ElementTypeRank* as previously discussed.

*PageComplexityRank* enables the exploration of more action elements than *ElementTypeRank* (Table VIII, value *u_actions*) because of the combination of the navigational strategy algorithm with the *PageComplexityRank* ranking function. The Tapir framework mechanism scans the following pages and prefers the more complex pages (usually containing more action elements to explore). Note that in page $w$, the ranking function *ElementTypeRank* is calculated for both link elements $l \in L_w$ and action elements $a \in A_w$, whereas the *PageComplexityRank* is only calculated for link elements $l \in L_w$ (refer to Tables I and II).

*PageComplexityRank* also slightly increased the average number of logged unique activated defects by 7.2% for the RANK_NEW strategy and 5.5% for the RANK_NEW_TEAM strategy. The difference in the average time to detect these defects was not significant.

From all analyzed data, the combination of navigational strategy and ranking function RANK_NEW_TEAM with *PageComplexityRank* presents the most efficient number of activated inserted defects and the extent of the explored SUT. However, when considering the time efficiency to explore the new SUT functions, RANK_NEW_TEAM with *ElementTypeRank* seems to be a better candidate.

### *E. Influence of Testers Experience*

The last issue to discuss is the relation between the experience of the testers and the activated defects (RQ4). As the total testing time differed for individual testers, no clear trend can be observed from *defects* and *u_defects* values in Figure 11. However, when we analyze average time to activate a defect (*time_defec*t) and average time to activate unique defect (*time_u_defect*), both of there values decrease with tester's experience in case of manual exploratory testing. This effect has been observed previously also in a study by Micallef *et al.* [7]. With the Tapir framework support, we can observe a similar trend, nevertheless, the decrease of *time_defec*t and *time_u_defect* values is not so significant as in the case of manual testing. This is because, for more junior testers (0.5 up to 1 year of experience), the Tapir framework allowed more efficient exploratory testing and the values *time_defec*t and *time_u_defect* are lower. For more senior testers (2 up to 3 years of experience), the difference in *time_defec*t and *time_u_defect* is not significant.

### *F. Summary*

To conclude all three case studies, Case Studies 1 and 3 provided data to answer RQ1 and RQ2. Compared with the manual approach, the support of the Tapir framework enables the testers to explore larger extents of the SUT and parts of the SUT that were previously unreached. This finding is also documented by the number of SUT pages explored per time unit, which slightly increases in the case of the SUT pages in favor of the Tapir framework (6.6% for MantisBT, 7.4% for Pluto). However, this difference becomes significant when we consider unique pages (38.7% for MantisBT, 17.8% for Pluto), total links (43.2% for MantisBT, 35.3% for Pluto), unique links (68.1% for MantisBT, 44.5% for Pluto), total action elements (97.3% for MantisBT, 62.1% for Pluto), and unique action elements (141.9% for MantisBT, 86.6% for Pluto). However, these figures only documented the ability of the framework to enable more efficient exploration of new parts of the SUT by the testers, and a relationship to the intensity of testing is not expressed here. More thorough testing involves repetition of the SUT parts. Thus, the reliability of this indicator will be discussed in this case. The measured defect detection potential also indicates the superiority of the Tapir framework. With systematic support (and because the testers were able to explore a larger extent of the SUT), the testers activated 31.1% additional unique inserted defects in MantisBT and 22.6% in the Pluto system. Regarding the time efficiency to detect a defect, in the case of the Tapir framework, this indicator improved by 18.7% for the unique inserted defects for MantisBT and 11.8% for the Pluto system. Case Study 3 differed from Case Study 1 in the defects that were activated during the experiment. Instead of the artificial defects used in Case Study 1, the SUT code, which was the subject of Case Study 3, contained a set of real historical defects. Improvements achieved by the Tapir framework in Case Study 3 were slightly less than the improvements achieved by the Tapir framework in Case Study 1. However, improvements can also be observed in Case Study 3. Regarding RQ2, the analysis of the results did not indicate a decrease in the efficiency of the exploratory testing process supported by the Tapir framework.

Regarding the RQ2, no indicator documenting an aspect in which efficiency of the exploratory testing process supported by the Tapir framework would decrease was found during the analysis of the results.

Case Study 2 provided data to answer RQ3. Regarding the comparison between the RANK_NEW and RANK_NEW_TEAM navigational strategies, the team navigational strategy performs better in all measured aspects. For the ranking function, the *PageComplexityRank* ratio of unique explored pages increased by 7.0%, the ratio of unique link elements increased by 14.6%, and the ratio of unique action elements increased by 7.4%. The average time spent on a unique page improved by 21.3%, the average time spent on a unique link improved by 40.8%, and the average time spent on unique action elements improved by 28.9%. The total unique activated defects that were logged improved by 18.7%, and the average time to detect one unique defect improved by 22.5%. Separately analyzed, the ranking function *ElementTypeRank* in terms of the extent of explored SUT functions and generates a slightly higher number of activated unique defects.

The combination of the navigational strategy RANK_NEW_TEAM with the ranking functions *ElementTypeRank* and *PageComplexityRank* does not indicate a preference. RANK_NEW_TEAM with *PageComplexityRank* performed slightly better regarding the extent of explored SUT and detected defects; conversely, RANK_NEW_TEAM with *ElementTypeRank* was slightly more time efficient.

Regarding the RQ4, average times needed to activate a defect decreased with tester's experience in case of manual exploratory testing and slightly in case of Tapir framework support. With the Tapir framework support, these times decreased significantly for junior testers (0.5 up to 1 year of experience). For more senior testers (2 up to 3 years of experience), the improvement in these indicators was not significant.

## VII. Threats to Validity

During the case study experiments, we attempted to equalize the conditions of the compared groups and compare only comparable alternatives while keeping other conditions fixed for all participant groups. Several concerns were identified regarding the validity of the data, which we discuss in this section.

We previously discussed the relevance of the metrics in Section VI, which was intended to be objective. For each of the key metrics, proper disclaimers and possible limiting conditions are described. In all studies, we employed a defect activation concept instead of a defect detection concept. Defect activation expresses the likelihood that the tester will notice the defect when executed. In Case Studies 1 and 2, the Tapir framework logging mechanism exactly logged the fact that a defect was activated. In Case Study 3, we obtained information about the presence of the defects in the SUT code employed in the experiment. The SUT code was accompanied by the logging mechanism, which recorded the flow calls of the SUT methods during the tests. Then, we performed an automated analysis of the defects that were activated by the exercised tests. This analysis ensured the accuracy of the collected data related to defect activation. In practice, we can expect a lower real defect reporting ratio because certain activated defects will not be noticed and reported by the testers. Because this metric was employed in all comparisons, our opinion is that it can be used for measuring a trend that is alternative to the defect detection ratio (which can be biased by individual flaws in defect reporting by experimental team members). The idle time of a tester can influence the measured times during a session (which is a likely scenario when analyzing measured data; refer to the graph in Figure 8). In the experiments, we did not have a better option for measuring the time spent during the testing process. Initially, we experimented with a subjective tester's report of time spent by individual testing tasks; however, this method proved to be less reliable than the automated collection of time stamps related to the tester's actions in the UI, which was employed in the case studies. We attempted to minimize this problem by excluding excessively lengthy steps (tester spent more than 15 minutes on a particular page) caused by leaving the session open and not testing.

We can consider that the influence of these excluded steps is not significant because the ratio of these steps was lower than 1.26% for the manual exploratory testing for all case studies and lower than 0.87% for the exploratory testing supported by the Tapir framework for all case studies.

In Case Study 1, the group that uses the Tapir framework was larger (31 versus 23 in the group that performed exploratory testing without support). The size of the group was sufficiently large to mitigate this risk, and all testers acted individually. Thus, team synergy did not have a role in this case study, and all analyzed data were averages for a particular group.

Previous experience in exploratory testing of the experiment participants and the natural ability of individuals to efficiently perform this type of testing can differ among the participants. In the experimental group, none of the testers were expertly specialized in exploratory testing, which could favor one of the groups. Participants of Case Studies 1 and 2 were randomly distributed to the groups, which should mitigate this issue. In the Case study 3, we distributed the testers to both groups to have the average length of praxis in both groups equal 1.5.

Regarding the size of the SUTs (Mantis BT tracker and Pluto), their workflows and screen-flow model are sufficiently extensive to draw conclusions regarding all defined research questions. The employed version of MantisBT consists of 202964 lines of code, 938 application files and 31 underlying database tables. The utilized version of the Pluto system consists of 56450 lines of code, 427 application files and 41 underlying database tables.

## VIII. Related Work

In this section, we analyze three principal areas relevant to the presented approach: (1) engineering of the SUT model, (2) generation of tests from the SUT model and (3) exploratory testing technique. The first two areas are usually closely connected in the MBT approach. However, in the Tapir framework, we reengineer the SUT model based on the SUT via a continuous process, which represents a difference from the standard MBT approach.

Current web applications have specific elements that differentiate them from any other software application, and these specific elements significantly affect the testing of these applications. The contemporary development styles of UI construction are not only HTML based, and the user experience is dynamically enhanced using JavaScript, which increases the difficulty of identifying the HTML elements. The dynamic nature of the UI hinders the correct identification of the UI elements. Crawlers, which are tools that systematically visit all pages of a web application, are often employed to rapidly collect information about the structure, content, and navigation relationships between pages and actions of the web application [10], [11]. Although crawlers can rapidly inspect an entire application, the sequence of steps by a crawl may differ from the sequence made by a manual tester. Some parts of the application are not reachable by the crawler. In addition, crawlers can address difficulties when user authorization is required in the SUT.

Several solutions for reengineering the SUT model from the actual SUT were discussed in the literature. As an example, we refer to Guitar [12], in which web applications are crawled and a state machine model is created based on the SUT user interface. A mobile application version of this crawler, MobiGuitar, has been developed [13]. As an alternative notation to these state machine models, the Page Flow Graph can be constructed from the actual web-based SUT [14]. The graph captures the relationship among the web pages of the application, and the test cases are generated by traversing this graph (all sequences of web pages). The PFG is converted into a syntax model, which consists of rules, and test cases are generated from this model. The construction of the PFG is not described in this paper. An extra step is required to convert the PFG to a syntax model to generate subsequent tests.

Common elements, including UI patterns, are utilized when developers create the UI of the applications. Examples of this pattern are Login, Find, and Search. Users with some level of experience know how to use the UI or how it should be used. As described by Nabuco *et al.* [15], generic test strategies can be defined to test these patterns. Pattern-based model testing uses the domain specific language PARADIGM to build the UI test models based on UI patterns and PARADIGM-based tools to build the test models and generate and execute the tests. To avoid the test case explosion, the test cases are filtered based on a specific configuration or are randomly filtered. An alternative approach is iMPAcT [16], [17], which analyzes UI patterns in mobile applications. These patterns can be employed in SUT model reengineering.

Numerous approaches can be employed to generate the tests from the SUT model. Extensive usage of UML as the design modeling language implies its use for the MBT techniques. A survey to improve the understanding of UML-based testing techniques was conducted by Shirole *et al.* [18], who who focused on the usage of behavioral specifications diagrams, e.g., sequence, collaboration, and state-chart and activity diagrams [19]. The research approaches were classified by the formal specifications, graph theory, heuristics of the testing process, and direct UML specification processing [18]. Use-case models may represent another source for creating functional test cases as explored by [20]. In addition to UML, other modeling languages, such as SysML [21] or IFML [22], mainstream programming languages, finite machine notations, and mathematical formalisms, such as coq [23], are utilized. The SUT user interface is subject to the model-based generation of test cases, and integration testing is also under investigation [24].

Open2Test test scripts are automatically generated from software design documents [25]. These documents are artifacts of the design process. A design document, design model and test model for the integration testing tool TesMa [26] are extended with information on the structure of the web page (identification of the HTML elements). When the software specification is changed, the latest scripts are regenerated from recent test design documents. This approach reduces the cost of the maintenance of the test scripts. However, the design documents must be extended with the detailed information before the test generation process can start.

The code-based API testing tool Randoop[4] generates random sequences of method calls for a tested class. These sequences are stored as JUnit test cases. Klammer *et al.* [27] have harnessed this tool to generate numerous random interaction test sequences. The tool cannot interact with the UI of a tested application; therefore, a reusable set of adapters that transforms the API calls into UI events was created. The adapters also set up and tear down the runtime environment and provide access to the internal state of the SUT. Fraser *et al.* [28] have evaluated a test case generation tools called EvoSuite on a number of Java projects (open source, industrial and automatically generated projects). The large empirical study presented in the paper shows that EvoSuite can achieve high test coverage but not on all types of classes (for example the file system I/O classes). The interaction with the SUT can be recorded using a capture & replay tool, or the source code is analyzed to create a call graph, analyzes the event sequences and later generate the test cases [29]. The event sequences describe the executable actions that can be performed by the user when employing an application (an Android application, in this case). The actions are converted to JUnit tests classes that are executed in Robotium, which is a test automation framework.

The manual version of the exploratory testing technique has been investigated by several studies, e.g., [3], [4], [7]. The influence of a tester's experience on the efficiency of the testing process has been examined [7]. During this experiment, more experienced testers detect more defects in the SUT. Teamwork organization in exploratory testing as a potential method to improve its efficiency was also explored [5]. In this proposal, team sessions are used to organize the work of the testers. The Tapir framework also employs the idea of teamwork, although none of the team sessions are organized. However, the framework collects information about explored parts of the SUT and test data and uses this information to prevent the duplication of tests within a defined team of testers.

Conceptually, combinations of exploratory testing and MBT can be traced in the related literature; however, the main use case of these concepts differs from the proposed Tapir framework. For instance, recordings of performed tests are used to detect possible inconsistent parts of the SUT design model [30].

In our approach, we use the web application model that was created via adopting, modifying, and extending the model by Deutsch *et al.* [9]. During our study, the model underwent significant changes, which we explained in Section II-B. From the related study, the model conceptually resembles the IFML standard. In our previous study, we explored the possibility of using IFML as an underlying model [22]. Specifics of the presented case enable us to retain the model as defined in this paper.

[4] https://randoop.github.io/randoop

## IX. Conclusion

To conduct the exploratory testing process in a resource-efficient manner, the explored path in the SUT shall be recorded. The entered test data is remembered to fix defects or perform the regression testing process, and work is distributed to individual testers in a controlled and organized manner to prevent duplicate and inefficient tests. This step can be manually performed; however, this process usually requires the intensive and permanent attention of a Test Lead. A considerable amount of time is spent on administrative tasks related to recording the explored parts of the SUT and other details of tests. A strong need to communicate the progress of the team of testers and operationally re-plan the work assignments could render the exploratory testing technique more challenging for distributed work teams.

For the more extensive SUT, this organization of work can become difficult. To visualize the current state of the exploratory testing, a shared dashboard, which is either an electronic dashboard or a physical dashboard located in an office space, where the exploratory testers are located, can be utilized. In the case of a more extensive SUT, capturing the state of testing on a physical dashboard can become problematic, and the maintenance of data in an electronic dashboard may become a tricky task. To remember the previously employed test data, shared online tables can be utilized. However, accessing these tables and maintaining their content can generate particular overhead for the testers during the exploratory testing process. Thus, the efficiency gained by a systematic approach to the test data can be decreased by this overhead.

The proposed Tapir framework aims to automate the administrative overhead caused by the necessity to document the path in the SUT, test data, and related information. The framework serves to distribute the work in the team of exploratory testers by automated navigation based on several navigational and test data strategies. These strategies are designed to explore new SUT functions that were previously revealed by testing, retesting after SUT defect fixes and testing regressions. To prevent duplicate tests and duplicate test data (which is one of the risks of the manual exploratory testing process), team versions of the navigational strategies were designed.

To assess the efficiency of the proposed framework, we conducted three case studies. In these case studies, we used two systems under testing. The first system was an open-source MantisBT issue tracker with 19 inserted artificial defects, and it was accompanied by a logging mechanism to record the activation of the defects during the exploratory testing process. The second SUT was the Pluto system developed via a commercial software industrial project with 48 real software defects. The SUT code was accompanied by a logging mechanism that records the flow calls of the SUT methods during the tests. This mechanism enabled the automation of exact analyses to determine the defects that were activated during the tests.

A comparison of the same process performed by the manual exploratory testing and the exploratory testing supported by the Tapir framework focuses on the individual exploration of the SUT by particular testers, and several significant results were obtained. The exploratory testing supported by the Tapir framework enabled the testers to explore more SUT functions and to explore components of the SUT that were previously unreached. This effect was also confirmed by the measured number of SUT pages explored per time unit, which improved by 6.6% for MantisBT and by 7.4% for the Pluto system in the case of the Tapir framework. In the case of other elements, this metric significantly improved. In the case of MantisBT, the improvement was 38.7% for unique pages, 43.2% for total links, 68.1% for unique links, 97.3% for total action elements and 141.9% for unique action elements. In the case of the Pluto system, the improvement was 17.8% for unique pages, 35.3% for total links, 44.5% for unique links, 62.1% for total action elements and 86.6% for unique action elements.

However, these metrics document the capability of the framework to enable testers to efficiently explore new parts of the SUT, and they are the most relevant in the case of rapid lightweight exploratory testing, in which a tester's goal is to efficiently explore new SUT functions and as many previously unexplored functions as possible. In the case of more thorough testing, the relevance of these metrics will be revised. In principle, SUT pages and elements would repeat the tests when exercised more frequently by more extensive input test data combinations. With the support of the Tapir framework, testers performing exploratory tests of MantisBT were able to reach and activate 31.1% more unique inserted artificial defects, and the time needed to activate one unique defect improved by 18.7%. In the experiment with the Pluto system, 22.6% additional unique real defects were activated. Regarding the time needed to activate one unique defect, the improvement was lower relative to MantisBT at 11.8% but still significant.

A comparison of navigational strategies and ranking functions clearly documented that team-based navigational strategies are more efficient than individual navigational strategies. For the *PageComplexityRank* ranking function employed in the comparison of the strategies, the ratio of unique explored pages increased by 7.0% for the team navigational strategy. In addition, the ratio of unique link elements increased by 14.6%, the ratio of unique action elements increased by 7.4%, the average time spent on a unique page improved by 21.3%, the average time spent on a unique link improved by 40.8%, the average time spent on unique action elements improved by 28.9%, the total unique activated defects that were logged improved by 18.7%, and the average time to detect one unique defect improved by 22.5%.

Regarding the ranking functions, *PageComplexityRank* performed slightly better than *ElementTypeRank* in several aspects. We are currently conducting additional experiments to obtain an optimal ranking function. The concept of the Tapir framework enables this function to be flexibly configured by setting the calibration constants (refer to II). In our opinion, the structure of the SUT pages and the complexity of its workflows have a role in determining the optimal ranking function for a specific case.

Average times needed to activate a defect decreased with tester's experience. This trend was obvious in case of manual exploratory testing and also present, however, less significantly, in case of Tapir framework support. The Tapir framework support decreased times needed to activate a defect significantly for testers having 0.5 up to 1 year of experience. For more senior testers, having 2 up to 3 years of experience, no clear improvement trend was observed.

The results of the manual exploratory testing process can be influenced by the Test Lead. Even for a manual process, a systematic approach can be efficient. Providing systematic guidance to the testers (including proper documentation of the explored SUT parts) is a challenging task because this activity has to be continuously performed during the testing process. Thus, the proposed Tapir framework can provide efficient support in eliminating the administrative overhead and enabling the Test Lead to focus on the analyzing the state, performing strategic decisions during the testing process and motivating the testing team. The effect of this support would be even stronger in the case of extensive SUTs and exploratory software testing in business domains, with which the testers do not have familiarity. The results from the presented case studies documented the viability of the proposed concept and motivate us to evolve the framework.

## Acknowledgments

This research is conducted as part of the project TACR TH02010296 Quality Assurance System for Internet of Things Technology. The research is supported by the internal grant of CTU in Prague, SGS17/097/OHK3/1T/13.

## References

[1] C. Ebert and S. Counsell, "Toward software technology 2050," *IEEE Software*, vol. 34, no. 4, pp. 82–88, 2017.

[2] M. Lindvall, D. Ganesan, S. Bjorgvinsson, K. Jonsson, H. S. Logason, F. Dietrich, and R. E. Wiegand, "Agile metamorphic model-based testing," in *Proceedings of the 1st International Workshop on Metamorphic Testing*, ser. MET '16. New York, USA: ACM, 2016, pp. 26–32.

[3] D. Pfahl, H. Yin, M. V. Mäntylä, and J. Münch, "How is exploratory testing used? a state-of-the-practice survey," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2014, p. 5.

[4] C. Ş. Gebizli and H. Sözer, "Impact of education and experience level on the effectiveness of exploratory testing: An industrial case study," in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, March 2017, pp. 23–28.

[5] P. Raappana, S. Saukkoriipi, I. Tervonen, and M. V. Mäntylä, "The effect of team exploratory testing – experience report from f-secure," in *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, April 2016, pp. 295–304.

[6] S. M. A. Shah, C. Gencel, U. S. Alvi, and K. Petersen, "Towards a hybrid testing process unifying exploratory testing and scripted testing," *Journal of Software: Evolution and Process*, vol. 26, no. 2, pp. 220–250, 2014.

[7] M. Micallef, C. Porter, and A. Borg, "Do exploratory testers need formal training? an investigation using hci techniques," in *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, April 2016, pp. 305–314.

[8] B. S. Ahmed, K. Z. Zamli, and C. P. Lim, "Application of particle swarm optimization to uniform and variable strength covering array construction," *Applied Soft Computing*, vol. 12, no. 4, pp. 1330–1347, 2012.

[9] A. Deutsch, L. Sui, and V. Vianu, "Specification and verification of data-driven web applications," *Journal of Computer and System Sciences*, vol. 73, no. 3, pp. 442–474, 2007.

[10] H. Tanida, M. R. Prasad, S. P. Rajan, and M. Fujita, *Automated System Testing of Dynamic Web Applications*. Berlin, Heidelberg: Springer, 2013, pp. 181–196.

[11] V. Dallmeier, M. Burger, T. Orth, and A. Zeller, *WebMate: Generating Test Cases for Web 2.0*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 55–69.

[12] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "Guitar: an innovative tool for automated testing of gui-driven software," *Automated Software Engineering*, vol. 21, no. 1, pp. 65–105, Mar 2014.

[13] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "Mobiguitar: Automated model-based testing of mobile apps," *Software, IEEE*, vol. 32, no. 5, pp. 53–59, 2015.

[14] J. Polpong and S. Kansomkeat, "Syntax-based test case generation for web application," in *2015 International Conference on Computer, Communications, and Control Technology (I4CT)*, April 2015, pp. 389–393.

[15] M. Nabuco and A. C. Paiva, "Model-based test case generation for web applications," in *Proceedings of the 14th International Conference on Computational Science and Its Applications ICCSA 2014 - Volume 8584*. New York, NY, USA: Springer-Verlag, 2014, pp. 248–262.

[16] I. C. Morgado and A. C. R. Paiva, "The impact tool: Testing ui patterns on mobile applications," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2015, pp. 876–881.

[17] I. C. Morgado and A. C. R. paiva, "Testing approach for mobile applications through reverse engineering of ui patterns," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, Nov 2015, pp. 42–49.

[18] M. Shirole and R. Kumar, "Uml behavioral model based test case generation: A survey," *SIGSOFT Softw. Eng. Notes*, vol. 38, no. 4, pp. 1–13, Jul. 2013.

[19] A. K. Jena, S. K. Swain, and D. P. Mohapatra, "A novel approach for test case generation from uml activity diagram," in *2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*, Feb 2014, pp. 621–629.

[20] R. Lipka, T. Potuzak, P. Brada, P. Hnetynka, and J. Vinarek, "A method for semi-automated generation of test scenarios based on use cases," in *Software Engineering and Advanced Applications (SEAA), 2015 41st Euromicro Conference on*. IEEE, 2015, pp. 241–244.

[21] C. H. Chang, C. W. Lu, W. P. Yang, C. Chu, C. T. Yang, C. T. Tsai, and P. A. Hsiung, "A sysml based requirement modeling automatic transformation approach," in *2014 IEEE 38th International Computer Software and Applications Conference Workshops*, July 2014, pp. 474–479.

[22] K. Frajtak, M. Bures, and I. Jelinek, "Transformation of ifml schemas to automated tests," in *Proceedings of the 2015 Conference on Research in Adaptive and Convergent Systems*, ser. RACS. New York, USA: ACM, 2015, pp. 509–511.

[23] Z. Paraskevopoulou, C. Hritcu, M. Denes, L. Lampropoulos, and B. C. Pierce, *Foundational Property-Based Testing*. Cham: Springer International Publishing, 2015, pp. 325–343.

[24] T. Potuzak and R. Lipka, "Interface-based semi-automated generation of scenarios for simulation testing of software components," in *SIMUL*. IARIA, 2014, pp. 35–42.

[25] H. Tanno and X. Zhang, "Test script generation based on design documents for web application testing," in *2015 IEEE 39th Annual Computer Software and Applications Conference*, vol. 3, July 2015, pp. 672–673.

[26] H. Tanno, X. Zhang, T. Hoshino, and K. Sen, "Tesma and catg: Automated test generation tools for models of enterprise applications," in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 717–720.

[27] C. Klammer, R. Ramler, and H. Stummer, "Harnessing automated test case generators for gui testing in industry," in *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Aug 2016, pp. 227–234.

[28] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using evosuite," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 2, pp. 8:1–8:42, Dec. 2014.

[29] J. L. San Miguel and S. Takada, "Gui and usage model-based test case generation for android applications with change analysis," in *Proceedings of the 1st International Workshop on Mobile Development*, ser. Mobile! 2016. New York, USA: ACM, 2016, pp. 43–44.

[30] C. Ş. Gebizli and H. Sözer, "Improving models for model-based testing based on exploratory testing," in *2014 IEEE 38th International Computer Software and Applications Conference Workshops*, July 2014, pp. 656–661.