Automated Test Case Implantation to Test Untested Configurations: A Cost-Effective Search-Based Approach

Dipesh Pradhan, Shuai Wang, Tao Yue, Shaukat Ali, Marius Liaaen

Abstract

Context: Modern large-scale software systems are highly configurable, and thus require a large number of test cases to be implemented and revised for testing a variety of system configurations. This makes testing highly configurable systems very expensive and time-consuming.

Objective: Driven by our industrial collaboration with a video conferencing company, we aim to automatically analyze and implant existing test cases (i.e., an original test suite) to test the untested configurations.

Method: We propose a search-based test case implantation approach (named as SBI) consisting of two key components: 1) *Test case analyzer* that statically analyzes each test case in the original test suite to obtain the program dependence graph for test case statements and 2) *Test case implanter* that uses multi-objective search to select suitable test cases for implantation using three operators, i.e., selection, crossover, and mutation (at the test suite level) and implants the selected test cases using a mutation operator at the test case level including three operations (i.e., addition, modification, and deletion).

Results: We empirically evaluated SBI with an industrial case study and an open source case study by comparing the implanted test suites produced by SBI with the original test suite using evaluation metrics such as statement coverage (*SC*), branch coverage (*BC*), mutation score (*MS*). Results show that for both the case studies, the implanted test suites performed significantly better than the original test suites with on average 21.9% higher coverage of configuration variable values. For the open source case study, SBI managed to improve *SC*, *BC*, and *MS* with 4.8%, 7.5%, and 2.6%, respectively.

Conclusion: SBI can be applied to automatically implant an existing test suite with the aim of testing untested configurations and thus achieving higher configuration coverage.

Keywords— search; multi-objective optimization; genetic algorithms, test case implantation.

1. INTRODUCTION

Testing plays a key role to ensure that software systems can be released to market with high quality and more than 50% of time and budget are spent for testing [1]. It is even significantly worse when testing large-scale software systems (e.g., cyber-physical systems) that are usually highly configurable since test engineers need to spend a great deal of effort to implement and revise test cases for testing various configurations, which decreases the efficiency of testing [2].

We have been working with a video conferencing company since 2009 with the aim to assist their current practice of testing large-scale Video Conferencing Systems (VCSs). For each VCS, there are more than 100 configuration variables (e.g., *protocol*), and each variable can be configured with a

- *Dipesh Pradhan is a Ph.D. student with Simula Research Laboratory, Oslo, Norway. E-mail: dipesh@simula.no.*
- *Shuai Wang is with Simula Research Laboratory, Oslo, Norway. E-mail: shuai@simula.no.*
- *Tao Yue is with Simula Research Laboratory and University of Oslo, Oslo, Norway. E-mail: tao@simula.no.*
- *Shaukat Ali is with Simula Research Laboratory, Oslo, Norway. E-mail: shaukat@simula.no.*
- *Marius Liaaen is with Cisco Systems, Oslo, Norway. E-mail: marliaae@cisco.com.*

number of values (e.g., *protocol* can be *SIP* and *H323*). Such highly configurable VCSs bring great challenges for test engineers to manually and systematically design and develop test cases. For example, the *calltype* indicating a particular call type can be configured as *video* or *audio* and the *callrate* that specifies the call rate to be used when placing or receiving video calls can be configured with an integer from *64* to *6000*. For the VCSs that support these two configuration variables, there are in total 2*5,937 =11,874 configurations that are needed to be thoroughly tested. For each configuration, a set of test API commands with a number of parameters has to be called (e.g., dial (*calltype=video*, *callrate= 64*)) and a set of corresponding system status variables need to be checked (e.g., assert (*activecalls*=1,*videocalls*=1)). Manually implementing such test cases (e.g., specifying configurations, calling relevant test API commands, checking corresponding system status) to test the configurations require a large amount of manual work that is practically infeasible. Test engineers at the company usually choose to develop a certain number of test cases by including a limited number of configurations (e.g., *video* with *callrate 6000*) based on their experience of testing VCSs. Such practice may result in high chances that potential errors cannot be detected since some configurations might not be tested.

With the above-mentioned challenges in mind, we argue that it is worth investigating how to automatically and systematically analyze and implant existing test cases to increase the overall configuration coverage and thereby improve the efficiency of testing. Therefore, we propose a search-based test case implantation approach (coined as SBI) to automatically analyze and implant an existing test suite with the aim to test the untested configurations. More specifically, SBI includes two key components: 1) *Test case analyzer* that statically analyzes each test case in the original test suite to obtain the program dependence graph for the statements; and 2) *Test case implanter* that uses multi-objective search to select suitable test cases for implantation using three *operators*, i.e., *select*ion, *crossover*, and *mutation* (at the test suite level) and implants the selected test cases using a *mutation operator* at the test case level that includes three operations: *addition*, *modification,* and *selection*. To assess the quality of the implanted test suites, we define five cost-effectiveness measures: number of configuration variable values covered ($NCVV$), pairwise coverage of parameter values of test API commands ($PCPV$), number of implanted test cases (NIT), number of changed statements (NCS), and estimated execution time (EET). Moreover, the *test case implanter* is implemented on top of a widely applied multi-objective search algorithm, Non-dominated Sorting Genetic Algorithm II (NSGA-II) [3].

To evaluate SBI, we employed one industrial case study from a video conferencing company with a test suite including 118 test cases and one open-source case study (i.e., SafeHome [4]) with 94 test cases. We also applied three evaluation metrics: statement coverage (SC), branch coverage (BC), and mutation score (MS) to evaluate SBI for the SafeHome case study by generating in total 1594 non-equivalent mutants. Note that we cannot apply these metrics (i.e., *SC*, *BC*, and *MS*) to the industrial case study since we do not have access to the source code. The evaluation results showed that the implanted test suites produced by SBI significantly outperformed the original suite for both the case

studies by achieving on average 21.9% higher $NCVV$ and 59.4% higher $PCPV$. Moreover, for the SafeHome case study, the implanted test suites managed to improve *SC*, *BC* and *MS* with on average 4.8%, 7.5%, and 2.6%, respectively.

The key contributions of this paper include:

1) A formalization of the test case implantation problem (Section 3.2);
2) A mathematical definition of five cost-effectiveness measures to assess the quality of implanted test suites (Section 3.3);
3) SBI: A novel search-based test case implantation approach with two key components, i.e., *test case analyzer* and *test case implanter* (Section 4);
4) An empirical evaluation of SBI using two case studies (Section 6).

The rest of the paper is organized as follows: Section 2 introduces a running example for illustrating SBI and the overall context, followed by the formalization of the problem (Section 3). Section 4 presents SBI in detail, followed by the experiment design (Section 5) and results of the empirical study (Section 6). Section 7 presents the threats to validity. Section 8 discusses the related work, and Section 9 concludes the paper.

2. RUNNING EXAMPLE AND CONTEXT

The running example is an excerpt of a sanitized test case from a video conferencing company, which will be used to illustrate SBI throughout the paper. A typical test case at the video conferencing company consists of one *setup class*, one or more *test methods*, one *teardown*, and one *teardown class* (Table 1) as recommended in the unit testing framework in python, PyUnit [5]. A *setup class* is for initializing and setting up the system under test (SUT) (e.g., registering SUT to a registrar at line 1 in Table 1) to be ready for executing the *test methods* in the test case. The *test methods* are for testing SUT functionalities (e.g., the *dial* functionality for making a call from one system to another, as shown at line 4 in Table 1). *Teardown* resets the SUT (e.g., *disconnect* the SUT, as shown at line 7 in Table 1), and it is executed after each *test method* has been called. Lastly, *teardown class* is called after all the *test methods* have been executed to reset the statuses of the SUT that might have been modified at the *setup class* (e.g., disconnecting the SUT from the registrar).

Table 1. An Excerpt of a Sanitized Test Case

Part	Line	Example	Comment
Setup class	1	register SUT to a registrar	Register SUT
Test method	2	*packetlossresilence* = off	Configure SUT
	3	*callrate_var* = 6000	Assign variable
	4	dial(*protocol*=sip, *calltype*= video, *callrate*=*callrate_var*, *autoanswer*=true)	Execute test API command on SUT
	5	wait(4)	Wait 4 seconds
	6	assert(NumberOfActiveCalls=1, NumberofVideoCalls=1)	Verify statuses of SUT
Teardown	7	disconnect call	Reset statuses
Teardown class	8	disconnect SUT from the registrar	Execution completed

Moreover, Fig. 1 presents an overview of a typical testing process for testing VCSs, i.e., SUTs. As a first step, a test case makes the SUT ready for testing, e.g., registering the SUT to a registrar (line 1 in Table 1) as a part of *setup class*. Secondly, the SUT is configured if necessary. For example, the configuration variable *packetlossresilence* at the SUT is configured with *off* in the *test method* (line 2 in Table 1). In the third step, one or more test API command is executed on the SUT. For example, *dial* is executed, which consists of four parameters: *protocol*, *calltype, callrate,* and *autoanswer* assigned with values *sip*, *video, 6000,* and *true,* respectively in Table 1. Then, the statuses of the SUT are verified with an *assertion*. For example, the *assertion* checks if the values of *NumberOfActiveCalls* and *NumberOfVideoCalls* are both 1 in Table 1. Note that typically each *test method* consists of at least one test API command (e.g., *dial*) and an *assertion*. At last, the statuses of the SUT are reset to the original statuses, e.g., *disconnect* as a part of *teardown*. If the test case has more than one *test method*, the next *test method* is executed followed by the *teardown*, and this process is repeated until all the *test methods* in the test case are executed. At the final step, *teardown class* is called to reset the statuses of the SUT that might have been initialized at the *setup class*.

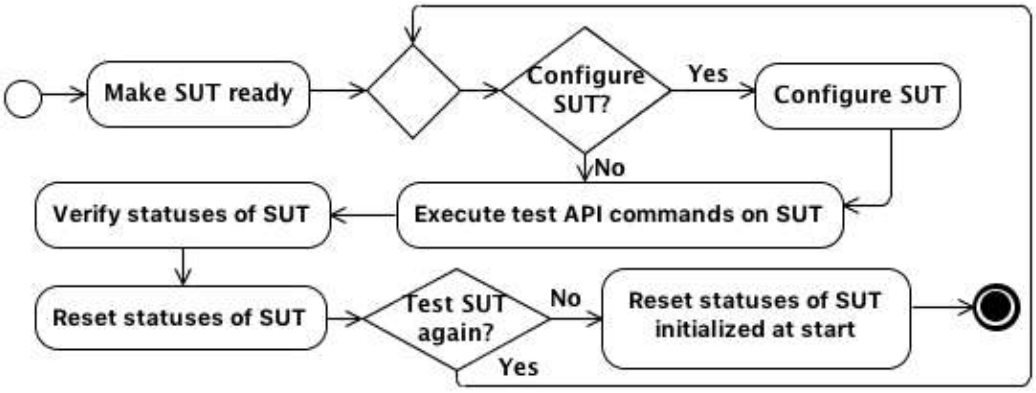


Fig. 1. Overview of testing a VCS (SUT)

Each VCS developed by the video conferencing company is highly configurable. For example, a VCS includes more than 100 configuration variables (e.g., *packetlossresilence*) and each configuration variable can take a set of values (e.g., *packetlossresilence* can take two values: *off* and *on*). Moreover, each test API command requires configuring one or more parameters (e.g., four parameters need to be configured for the test API command *dial*), and each parameter in the test API command can take a number of different values. For example, the test API command *dial* allows values *audio* and *video* for *calltype*, values between *64* and *6000* inclusive for the *callrate*, and values *true* and *false* for *autoanswer*.

Testing all the values for the configuration variables requires a large number of *test methods* if each *test method* covers one configuration variable. Moreover, testing all the combinations of parameter values for the test API commands also requires a large number of *test methods*. Additionally, if one *test method* includes more than one test API command, the combinations could exponentially increase, which makes the manual test case development expensive and even infeasible at certain contexts. Developing new test cases is practically expensive since each test case should include the *setup class, teardown,* and *teardown class*, which cause an extra overhead in terms of test case execution. This is due

to the fact that the *setup class* (for setting up the SUT) and *teardown class* (for resetting the SUT) need to be executed for all the newly implemented test cases, which is quite time consuming. However, if the new *test methods* can be directly added to the existing test cases, the overhead for executing the *setup class* and *teardown class* can be reduced and thereby improving the efficiency of testing. Moreover, using test reduction strategies (e.g., boundary value analysis [6-8]) can further reduce the number of combinations of variables/parameters without significantly decreasing the effectiveness of the test suite.

Additionally, some existing test cases might test the same combinations of values for the configuration variables and test API command parameters. For example, when different test engineers develop test cases that require *dial*, it is possible that the same values for the parameters *protocol*, *calltype*, *callrate*, and *autoanswer* are taken, which can decrease the efficiency of testing since a different combination of configuration variable or test API command parameters could have been used. Notably, more diverse test cases (e.g., in terms of combinations of parameter values in our context) can lead to higher efficiency of testing [9, 10].

Thus, the key objective of this work is to propose a cost-effective search-based approach to automatically implant an existing test suite with the aim to 1) achieve a higher coverage of configuration variable values, 2) cover more combinations of parameter values of test API commands, and 3) increase the efficiency of testing by modifying or removing redundant *test methods* that cover same configuration variable values or the same combinations of parameter values of test API commands.

3. Problem Representation and Measures

This section first defines basic notations (Section 3.1) and the test case implantation problem (Section 3.2), followed by presenting the five cost/effectiveness measures (Section 3.3).

3.1 Basic Notations

We assume that a given original test suite consists of n test cases $T = \{t_1, t_2, \dots, t_n\}$. Each test case is composed of four parts (as mentioned in in Table 1): *setup class*, o number of *test methods*, *teardown*, and *teardown class*, i.e., $t = sc \cup \{tm_1, \dots, tm_o\} \cup td \cup tdc$, where sc, tm_i, td and tdc represent *setup class*, *test method i*, *teardown*, and *teardown class*, respectively.

Table 2. Different Types of Statements in a Test Method

Name	Description	Example
Assignment	Assign values to Numeric, Boolean, String variables	callrate_var = 6000
Conditional	If-then statement represented $p \rightarrow q$, where p is a hypothesis and q is a conclusion	if (wait > 4) accept
Configuration	Configure the SUT	*packetlossresilence* = off
Execution	Perform actions by executing test API commands on the SUT	dial (*protocol*=SIP, *calltype*=video, *callrate*=6000, *autoanswer*=true)
Assertion	Check the statuses of the SUT	assert(NumberofActiveCalls=1)
Wait	Hold the execution of the next statement(s) for a specific time	wait (4)

Each *test method* tm_i is composed of a sequence of i, q statements, $tm_i = \{st_{i,1}, \dots, st_{i,q}\}$ (e.g., the *test method* in Table 1 has 5 statements). Thus, the total statements in a test case is: $ST = \uplus_{tm \in t} F(tm) \uplus F(sc) \uplus F(td) \uplus F(tdc)$, where $F(tm), F(sc)$, $F(td)$ and $F(tdc)$ are functions that return all the statements in the *test method* tm, *setup class* sc, *teardown* td, and *teardown class* tdc, respectively. Moreover, ST is a multiset, which is a collection of objects (e.g., statements in this context) that allows the objects to occur more than once in a set [11]. To enable the implantation, we need to get the statements structured, and therefore we classify them into six categories as shown in Table 2.

Each test case covers one or more configuration variables, and each configuration variable is configured with a configuration value. For example, in the running example (Table 1), the test case covers configuration variable *packetlossresilence*, which is configured by using the value *off*. We represent a set of r configuration variables for test suite T as $CV_T = \{cv_1, \dots, cv_r\}$. Thus, the configuration variable values covered by the test suite T are defined as:

$$CVV_T = \bigcup_{cv \in CV_T} F(cv) \qquad (1)$$

Each test case executes one or more test API commands, each of which has one or more parameters. Each parameter is configured with a specific value at a *test method* for a test case. For example, in Table 1, the test case covers the test API command *dial*, which has four parameters: *protocol*, *calltype, callrate,* and *autoanswer* with the values of *SIP*, *Video, 6000*, and *true,* respectively. We represent the u number of test API commands covered by the test suite T as $AC_T = \{ac_1, ac_2, \dots, ac_u\}$. Each test API command ac_i has i, v parameters (i.e., $ac_i = \{ap_{i,0}, \dots, ap_{i,v}\}$). Systematically considering interactions of parameters during testing can lead to a high chance of finding software faults [9, 10]. Moreover, exhaustive testing (i.e., testing all combinations of parameters) is very expensive in practice. Therefore, pairwise testing has been proposed to reduce the number of interactions of test API parameters meanwhile maintain relatively high fault detection rates, from 50% to 97% as reported in [12]. Thus, we employ pairwise testing [13] in SBI. A set of pairwise tests PT_i is required to cover all interactions of each pair of parameters for each test API command ac_i, such that each test case in PT_i contains i, v values, one for each parameter in ac_i. In other words, for each pair of parameter values $apv_{j,a}$ and $apv_{k,b}$, where $apv_{j,a} \in ap_j$ and $apv_{k,b} \in ap_k$, there exists at least one test in PT_i that contains both $apv_{j,a}$ and $apv_{k,b}$ [9]. The set of pairwise tests required to cover all the pairwise interactions of each parameter pair for all the test API commands in the test suite is:

$$PT_T = \bigcup_{i=1}^{u} PT_i \qquad (2)$$

where u is the number of test API commands covered by T, and PT_i is the set of pairwise tests required for test API command i.

Based on the above-mentioned notations, we define test case implantation as: automatically modifying and/or deleting existing statements from the original test suite and/or adding new statements to the original test suite, with the aim to construct a new test suite, which meets a set of predefined criteria, e.g., maximizing the pairwise coverage of parameter values of test API commands.

Notably, the defined test case implantation does not increase the total number of test cases as compared with the original test suite. We use the following function to represent implanting t_a into $t_{a'}$: $Implant(t_a) \rightarrow t_{a'}$ (3)

3.2 Problem Representation

Let $S = \{s_1, s_2, \dots, s_{ns}\}$ represents a set of potential solutions, where $ns = 2^p - 1$ and $p = \sum_{t \in T} n(ST_t)$. Each solution $s = \{t_1, t_2, \dots, t_n\}$ has the same number of test cases as the original test suite T, and some of the test cases from T are chosen for implantation. $Cost = \{cost_1, \dots, cost_{ncost}\}$ refers to a set of cost measures (e.g., execution time of the test suite) and $Effect = \{effect_1, \dots, effect_{neffect}\}$ denotes a set of effectiveness measures (e.g., coverage of configuration variable values) for evaluating the quality of a solution.

Problem: Search a solution s_f from the total number of ns solutions in S that can achieve the maximum effectiveness with minimum cost.

$$\forall_{i=1\ to\ neffect} \forall_{j=1\ to\ ns}\ Effect\ (s_f, effect_i) \geq Effect\ (s_j, effect_i)$$
$$\cap\ \forall_{k=1\ to\ ncost} \forall_{j=1\ to\ ns}\ Cost\ (s_f, cost_k) \leq Cost\ (s_j, cost_k)$$

$Effect\ (s_j, effect_i)$ returns the i^{th} effectiveness measure of s_j, and $Cost\ (s_j, cost_k)$ returns the k^{th} cost measure of s_j.

3.3 Cost and Effectiveness Measures

This section formally defines three cost measures (Section 3.3.1) and two effectiveness measures (Section 3.3.2) based on the problem defined in Section 3.2.

3.3.1 *Cost Measures*

Number of implanted test cases (*NIT*): Since not all the test cases in the original test suite are selected for implantation, we define NIT to measure the total number of test cases chosen by the search for implantation, which can be calculated as the total number of test cases that exist in s but not in the original test suite T. The number of implanted test cases can be calculated as:

$$NIT = n(\bigcup_{t \in s} t : t \notin T) \quad (4)$$

Our aim is to minimize NIT so that changes are introduced to a minimum number of the existing test cases for simplifying maintenance [14, 15].

Number of changed statements (*NCS*): A statement is called a changed statement if an existing statement is modified or removed or a new statement is added to the test case. The number of changed statements in a solution s is the sum of the numbers of modified statements (MST), deleted statements (DST), and added statements (AST) for each test case in s, such that:

$$NCS = \forall_{t \in s}(n(MST_t) + n(DST_t) + n(AST_t) : t \notin T) \quad (5)$$

If $t_{a'}$ is the implanted test case for the original test case t_a:

$$MST_{ta'} = \forall_{tm \in ta'} \uplus_{st \in tm} st \notin t_a, DST_{ta'} = \forall_{tm \in ta} \uplus_{st \in tm} st \notin t_{a'} \text{ and}$$
$$AST_{ta'} = n(ST_{ta'}) - n(ST_{ta}) + DST_{ta'}.$$

We aim to minimize NCS to ensure that a statement is only changed when necessary (e.g., to cover more configuration variable values).

Estimated execution time (EET): The execution time of a solution (i.e., an implanted test suite) refers to the time required for executing all its test cases. The solutions update dynamically during search and many solutions are produced, which makes it difficult to execute the solutions for getting their execution time. For example, 25000 implanted test suites produced by search have to be executed 25000 times when the number of fitness evaluation is set as 25000, which is computationally expensive and infeasible. Thus, we propose to statically estimate the execution time of a test case in the solution based on the execution time of each statement of the test case. We measure the average execution time for each statement in a test case t_j ($ETES_j$) using the historical execution time of the test case: $ETES_j = \frac{et_j}{n(ST_j)}$, where et_j is the historical execution time of the statement and $n(ST_j)$ is the number of statements included in t_j. The estimated execution time of the test cases in the solution s is:

$$EET = \sum_{t \in s} n(ST_t) \times ETES_t \qquad (6)$$

where $n(ST_t)$ is the total number of statements in a test case. Our aim is to minimize the estimated execution time of the test cases included in a solution.

To ensure that EET is accurate for estimation, we conducted a pilot experiment by producing 20 implanted test suites using our approach, executing them and comparing the real execution time with the estimated execution time (i.e., EET). We noticed that the difference between the real execution time and EET was on average 395 seconds, which has no practical differences. Therefore, we used EET to estimate the real execution time in our context.

3.3.2 *Effectiveness Measures*

Number of configuration variable values covered ($NCVV$): Based on equation 1, the set of configuration variable values covered by a solution s is: $CCVV = \bigcup_{cv \in CV} F(cv)$, where $F(cv)$ is a function that returns the set of configuration variable values for cv. The number of configuration variable values covered by s can be calculated as: $NCVV = n(CCVV)$ (7)

For example, for the sanitized test case in Table 1, $NCVV$ is one as it covers one configuration variable with one value (i.e., *packetlossresilence* with the value *off*). The goal is to achieve the maximum coverage of configuration variable values.

Pairwise coverage of parameter values of test API commands ($PCPV$): $PCPV$ is defined to measure how much pairwise coverage of parameter values of test API commands can be covered by a solution s, and it is calculated as below:

$$PCPV = \forall_{ac \in s} \forall_{i=1}^{n(F(ac))} \forall_{j=i+1}^{n(F(ac))} \left(\sum n(F(ap_i)) \times n\left(F(ap_j)\right) \right) \qquad (8)$$

such that $n(F(ac)) > 1$, where $F(ac)$ is a function that returns the set of API parameters for the test API command ac, while $F(ap_i)$ and $F(ap_j)$ are functions that return the set of values in the test API parameters i and j, respectively. For example, for a solution with only one test case (i.e., sanitized test case in Table 1), $PCPV$ is six since the test API command dial covers six pairs of parameter values. The goal is to maximize the pairwise coverage of parameter values of test API commands.

4. SBI: Search-Based Test Case Implantation Approach

This section first provides an overview of SBI (Section 4.1) followed by the detailed presentation of *test case analyzer* (Section 4.2), and *test case implanter* (Section 4.3).

4.1 Overview of SBI

Fig. 2 presents an overview of SBI consisting of two key components: *test case analyzer* and *test case implanter*. The *test case analyzer* component ensures that implanted test cases are semantically correct (e.g., two new statements should be added in a correct order). For this, the *test case analyzer* component statically analyzes each test case in the original test suite to obtain the program dependence graph [16, 17] for each *test method*, which is required to know dependences among statements. For example, on removing one statement another dependent statement(s) also need to be removed in the *test method* (see Section 4.2). Nodes in the program dependence graph represent the statements in the *test method* and edges represent the control and data dependence edges [18]. Moreover, the *test case analyzer* component automatically classifies all statements into the six categories defined in Table 2 using the statement information document (created one time), which is described in detail in Section 4.2.

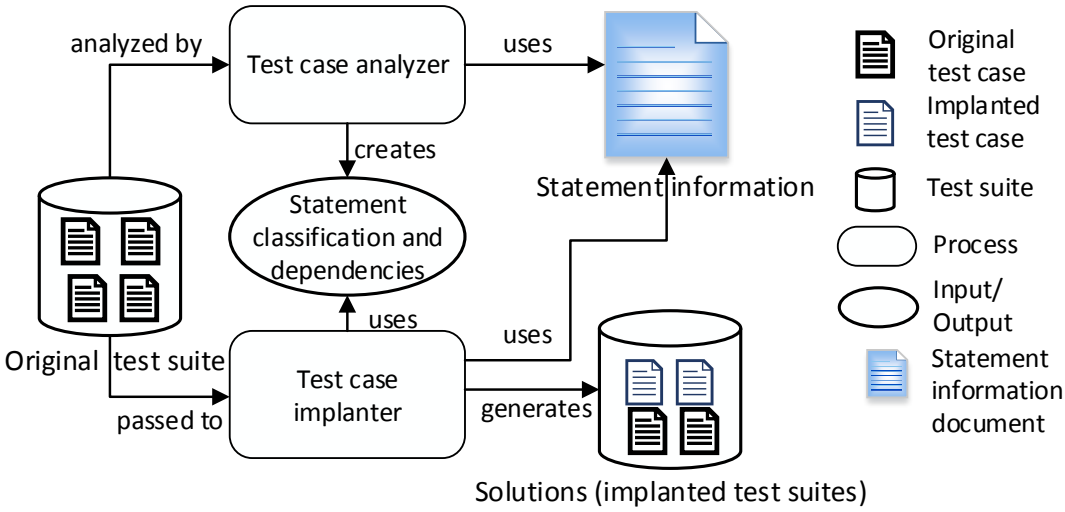


Fig. 2. Overview of SBI

As depicted in Fig. 2, the original test suite is passed to the *test case implanter* component to generate solutions by changing the values of the configuration variables and test API parameters from the list of available values provided in the statement information document (detailed in Section 4.3). Each generated solution includes implanted test cases and remaining (unchanged) test cases in the original test suite that are not chosen for implantation. For implanting a test case, changes are made to

one or more of its classified statement (recall Table 2) using the *test case implanter* component and program dependence graph produced by *the test case analyzer* component is used to change the affected statements.

4.2 Test Case Analyzer

For each test case in the original test suite, the *test case analyzer* component automatically classifies all the statements of the test case into the six categories (Table 2) and constructs a program dependence graph for each *test method*.

Statement classification. The *test case analyzer* component uses the statement information document (Fig. 2) for classifying the statements in the test case. Generally, the statement information document includes: 1) *keywords* to distinguish between the different statements (e.g., for the sanitized test case in Section 2, *packetlossresilence, dial*, *wait,* and *assert* are defined as *keywords* to differentiate *configuration*, *execution*, *wait*, and *assertion statements*, respectively), 2) allowed values for the variables/parameters (that the test engineers need to test), and 3) domain specific rules for identifying the dependency between statements (e.g., later statement(s) in a test case may use same values for the same variable as the preceding statement). Notice that test engineers can build such statement information document based on their specific testing practice in any format (e.g., XML in our case).

In our context, the list of configuration variable names and test API commands are specified in the statement information document to differentiate between *configuration* and *execution statements*. Moreover, *assertion* and *wait statements* are classified based on whether they contain the keyword "assert" or "wait", respectively. The *assignment* statement is classified based on if they are used as a value at the 1) configuration variable or 2) test API parameter/s (e.g., *callrate_var* at line 3 in Table 1 is used as a value in the parameter *callrate* in line 4). The program dependence graph is created using data and control dependences between statements, as explained below.

Data dependence. There exists data dependence between two statements if the second statement refers to the data of the first statement [17]. We define two sets of data dependences: 1) general and 2) domain specific. General dependences apply to all contexts, whereas domain specific data dependences are specific to a particular domain. There exists general data dependence between two statements in a *test method* if a variable in one statement has an incorrect value when the two statements are reversed. For example, as shown in Table 1, the statement in line 4 is data dependent on the statement in line 3 as parameter *callrate* in *dial* is defined in line 4 (i.e., *callrate_var*). We use domain specific rules defined explicitly in the statement information document (as illustrated in Fig. 2) to create domain specific data dependence. For example, in the context of the video conferencing company if there exist two or more *execution statements* such that the test API command in the *execution statements* use one or more same parameters (e.g., the test API commands *dial* and *call_transfer* have the same parameter *protocol*) the value of the parameter in the test API command in the second *execution*

statement must take the same value as the same parameter defined in the test API command at the first *execution statement*.

Control dependence. Similar to data dependences, there exist general and domain specific control dependences. There exists a general control dependence between two statements if the value of the first statement controls the execution of the second statement [17]. For example, in the sequence, **if** (*protocol = SIP*) **then** *accept*, the statement *accept* depends on the predicate statement **if** (*protocol = SIP*) since the value of *protocol* determines if *accept* is executed. As in data dependence, domain specific control dependence is based on domain specific rules based on the statement information document (as illustrated in Fig. 2). For example, in the context of the video conferencing company, if there are two *execution statements* (e.g., one with test API command *dial* and the other with *call_transfer*), the execution of the second *execution statement* (i.e., *call_transfer*) depends on the execution of the first *execution statement*. To capture this dependence, we keep track of the statement execution order at the *test method* in the test case.

4.3 Test Case Implanter

The *test case implanter* component includes two steps: *test case selection* and *test case implantation*. The first step is to select test cases from the original test suite for implantation. To this end, we use decision variables (Section 4.3.1) to guide the search for selecting test cases based on the defined fitness function. The second step is to implant the selected test cases by changing statement(s) (e.g., adding new statement) in *test methods* using a *mutation operator* with three *operations* (i.e., *addition*, *modification*, and *deletion*).

4.3.1 *Solution Representation*

We represent each solution at two different levels: test suite level and test case level, as shown in Fig. 3. At the test suite level, test cases in solution s are represented with an array of real variables, $V = \{v_1, \dots, v_n\}$, where each variable v_i is associated with test case t_i (Fig. 3). The value of v_i ranges from 0 to 1, and this value indicates whether t_i is selected for implantation in s. A value greater than 0.5 indicates that the test case is selected for implantation, while a value less than or equal to 0.5 indicates otherwise. Initially, each variable in V (i.e., v_i) is assigned a random value from 0 to 1, and during the search, the *test case implanter* component of SBI returns the solutions guided by the fitness functions defined in the next section.

Test Case	t_1	t_2	t_3	…	t_{n-2}	t_{n-1}	t_n
Variable	v_1	v_2	v_3	…	v_{n-2}	v_{n-1}	v_n

Test Suite Level

Test Method	$tm_{i,1}$	…	$tm_{i,o}$
Statements	$st_{i1,1}, \dots, st_{i1,q}$	…	$st_{io,1}, \dots, st_{io,q}$

Test Case Level for Test Case t_i

Fig. 3. Two Different Levels in a Solution

At the test case level, a particular test case is composed of a number of *test methods* and each *test method* includes a set of statements. Fig. 3 depicts a set of *test methods* ($tm_{i,j}$) for test case t_i with the total number of *test methods* being i, o.

4.3.2 *Fitness Function*

Recall that the goal of SBI is to cost-effectively implant existing test cases, while 1) maximizing the effectiveness (i.e., $NCVV$ and $PCVV$) and 2) minimizing the cost (i.e., NIT, NCS, and EET). With this goal in mind, we have defined the five cost-effectiveness measures in Section 3.3. Since values of different cost and effectiveness measures are not comparable, we use the normalization function suggested in [19] to normalize the values in the same magnitude of 0 and 1 for all the five cost and effectiveness measures: $Nor(F(x)) = \frac{F(x)}{F(x)+1}$, where $F(x)$ is a function for NIT, NCS, EET, $NCVV$, and $PCVV$ (equations 4 - 8). Thus, for the five cost and effectiveness measures (equations 4 – 8), we define the following five objective functions: $F(O_1) = Nor(NIT), F(O_2) = Nor(NCS), F(O_3) = Nor(EET)$
$F(O_4) = 1 - Nor(NCVV), F(O_5) = 1 - Nor(PCVV)$

Note that we define our multi-objective search problem as a minimization problem, i.e., a solution with a lower value for an objective implies a better performance of a solution. Therefore, we subtracted 1 for the effectiveness measures: $F(O_4)$ and $F(O_5)$.

4.3.3 *Test Case Implantation*

Test case implantation occurs at the test case level (Fig. 3). For this, values of configuration variables in *configuration statements* or values of parameters in *execution statements* are based on their allowed values provided in the statement information document (Fig. 2). When changing values of parameters, the pairwise testing strategy is applied as explained in Section 3.1. Recall that the statement classification is provided by the *test case analyzer* component (Section 4.2). When a particular statement in a test method is changed, forward and backward slicing [20] is applied to obtain affected statements of the test method (using the program dependence graph from the *test case analyzer* component) that should be changed as well. A slice refers to a set of statements that influence the value of a variable at a particular program location (i.e., the location of the changed statement in this context) [21]. The process is described in detail in the next section.

Search Operators at the Test Suite Level: The *test case analyzer* component integrates the defined fitness function into a multi-objective search algorithm, NSGA-II [3], which has achieved promising results when addressing a variety of software engineering problems [22, 23]. We chose the widely used tournament selector [3] as the selection operator to select individual solutions with the best fitness for inclusion into the next generation. The crossover operator is applied at the test suite level, which randomly swaps parts of two parent solutions (i.e., test suites) to produce two offspring solutions. To this end, we chose a single point crossover operator that randomly selects the same point in both the parent solutions for generating the offspring solutions as shown in Fig. 4.

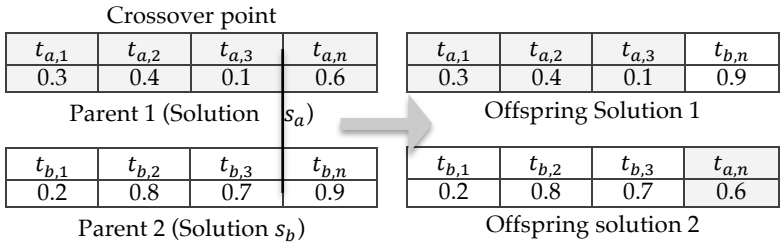


Fig. 4. Single Point Crossover applied between Two Solutions

The generated offspring solutions contain the test cases and the variable values associated with the test cases from the parent solutions as shown in Fig. 4. Note that we do not apply the crossover operator at the test case level since the *setup class*, *teardown*, and *teardown class* required for running the test methods may vary across test cases, which might lead to semantically incorrect test cases.

We apply the *mutation operator* at both the test suite and test case levels (Fig. 3). In terms of the test suite level, the *mutation operator* is defined to randomly swap the values of two variables (Section 4.3.1) based on the pre-defined mutation probability (e.g., $1/(size\ of\ the\ test\ suite)$), and recall that each variable represents a test case in our context. After the values of the variables have been swapped, if the value of the variable is greater than 0.5, the test case represented by the variable is selected for implantation. For example, in Fig. 4, if the *mutation operator* is applied to swap the values of the variables representing $t_{a,1}$ and $t_{b,n}$ in the offspring solution 1, the variable representing $t_{a,1}$ will have a new value 0.9 while the variable representing $t_{b,n}$ will have the value 0.3. This causes $t_{a,1}$ to be selected for implantation instead of $t_{b,n}$.

Mutation Operator at the Test Case Level: With respect to the test case level (Fig. 3), we defined three operations for the mutation operator: modification, addition, and deletion inspired from the work in [24, 25]. Each operation is randomly chosen with a probability of 1/3 for each test case selected for the implantation. Therefore, on average, at least one operation is applied to the selected test case for implantation. Moreover, for a test case t_i with the number of test methods as o, i.e., $t_i = \{tm_{i,1}, \dots, tm_{i,o}\}$ in Fig. 3, each test method is mutated with a probability $1/o$ using the chosen operation (e.g., addition) for the mutation operator. Note that the operation is applied to the *configuration* or *execution statement* in a test case (Table 2) because they determine the functionality of the SUT that is being tested. If tm_c is the test method to be changed in the test case t_i with the number of configuration and execution statements e, each configuration or execution statement is mutated with a probability of $1/e$. Suppose st_c is the statement to be changed, we explain the three operations for the mutation operator below.

Modification operation. The value of the configuration variable or parameter for the test API command in st_c is randomly changed to cover an uncovered 1) value of the configuration variable for *configuration statement* or 2) pairwise coverage of parameter values of the test API commands for *execution statement*. After the statement st_c is modified, if there exists statement(s) dependent on st_c (Section 4.2), they are also modified using the statement dependencies obtained from the *test case*

analyzer component (Section 4.2). For example, in Table 1 if the *modification operation* is applied to the *execution statement* at line 4, i.e., the test API command *dial*, the values of the parameters are randomly changed to increase the pairwise coverage. If the parameter *callrate* in *dial* (pointing to *callrate_var*) is required to be modified from *6000* to *64*, the variable *callrate_var* is modified to *64*.

Addition operation. A copy of statement st_c (i.e., $st_{c'}$) is created with the random uncovered 1) value of the configuration variable for *configuration statement* or 2) pairwise coverage of parameter values of the test API commands for *execution statement*. The new statement $st_{c'}$ is then added to a new *test method* $tm_{c'}$, and $tm_{c'}$ is filled with all the statements dependent on st_c in tm_c (Section 4.2). If the values of the statement(s) depend upon st_c', the dependent statement (s) are also modified after adding to the new *test method*. In the running example in Table 1, if the configuration variable *packetlossresilence* at line 2 is selected for applying the *addition operation*, *test case analyzer* will add a new statement *packetlossresilence* with the uncovered value (i.e., *on*) in a new *test method*. Since the statements in lines 4, 5, and 6 are control dependent on line 2 (Table 1), they are also added in the new *test method*. Moreover, the statement in line 4 is also added in the new *test method* since it is data dependent on line 3 (Table 1).

Deletion operation. A statement st_c is deleted from *test method* tm_c if the values of the configuration variables or the parameters (for the pairwise coverage of parameter values of test API commands) tested by tm_c have been already covered by other test cases in solution s. For example, if the *deletion operation* is applied in line 4 at Table 1 (i.e., statement with *dial*) and the parameter values for *dial* (line 4 in Table 1) are covered by another test case in solution s, then line 4 is removed from the *test method* in Table 1. Moreover, lines 2, 3, 5, and 6 are dependent on line 4 in Table 1, and therefore removed.

5. EXPERIMENT DESIGN

In this section, we describe the experiment design (as shown in Table 3), which includes the case studies (Section 5.1), research questions (Section 5.2), experiment tasks (Section 5.3), and statistical tests along with the experiment settings (Section 5.4).

5.1 Case Studies

To evaluate SBI, we chose one industrial case study from the video conferencing company referred as CS_1, and the open source case study of SafeHome [4] referred as CS_2. The industrial case study focuses on automated testing of large-scale VCSs developed by the video conferencing company. Each VCS has an average of three million lines of embedded C code and requires a thorough testing before releasing them to the market. We chose a test suite containing 118 test cases for evaluation, where on average, each test case consists of 4 *test methods* and 30.8 statements (as defined in Section 2).

Table 3. An Overview of the Experiment Design

RQ	Task	Comparisons	Case Study	Evaluation Metrics	Statistical Tests
1	J_1	Coverage of configuration variable values	CS_1, CS_2	$NCVV$ $PCPV$	One-sample Wilcoxon test
	J_2	Pairwise coverage of parameter values of test API commands			
2	J_3	Estimated execution time		EET	
	J_4	Number of implanted test cases and changed statements		NIT, NCS	
3	J_5	Statement coverage	CS_2	SC	One-sample Wilcoxon test
	J_6	Branch coverage		BC	
4	J_7	Mutation score		MS	

Moreover, the SafeHome case study was constructed based on the open source implementation of a home security and surveillance system [26], which consists of in total 13 Java classes. Each class has on average 263.4 lines of Java code and the detailed description of these classes can be consulted in [4]. Notice that the original implementation reported in [26] includes only 9 configuration variables (i.e., 6 Boolean variables and 3 Integer variables) and lacks sufficient parameters to evaluate SBI (i.e., most of the test API commands have 0 or 1 parameter). Therefore, we extended the SafeHome case study by adding: 1) additional 19 configuration variables that include 8 String variables with on average 5 values to configure for each, 2 Integer variables, and 9 Boolean variables and 2) in total 37 methods in the source code (e.g., *createUser*). 3,424 lines of non-comment Java source code (calculated using sloccount [27]) were added in total for the case study.

To obtain the original test suite for implantation for the SafeHome case study, we applied EvoSuite [28] to automatically generate in total 94 test cases (as the original test suite for implantation) including an average of 2.4 *test methods* and 19 statements for each test case. Note that our aim is to implant the original test suite for increasing the configuration coverage rather than comparing the performance between SBI and EvoSuite. To make the experiment reproducible, we have made the extended SafeHome case study publically available at [29].

5.2 Research Questions

To evaluate SBI, we aim at addressing the following four research questions (RQs).

RQ1 (Effectiveness). Can SBI significantly increase (i) the coverage of configuration variable values and (ii) pairwise coverage of the parameter values of test API commands (as defined in Section 3.3.2)?

RQ2 (Acceptability). Can the implanted test suites maintain an acceptable cost without largely increasing the test case execution time?

RQ3 (Coverage). Can SBI significantly increase the code coverage in terms of statement coverage (SC) and branch coverage (BC)?

RQ4 (Mutation Analysis). Can the implanted test suites produced by SBI significantly improve the mutation score as compared with the original test suite? We performed mutation analysis to further assess the fault detection capability achieved by SBI [30-33]. Notice that since we do not have access to the source code of our industrial case study from the video conferencing company (i.e., CS_1) due to confidential concerns, RQ3 and RQ4 are only addressed using CS_2.

5.3 Experiment Tasks and Evaluation Metrics

As shown in Table 3, we designed seven tasks (J_1-J_7) to answer the RQs. Tasks J_1 and J_2 are designed to address RQ1. More specifically, J_1 is designed to compare the coverage of the configuration variable values achieved by an implanted test suite (T') produced by SBI and the original test suite (T) using the evaluation metric $NCVV$ (equation 7). J_2 was conducted to compare the pairwise coverage of parameter values of test API commands achieved by T and T' for both CS_1 and CS_2 with the evaluation metric $PCPV$ (equation 8).

To address RQ2, J_3 and J_4 were performed to evaluate the acceptability of the implanted test suites produced by SBI in terms of execution time, number of implanted test cases, and changed statements. The cost measures (i.e., EET, NIT, and NCS) were used as the evaluation metrics for J_3 and J_4. For RQ3, J_5 and J_6 were conducted to measure the code coverage with the evaluation metrics SC and BC. SC measures the number of statements in the source code that are executed when executing a given test suite, while BC measures the number of possible branch(es) from each decision point that is executed [34]. Finally, RQ4 is addressed by task J_7 using the mutation score (MS) [35] as the evaluation metric, which is widely used [30-33] to measure the fault detection capability of the test suite. MS is the ratio of killed mutants out of the total number of non-equivalent mutants [35].

5.4 Statistical Tests and Experiment Settings

5.4.1 Statistical Tests

To choose an appropriate statistical test, we first performed the Shapiro-Wilk test [36, 37] to assess whether the data samples produced are normally distributed. The results of the Shapiro-Wilk test showed the obtained data samples were not normally distributed, and thus we chose the one-sample Wilcoxon test as recommended in [38] to statistically evaluate results of RQ1, RQ3, and RQ4 (Table 3). We used the one-sample Wilcoxon test (p-value) since the coverage of the original test suite (e.g., $NCVV$, SC) is fixed, and we chose the significance level of 0.05, i.e., there is a significant difference if the p-value is less than 0.05. Moreover, we compare mean values for the coverage of the original test suite and the test suites implanted by SBI to see in which direction the results are significant, i.e., which approach is better when the p-value is less than 0.05.

5.4.2 *Experiment Settings*

SBI is implemented using a Java framework jMetal [39], which has been widely used for various multi-objective optimization problems [40-42]. Recall that SBI is designed on the top of NSGA-II, and we used the standard settings of NSGA-II for configuring the parameters of SBI as suggested in [38], except for the mutation rate at the test case level (as discussed in Section 4.3.3) to be able to change each test case selected for implantation. More specifically, we chose 1) the population size as 100, 2) crossover rate as 0.9, 3) mutation rate at the test suite level as $1/(Total\ number\ of\ test\ cases)$, 4) the

mutation rate at the test case level is 1/3 for each test case selected for implantation, and 5) the maximum number of fitness evaluation (i.e., termination criteria of the algorithm) is set as 25,000. Note that parameter tuning may lead to different performance of search algorithms, but standard parameter settings are usually recommended [38].

Regarding *SC* and *BC* (RQ3), we used the open source tool Eclemma [43] to measure the *SC* and *BC* achieved by the implanted test suites and the original one. For mutation analysis (RQ4), we used the Java-based mutation tool PIT [44], which has been extensively used in mutation testing research [45, 46]. All the seven basic *mutation operators* in PIT (i.e., conditionals boundary, increments, invert negatives, math, negate conditionals, return values, and void method calls) were applied and 1594 non-equivalent mutants were generated for CS_2. In addition, we ran SBI 10 times to account for the random variation for each case study since SBI is built on top of NSGA-II [38].

6. Results, Analysis, and Overall Discussion

This section presents the result and analysis of the four research questions in Section 6.1 - 6.4 and the overall discussion in Section 6.5.

6.1 RQ1. Effectiveness of *SBI*

Recall that RQ1 aims to assess the effectiveness of the implanted test suites produced by SBI in terms of the two effectiveness measures (described in Section 3.3.2): number of configuration variable values covered ($NCVV$) and pairwise coverage of parameter values of test API commands ($PCPV$). The mean difference between the values produced by the implanted test suites and the original test suite is 19.6 and 112.2 for $NCVV$ and $PCPV$ respectively in CS_1 and 11.5 and 156.9 for $PCPV$ in CS_2. Moreover, all the mean differences are statistically significant since all the p-values are less than 0.001 (from the one-sample Wilcoxon test), which shows that SBI managed to perform significantly better than the original test suite in terms of $NCVV$ and $PCPV$.

Table 4 summarizes the values for different evaluation metrics achieved by SBI for the 1000 implanted test suites (i.e., solutions) and the original test suites for the two case studies (i.e., CS_1 and CS_2). Recall from Section 5.4.2 that SBI is executed 10 times, and each run produces 100 optimal solutions. For the effectiveness measures (i.e., $NCVV$ and $PCPV$), we can conclude from Table 4 that on average SBI managed to achieve 22.8% higher $NCVV$ and 52.9% higher $PCPV$ for CS_1 and 20.9% and 65.9% higher $PCPV$ for CS_2.

Table 4. Results of Different Evaluation Metrics for the Original Test Suites and Implanted Test Suites*

CS	Test Suite	*NCVV* Values		*PCPV* Values		*EET* Values		*NIT* Values		*NCS* Values		*SC* Values		*BC* Values	
		Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD
CS_1	SBI	105.6	12.3	324.2	80.1	9607s	2745.1s	42.8	16.8	674.8	819.8				
	Original	86		212		8222s									
CS_2	SBI	66.5	6	394.9	123.5	3.8s	0.3s	32.8	14.1	760.3	832.9	71.5%	3.3%	57.4%	5.3%
	Original	55		238		3.5s						66.7%		49.9%	

*SD: standard deviation, s: seconds.

Since the results from CS_1 and CS_2 are consistent and all the implanted test suites have significantly higher coverage $NCVV$ and $PCPV$ as compared to the original test suite, we can answer RQ1 as: the implanted test suites produced by SBI achieved significantly higher effectiveness than the original one, which demonstrates that SBI is effective.

6.2 RQ2. Acceptability of *SBI*

We can observe from Table 4 that the implanted test suites produced by SBI require on average 16.8% (1385 seconds) and 8.6% (0.3 second) more EET for CS_1 and CS_2, respectively as compared to the original test suite. Such an increase in the execution time is practically acceptable for both the case studies since the effectiveness of the test suite is significantly improved (e.g., 52.9% higher $PCPV$ for CS_2, 65.9% higher $PCPV$ for CS_1). Therefore, we conclude that SBI can maintain acceptable cost without largely increasing the test case execution time indicating that SBI is cost-effective.

6.3 RQ3. Code Coverage of Implanted Test Suite by *SBI*

This RQ aims to evaluate whether SBI can increase the overall code coverage in terms of *SC* and *BC* using the SafeHome case study (i.e., CS_2). From Table 4, we can observe that the mean difference between the values produced by the implanted test suites and the original test suite is 4.8% and 7.5% for *SC* and *BC* respectively. Additionally, all the mean differences are statistically significant since the p-values are less than 0.001 (obtained from the one-sample Wilcoxon test). Thus, we summarize that SBI can significantly increase the code coverage of the original test suite.

6.4 RQ4. Mutation Score

Recall that this RQ aims to check whether the test suites implanted by SBI has a higher mutation score (*MS*) than the original test suite. Recall from Section 5.4.2 that each execution of SBI produces 100 optimal solutions, and it is quite expensive to perform mutation analysis for all the 1000 solutions (produced by executing SBI 10 times) since it takes more than four minutes to perform mutation analysis for one solution. Thus, we chose only two solutions produced in each run of SBI to perform mutation analysis. Based on the existing work [47, 48], we chose the solutions based on the following two ways: 1) random, referred as *random solution* and 2) highest average value of all the defined cost-effectiveness measures (Section 3.3) referred as *selected solution.* Table 4

Table 5 summarizes the results of *MS* for the original test suite, the average of 10 *random solutions*, and 10 *selected solutions*. Moreover, Table 5 shows the result of the mean difference and the one-sample Wilcoxon test between the *MS* produced by the original test suite and 10 *random solutions* and 10 *selected solutions*. Based on Table 5, we can conclude for RQ4 that the solutions implanted by SBI have a significantly higher *MS* since the p-values for the random solutions and selected solutions are less than 0.05 and the mean difference is positive (e.g., 3.1% indicating that the *selected solutions* improved *MS* on average 3.1% as compared to the original test suite). Thus, we can answer RQ4 as SBI can detect more faults (as indicated by a higher *MS*).

Table 5. Results of RQ4*

Solution	*MS*	Mean Difference (RA/SA-Original)	*p*-value
Original	33.9%		
RA	36%	2.1%	**0.002**
SA	37%	3.1%	**0.002**

*RA: Average *MS* for the random solutions, SA: Average *MS* for the selected solutions.

Notice that running time is an important perspective when evaluating a search-based approach [22, 49], and thus we report the running time of SBI. SBI took an average of 72.3 minutes and 78.1 minutes for CS_1 and CS_2, respectively. Such running time has no practical impact on the use of our approach since test case implantation is a one-time effort for a given test suite.

6.5 Overall Discussion

For RQ1 and RQ2, we observed that SBI managed to significantly increase the effectiveness of the original test suite (measured by $NCVV$ and $PCPV$) without significantly increasing the cost (measured by EET). The reason behind this is because SBI modifies the original test cases by changing (i.e., modifying/adding/removing) the statements to maximize the effectiveness measures and minimize the cost measures as defined in Section 3.3. Thus, we conclude that SBI is a cost-effective approach for tackling the test case implantation problem.

Regarding RQ3, the results showed that SBI did not manage to improve the *SC* and *BC* by a large percentage for CS_2. This is because SBI cannot further increase the code coverage if the original test suite has already covered all the parameters of a method in the source code or some methods are not targeted at all by the original test suite, which can be considered as the limitation of SBI and will be further investigated in the future. For instance, in the class *SensorTest* (available in [29]), all the parameters in the method *armMotionDetector* have already been tested by the original test suite while the method *actionPerformed* in the class was not targeted by the original test suite. Thus, SBI was not able to increase the code coverage for the class *SensorTest*.

Furthermore, Fig. 5 presents *SC* and *BC* for the original test suite and the implanted test suites by SBI for the 13 classes and the overall coverage (i.e., *Total*) that is the ratio of the total number of statements/branches covered and total number of statements/branches present in the source code in CS_2. From Fig. 5, we can observe that SBI managed to improve *SC* and *BC* for 6 of the 13 classes (e.g., on average 15.2% higher *SC* and 14.9% higher *BC* for class *User* in Fig. 5), and there was no change for the remaining 7 classes (e.g., *ControlPanel*) since all the parameters in the methods have already been tested or the methods are not targeted by the original test suite.

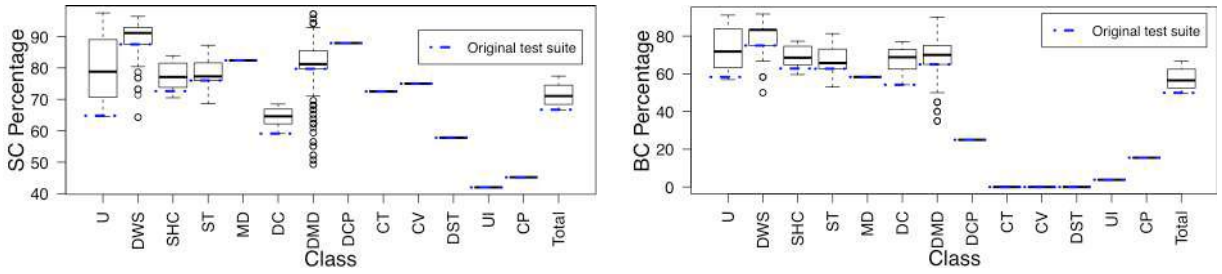
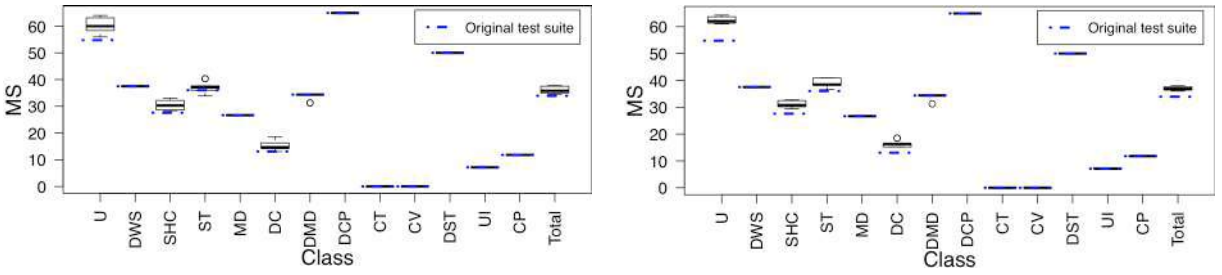


Fig. 5. *SC* and *BC* for the Original Test Suite and Test Suites Implanted using SBI*



Random solution Selected solution

Fig. 6. *MS* for the Original Solution compared with Random and Selected solutions*

*U: user, DWS: DeviceWindowSensor, SHC: SafeHomeController, ST: SensorTest, MD: MainDemo, DC: DeviceCamera, DMD: DeviceMotionDetector, DCP: DeviceControlPanelAbstract, CT: CameraTest, CV: CameraView, DST: DeviceSensorTester, UI: UserInterface, CP: ControlPanel.

Regarding RQ4, SBI increased *MS* for 4 classes (out of 13 classes) in CS_2 as shown in Fig. 6. Note that *MS* did not increase in all the classes where *SC* and *BC* increased. For instance, *MS* increased in class User where *SC* and *BC* also grew. Class *DeviceWindowSensor* [29] had an increasing *SC* and *BC*, but *MS* remained similar as compared with the original test suite. Such observation is consistent with the findings of the state-of-the-art [30] showing that the code coverage (e.g., *SC*) is not strongly correlated with test suite effectiveness (e.g., *MS*).

Finally, for the real industrial case study, SBI needs to be run only once in practice (using 72.3 minutes) to obtain an implanted test suite (including 118 test cases). This is equivalent to modifying one test case using on average 0.61 minutes (72.3/118) or 37 seconds. Clearly, modifying a test case manually within 37 seconds is practically not possible. In addition, SBI produces the optimal implanted test cases that satisfy various cost and effectiveness objectives (Section 3.3). Thus, we can conclude SBI is beneficial in practice, at least for our industrial case study.

7. THREATS TO VALIDITY

This section presents some of the potential threats to the validity of the two case studies investigated in this paper.

Threats to *internal validity* consider the internal factors (e.g., algorithm parameters) that could influence the obtained results [50]. In our context, *internal validity* threats arise due to experiment with only one set of configuration settings for the algorithm parameters [51]. However, these settings are in accordance with the guidelines from the literature [52], and previously we have achieved good results with these settings in different software engineering problems [53, 54]. Regarding the mutation rate applied on the test case level, we chose a rate that has earlier been investigated in the literature [24].

Another threat to *internal validity* involves instrumentation effects, i.e., the quality of the coverage information and mutation score measured [22]. To mitigate this threat, we used open source tools Eclemma and PIT that have been widely used in the literature [45, 46, 55, 56].

Threats to *external validity* are related to the factors that affect the generalization of the results [50]. To mitigate this threat, we chose two different case studies (i.e., industrial case study and open source case study) for evaluating SBI. We plan to conduct more case studies in the future to generalize the results. It is also worth mentioning that such threats to *external validity* are common in empirical studies [23, 57]. Another *external validity* threat is due to the selection of a search algorithm for SBI. To reduce this threat, we selected the most widely used search algorithm (NSGA-II) that has been widely applied in different contexts [23, 42, 57].

Threats to *construct validity* arise when the measurement metrics do not sufficiently cover the concepts they are supposed to measure [22]. To mitigate this threat, we compare the implanted test suites by SBI and the original test suite based on evaluation metrics that have been widely adopted in the literature: statement coverage, branch coverage, mutation score, and running time.

Threats to *conclusion validity* are related to the factors that influence the conclusion drawn from the results of the experiments [58]. The *conclusion validity* threat when using randomized algorithms is related to random variation in the produced results. To mitigate this threat, we repeated each experiment 10 times for SBI to reduce the possibility that the results were obtained by chance. Moreover, we carefully applied statistical tests by following the guidelines of reporting results for randomized algorithms [38].

8. RELATED WORK

There are a number of existing works related to code transplantation, test suite augmentation, test generation, and testing of highly configurable software systems that have certain similarities with our work (i.e., test case implantation). We discuss each of them in detail as below.

8.1 Code Transplantation

In recent years, there has been an increasing attention on code transplantation within/across software systems [59-62]. For instance, Weimer et al. [59] used genetic programming (GP) to evolve defective programs to fix defects while maintaining specified functionalities for automatic program repair. Petke et al. [61] used GP to evolve a program by transplanting code from other programs for improving system's performance. Barr et al. [60] automatically transplanted functionalities of programs across different software systems using GP and program slicing.

As compared with the existing work for code transplantation (e.g., [59-61]), SBI has at least two key differences: 1) The goal is different, i.e., we aim at automatically implanting existing test cases to test untested configurations rather than transplanting software code; 2) Five objectives (e.g., maximizing the number of configuration variable values covered) are defined to guide the search for selecting and implanting test cases.

8.2 Test Suite Augmentation

There is a number of studies focusing on test suite augmentation that refers to identifying code elements affected by software changes as it evolves (e.g., new functionalities are added), and generating test cases to test those elements [63-67]. For instance, dependence analysis and partial symbolic execution were used in [65] to identify the changed test requirements when the program is evolved, however they do not generate test cases. In [63] a directed test suite augmentation technique was proposed for 1) identifying the code affected by changes in the program and 2) generating new test cases for testing the affected code using a concolic test case generation approach [68].

As compared with the above-mentioned literature, SBI aims to cost-effectively increase the configuration coverage of the original test suite rather than generating new test cases for testing the modified code. Furthermore, we defined three operations (i.e., *addition*, *modification* and *deletion*) to automatically implant the test cases, which is not the case in the existing work.

8.3 Test Generation

Different techniques have been used for test generation such as random testing [69], symbolic execution [70, 71] and search techniques [24, 28, 72-74] (that is the most relevant to this work). For instance, Miller et al. [73] used program dependence graphs and genetic algorithm to generate test data for maximizing condition-decision coverage. Ali et al. [72] designed a search-based Object Constraint Language (OCL) constraint solver by defining branch distance functions to support test data generation for model-based testing. Fraser and Arcuri [24, 28] designed and implemented a tool (i.e., EvoSuite) to generate test cases with an aim to maximize different coverage criteria (e.g., line, branch) and mutation testing using search. As compared with the state-of-the-art for test generation, SBI focuses on automated implanting an existing test suite to test untested configurations instead of generating test cases from scratch.

8.4 Testing of Highly Configurable Software Systems

There is a large body of research with respect to the testing of highly configurable software systems with many configurations [75-81]. Existing works have proposed many sampling techniques to select a subset of representative configurations for testing [78-81]. For instance, Swanson [78] modeled a highly configurable system using feature model followed by applying random sampling to repeatedly generate a random configuration from the feature model for testing. Qu et al. [79] and Yilmaz et al. [80] used covering array sampling method to generate at least one *t*-combination configuration (to be tested) for representing all valid *t*-combination configurations in the configuration space. Cohen et al. [82] combined pairwise algorithms (e.g., meta-heuristic search algorithm) with Boolean satisfiability (SAT) solvers to handle constraints while generating configurations for interaction testing of highly configurable systems.

As compared with the existing studies of testing highly configurable software systems, the focus of our work is totally different, i.e., we aim at implanting an original test suite to test untested

configurations, and thus increase the configuration coverage of the existing test suite. To achieve this goal, we proposed a search-based approach (i.e., SBI) for automated test case implantation (Section 4).

9. Conclusion

This paper introduced a novel search-based test case implantation approach (SBI) including two key components (i.e., *test case analyzer* and *test case implanter*) with the aim to automatically analyze and implant the existing test cases to test the untested configurations. SBI was evaluated using one industrial and one open source case study. The results showed that the implanted test suites produced by SBI performed significantly better than the original test suite for both the case studies. More specifically, SBI significantly outperformed the original suite for both the case studies by achieving on average 21.9% higher number of configuration variables values and 59.4% higher pairwise coverage of parameter values of test API commands. Moreover, for the open source case study, the implanted test suites managed to improve statement coverage, branch coverage, and mutation score with on average 4.8%, 7.5%, and 2.6%, respectively.

As future work, we first plan to apply more case studies to further strengthen the applicability of SBI. We also want to evaluate the performance of SBI by integrating with other multi-objective search algorithms (e.g., Strength Pareto Evolutionary Algorithm (SPEA2) [83]).

Acknowledgment

This research was supported by the Research Council of Norway (RCN) funded Certus SFI. Shuai Wang is also supported by RFF Hovedstaden funded MBE-CR project and COST Action CA15140 (ImAppNIO). Tao Yue and Shaukat Ali are also supported by RCN funded Zen-Configurator project, the EU Horizon 2020 project funded U-Test, RFF Hovedstaden funded MBE-CR project, and RCN funded MBTCPS project.

REFERENCES

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

[2] Asadollah, S. A., Inam, R. and Hansson, H. 2015. A Survey on Testing for Cyber Physical System. IFIP International Conference on Testing Software and Systems.

[3] Deb, K., Pratap, A., Agarwal, S. and Meyarivan, T. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation 6, 2.

[4] SafeHome Project. https://github.com/Suckzoo/CS350.

[5] Unit testing framework. https://docs.python.org/2/library/unittest.html. 2017.

[6] Korel, B. and Al-Yami, A. M. 1998. Automated regression test generation. ACM SIGSOFT Software Engineering Notes 23, 2.

[7] Pandita, R., Xie, T., Tillmann, N. and De Halleux, J. 2010. Guided test generation for coverage criteria. IEEE International Conference on Software Maintenance (ICSM), 2010.

[8] Kosmatov, N., Legeard, B., Peureux, F. and Utting, M. 2004. Boundary coverage criteria for test generation from formal models. 15th International Symposium on Software Reliability Engineering, 2004. ISSRE 2004.

[9] Czerwonka, J. 2006. Pairwise testing in the real world: Practical extensions to test-case scenarios. Proceedings of 24th Pacific Northwest Software Quality Conference, Citeseer.

[10] Nie, C. and Leung, H. 2011. A survey of combinatorial testing. ACM Computing Surveys (CSUR) 43, 2.

[11] Blizard, W. D. 1988. Multiset theory. Notre Dame Journal of formal logic 30, 1.

[12] Kuhn, R., Kacker, R., Lei, Y. and Hunter, J. 2009. Combinatorial software testing. Computer 42, 8.

[13] Lei, Y. and Tai, K.-C. 1998. In-parameter-order: A test generation strategy for pairwise testing. Third IEEE International Symposium on High-Assurance Systems Engineering.

[14] Athanasiou, D., Nugroho, A., Visser, J. and Zaidman, A. 2014. Test code quality and its relation to issue handling performance. IEEE Transactions on Software Engineering 40, 11.

[15] Heitlager, I., Kuipers, T. and Visser, J. 2007. A practical model for measuring maintainability. 6th International Conference on the Quality of Information and Communications Technology, 2007. QUATIC 2007.

[16] Ottenstein, K. J. and Ottenstein, L. M. 1984. The program dependence graph in a software development environment. ACM Sigplan Notices.

[17] Ferrante, J., Ottenstein, K. J. and Warren, J. D. 1987. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems (TOPLAS) 9, 3.

[18] Zhao, J. 1998. Applying program dependence analysis to Java software. Proceedings of Workshop on Software Engineering and Database Systems, 1998.

[19] Ali, S. and Yue, T. 2014. Evaluating Normalization Functions with Search Algorithms for Solving OCL Constraints. Springer.

[20] Komondoor, R. and Horwitz, S. 2001. Using slicing to identify duplication in source code. International Static Analysis Symposium.

[21] Weiser, M. 1981. Program slicing. Proceedings of the 5th International Conference on Software Engineering.

[22] Li, Z., Harman, M. and Hierons, R. M. 2007. Search algorithms for regression test case prioritization. IEEE Transactions on Software Engineering 33, 4.

[23] Sarro, F., Petrozziello, A. and Harman, M. 2016. Multi-objective software effort estimation. Proceedings of the 38th International Conference on Software Engineering.

[24] Fraser, G. and Arcuri, A. 2013. Whole test suite generation. IEEE Transactions on Software Engineering 39, 2.

[25] Fraser, G. and Arcuri, A. 2011. Evolutionary generation of whole test suites. 11th International Conference on Quality Software (QSIC), 2011.

[26] Pressman, R. S. 2005. *Software engineering: a practitioner's approach*. Palgrave Macmillan.

[27] Wheeler, D. Sloccount. https://www.dwheeler.com/sloccount/.

[28] Fraser, G. and Arcuri, A. 2011. Evosuite: automatic test suite generation for object-oriented software. Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering.

[29] Supplimentary material. https://sbi.netlify.com/

[30] Inozemtseva, L. and Holmes, R. 2014. Coverage is not strongly correlated with test suite effectiveness. Proceedings of the 36th International Conference on Software Engineering.

[31] Kracht, J. S., Petrovic, J. Z. and Walcott-Justice, K. R. 2014. Empirically evaluating the quality of automatically generated and manually written test suites. 14th International Conference on Quality Software (QSIC), 2014

[32] Just, R., Jalali, D., Inozemtseva, L., Ernst, M. D., Holmes, R. and Fraser, G. 2014. Are mutants a valid substitute for real faults in software testing? Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.

[33] Gopinath, R., Jensen, C. and Groce, A. 2014. Code coverage for suite evaluation by developers. Proceedings of the 36th International Conference on Software Engineering.

[34] Rothermel, G., Untch, R. H., Chu, C. and Harrold, M. J. 1999. Test case prioritization: An empirical study. Proceedings of the IEEE International Conference on Software Maintenance, 1999.(ICSM'99).

[35] Offutt, A. J., Rothermel, G. and Zapf, C. 1993. An experimental evaluation of selective mutation. Proceedings of the 15th international conference on Software Engineering.

[36] Shapiro, S. S. and Wilk, M. B. 1965. An analysis of variance test for normality (complete samples). Biometrika 52, 3-4.

[37] Sheskin, D. J. 2003. *Handbook of parametric and nonparametric statistical procedures*. crc Press.

[38] Arcuri, A. and Briand, L. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. Proceedings of the 33rd International Conference on Software Engineering.

[39] Durillo, J. J. and Nebro, A. J. 2011. jMetal: A Java framework for multi-objective optimization. Advances in Engineering Software 42, 10.

[40] Nebro, A. J., Durillo, J. J., Garcia-Nieto, J., Coello, C. C., Luna, F. and Alba, E. 2009. Smpso: A new pso-based metaheuristic for multi-objective optimization. Proceedings of the Symposium on Computational Intelligence in Multi-criteria Decision-making.

[41] Durillo, J. J., Nebro, A. J., Luna, F. and Alba, E. 2008. Solving three-objective optimization problems using a new hybrid cellular genetic algorithm. Springer.

[42] Sayyad, A. S., Menzies, T. and Ammar, H. 2013. On the value of user preferences in search-based software engineering: a case study in software product lines. Proceedings of the 35th International Conference on Software Engineering.

[43] Java Code Coverage for Eclipse. http://www.eclemma.org/

[44] Coles, H., Laurent, T., Henard, C., Papadakis, M. and Ventresque, A. 2016. PIT: a practical mutation testing tool for Java. Proceedings of the 25th International Symposium on Software Testing and Analysis.

[45] Inozemtseva, L., Hemmati, H. and Holmes, R. 2013. Using fault history to improve mutation reduction. Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering.

[46] Zhang, J., Wang, Z., Zhang, L., Hao, D., Zang, L., Cheng, S. and Zhang, L. 2016. Predictive mutation testing. Proceedings of the 25th International Symposium on Software Testing and Analysis.

[47] Zhang, M., Ali, S., Yue, T. and Hedman, M. 2016. Uncertainty-based Test Case Generation and Minimization for Cyber-Physical Systems: A Multi-Objective Search-based Approach. Simula Research Laboratory.

[48] Wang, S., Ali, S. and Gotlieb, A. 2015. Cost-effective test suite minimization in product lines using search techniques. Journal of Systems and Software 103.

[49] Yoo, S. and Harman, M. 2010. Using hybrid algorithm for pareto efficient multi-objective test suite minimisation. Journal of Systems and Software 83, 4.

[50] Runeson, P., Host, M., Rainer, A. and Regnell, B. 2012. *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons.

[51] de Oliveira Barros, M. and Neto, A. 2011. Threats to Validity in Search-based Software Engineering Empirical Studies. UNIRIO-Universidade Federal do Estado do Rio de Janeiro, techreport 6.

[52] Arcuri, A. and Fraser, G. 2011. On parameter tuning in search based software engineering. Springer.

[53] Wang, S., Ali, S., Yue, T., Bakkeli, Ø. and Liaaen, M. 2016. Enhancing test case prioritization in an industrial setting with resource awareness and multi-objective search. Proceedings of the 38th International Conference on Software Engineering Companion.

[54] Wang, S., Ali, S., Yue, T., Li, Y. and Liaaen, M. A Practical Guide to Select Quality Indicators for Assessing Pareto-based Search Algorithms in Search-Based Software Engineering *Proceedings of the 38th International Conference on Software Engineering*. (2016). ACM, [insert City of Publication].

[55] Kim, J., Batory, D., Dig, D. and Azanza, M. 2016. Improving refactoring speed by 10x. Proceedings of the 38th International Conference on Software Engineering.

[56] Johnson, B., Pandita, R., Smith, J., Ford, D., Elder, S., Murphy-Hill, E., Heckman, S. and Sadowski, C. 2016. A cross-tool communication study on program analysis tool notifications. Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering.

[57] Wang, S., Ali, S., Yue, T. and Liaaen, M. 2015. UPMOA: An improved search algorithm to support user-preference multi-objective optimization. Proceedings of the 26th International Symposium on Software Reliability Engineering.

[58] Wohlin, C., Runeson, P., Host, M., Ohlsson, M., Regnell, B. and Wesslen, A. 2000. Experimentation in software engineering: an introduction. 2000. Kluwer Academic Publishers.

[59] Weimer, W., Nguyen, T., Le Goues, C. and Forrest, S. 2009. Automatically finding patches using genetic programming. Proceedings of the 31st International Conference on Software Engineering.

[60] Barr, E. T., Harman, M., Jia, Y., Marginean, A. and Petke, J. 2015. Automated software transplantation. Proceedings of the 2015 International Symposium on Software Testing and Analysis.

[61] Petke, J., Harman, M., Langdon, W. B. and Weimer, W. 2014. Using genetic improvement and code transplants to specialise a C++ program to a problem class. European Conference on Genetic Programming.

[62] Sidiroglou-Douskos, S., Lahtinen, E. and Rinard, M. 2014. Automatic error elimination by multi-application code transfer.

[63] Xu, Z. and Rothermel, G. 2009. Directed test suite augmentation. Software Engineering Conference, 2009. APSEC'09. Asia-Pacific.

[64] Yoo, S. and Harman, M. 2008. Test data augmentation: generating new test data from existing test data. Centre for Research on Evolution, Search & Testing (CREST).

[65] Santelices, R., Chittimalli, P. K., Apiwattanapong, T., Orso, A. and Harrold, M. J. 2008. Test-suite augmentation for evolving software. Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering.

[66] Xu, Z., Kim, Y., Kim, M., Rothermel, G. and Cohen, M. B. 2010. Directed test suite augmentation: techniques and tradeoffs. Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering.

[67] Xu, Z., Cohen, M. B., Motycka, W. and Rothermel, G. 2013. Continuous test suite augmentation in software product lines. Proceedings of the 17th International Software Product Line Conference.

[68] Sen, K., Marinov, D. and Agha, G. 2005. CUTE: a concolic unit testing engine for C. ACM SIGSOFT Software Engineering Notes.

[69] Csallner, C. and Smaragdakis, Y. 2004. JCrasher: an automatic robustness tester for Java. Software: Practice and Experience 34, 11.

[70] Xie, T., Marinov, D., Schulte, W. and Notkin, D. 2005. Symstra: A framework for generating object-oriented unit tests using symbolic execution. International Conference on Tools and Algorithms for the Construction and Analysis of Systems.

[71] Tillmann, N. and De Halleux, J. 2008. Pex–white box test generation for. net. International conference on tests and proofs.

[72] Ali, S., Iqbal, M. Z., Arcuri, A. and Briand, L. 2011. A search-based OCL constraint solver for model-based test data generation. Quality Software (QSIC), 2011 11th International Conference on.

[73] Miller, J., Reformat, M. and Zhang, H. 2006. Automatic test data generation using genetic algorithm and program dependence graphs. Information and Software Technology 48, 7.

[74] Lakhotia, K., Harman, M. and McMinn, P. 2007. A multi-objective approach to search-based test data generation. Proceedings of the 9th annual conference on Genetic and evolutionary computation.

[75] Lochau, M., Oster, S., Goltz, U. and Schürr, A. 2012. Model-based pairwise testing for feature interaction coverage in software product line engineering. Software Quality Journal 20, 3-4.

[76] Reisner, E., Song, C., Ma, K.-K., Foster, J. S. and Porter, A. 2010. Using symbolic evaluation to understand behavior in configurable software systems. Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1.

[77] Kuhn, D. R., Wallace, D. R. and Gallo, A. M. 2004. Software fault interactions and implications for software testing. IEEE transactions on software engineering 30, 6.

[78] Swanson, J. 2014. A Self-Adaptive Framework for Failure Avoidance in Configurable Software.

[79] Qu, X., Cohen, M. B. and Rothermel, G. 2008. Configuration-aware regression testing: an empirical study of sampling and prioritization. Proceedings of the 2008 international symposium on Software testing and analysis.

[80] Yilmaz, C., Cohen, M. B. and Porter, A. A. 2006. Covering arrays for efficient fault characterization in complex configuration spaces. IEEE Transactions on Software Engineering 32, 1.

[81] Perrouin, G., Oster, S., Sen, S., Klein, J., Baudry, B. and Le Traon, Y. 2012. Pairwise testing for software product lines: comparison of two approaches. Software Quality Journal 20, 3-4.

[82] Cohen, M. B., Dwyer, M. B. and Shi, J. 2007. Interaction testing of highly-configurable systems in the presence of constraints. Proceedings of the 2007 international symposium on Software testing and analysis.

[83] Zitzler, E., Laumanns, M. and Thiele, L. 2001. SPEA2: Improving the strength Pareto evolutionary algorithm.