SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2012; **22**:67–120
Published online 11 March 2010 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/stvr.430

Regression testing minimization, selection and prioritization: a survey

S. Yoo*, † and M. Harman

King's College London, Centre for Research on Evolution, Search and Testing, Strand, London WC2R 2LS, U.K.

SUMMARY

Regression testing is a testing activity that is performed to provide confidence that changes do not harm the existing behaviour of the software. Test suites tend to grow in size as software evolves, often making it too costly to execute entire test suites. A number of different approaches have been studied to maximize the value of the accrued test suite: minimization, selection and prioritization. Test suite minimization seeks to eliminate redundant test cases in order to reduce the number of tests to run. Test case selection seeks to identify the test cases that are relevant to some set of recent changes. Test case prioritization seeks to order test cases in such a way that early fault detection is maximized. This paper surveys each area of minimization, selection and prioritization technique and discusses open problems and potential directions for future research. Copyright © 2010 John Wiley & Sons, Ltd.

Received 4 September 2008; Revised 4 January 2010; Accepted 6 January 2010

KEY WORDS: regression testing; test suite minimization; regression test selection; test case prioritization

1. INTRODUCTION

Regression testing is performed when changes are made to existing software; the purpose of regression testing is to provide confidence that the newly introduced changes do not obstruct the behaviours of the existing, unchanged part of the software. It iza complex procedure that is all the more challenging because of some of the recent trends in software development paradigms. For example, the component-based software development method tends to result in the use of many black-box components, often adopted from a third party. Any change in the third-party components may interfere with the rest of the software system, yet it is hard to perform regression testing because the internals of the third-party components are not known to their users. The shorter life-cycle of software development, such as the one suggested by the *agile* programming discipline, also imposes restrictions and constraints on how regression testing can be performed within limited resources.

Naturally, the most straightforward approach to this problem is to simply execute all the existing test cases in the test suite; this is called a *retest-all* approach. However, as software evolves, the test suite tends to grow, which means that it may be prohibitively expensive to execute the

*Correspondence to: S. Yoo, King's College London, Centre for Research on Evolution, Search and Testing, Strand, London WC2R 2LS, U.K.
†E-mail: Shin.Yoo@kcl.ac.uk

Contract/grant sponsor: EPSRC; contract/grant numbers: EP/D050863, GR/S93684, GR/T22872
Contract/grant sponsor: EU; contract/grant number: IST-33472
Contract/grant sponsor: DaimlerChrysler Berlin and Vizuri Ltd., London

entire test suite. This limitation forces consideration of techniques that seek to reduce the effort required for regression testing in various ways.

A number of different approaches have been studied to aid the regression testing process. The three major branches include test suite minimization, test case selection and test case prioritization. Test suite minimization is a process that seeks to identify and then eliminate the obsolete or redundant test cases from the test suite. Test case selection deals with the problem of selecting a subset of test cases that will be used to test the *changed* parts of the software. Finally, test case prioritization concerns the identification of the 'ideal' ordering of test cases that maximize desirable properties, such as early fault detection. Existing empirical studies show that the application of these techniques can be cost-effective.

This paper surveys work undertaken in these three related branches of regression testing. Section 2 introduces the nomenclature. Section 3 describes different test suite minimization techniques as well as their efficiency and effectiveness. Section 4 examines test case selection techniques according to the specific analysis technique used, and evaluates the strengths and weaknesses of each approach. Section 5 introduces test case prioritization techniques. Section 6 introduces meta-empirical studies concerning evaluation methodologies and cost-effectiveness analysis of regression testing techniques. Section 7 presents a summary of the field and identifies trends and issues. Section 8 introduces some gaps in the existing literature, thereby suggesting potential directions for future work. Section 9 concludes.

1.1. Motivation

When writing a survey paper there are two natural questions that need to be asked:

1. Why is this the right set of topics for a survey?
2. Is there already a recent survey in this area?

The first question concerns the motivation for the scope chosen for the survey, whereas the second concerns the perceived need for such a survey, once a suitable scope is established. For this paper the scope has been chosen to include topics on test suite minimization, Regression Test Selection (RTS) and test case prioritization. The reason for the choice of this scope is that these three topics are related by a common thread of optimization; each is an approach that optimizes the application of an existing pool of test cases.

The fact that all three approaches assume that the existence of a pool of test cases distinguishes these topics from test case generation, which seeks to create pools of test data. The three topics form a coherent set of approaches, each of which shares a common starting point; that the tester has a pool of test cases that is simply too large to allow all cases to be applied to the System Under Test (SUT). Each of the three approaches denotes a different way of coping with this problem of scale.

The relationship between the three techniques goes deeper than the mere shared application to pools of test data. It is not only the sets of problems addressed by each that exhibit overlap, but also the solution approaches that are applied. There is an intimate relationship between solutions to the three related problems, as this survey reveals. For instance, one way of selecting (or minimizing) a set of n test cases from a test pool would be to prioritize the whole set and pick the first n in priority order. Of course, there may be more optimal choices, since prioritization has to contend with any possible choice of n, whereas selection merely requires that an optimal choice is found for a given value of n.

Turning to the second question, the previous papers closest to this survey are a survey of RTS techniques undertaken in 1996 [1] and a recent systematic review on RTS [2] that concerns a specific set of research questions rather than a survey of the whole area. No previous survey on regression testing considered minimization, selection and prioritization collectively. The present survey claims that these classes of techniques are closely related to each other and denote a coherent sub-area of study. Our survey also includes recent applications of techniques that were surveyed in the earlier work of Rothermel and Harrold [1]. There are existing comparative studies on RTS [3–7], but these papers only concern a small number of specific RTS techniques and do not provide an overview of the field.

DOI: 10.1002/stvr

1.2. Selection of papers

This survey aims to collect and consider papers that deal with three regression testing techniques: test suite minimization, RTS and test case prioritization. Our intention is not to undertake a systematic review, but rather to provide a broad state-of-the-art view on these related fields. Many different approaches have been proposed to aid regression testing, which has resulted in a body of literature that is spread over a wide variety of domains and publication venues. The majority of surveyed literature has been published in the software engineering domain, and especially in the software testing and software maintenance literature. However, the regression testing literature also overlaps with those of programming language analysis, empirical software engineering and software metrics.

Therefore, the paper selection criteria on which this survey is based are the problems considered in papers, while focusing on the specific topics of minimization, selection and prioritization. The formal definitions of these problems are presented in Section 2.2. The selected papers are listed in the Appendix. Fast abstracts and short papers have been excluded.

2. BACKGROUND

This section introduces the basic concepts and definitions that form a nomenclature of regression testing and minimization, selection and prioritization techniques.

2.1. Regression testing

Regression testing is performed between two different versions of software in order to provide confidence that the newly introduced features of the SUT do not interfere with the existing features. While the exact details of the modifications made to SUT will often be available, they may not be easily available in some cases. For example, when the new version is written in a different programming language or when the source code is unavailable, modification data will also be unavailable.

The following notations are used to describe concepts in the context of regression testing. Let P be the current version of the program under test, and P' be the next version of P. Let S be the current set of specifications for P, and S' be the set of specifications for P'. T is the existing test suite. Individual test cases will be denoted by lower case: t. $P(t)$ stands for the execution of P using t as input.

2.2. Distinction between classes of techniques

It is necessary at this point to establish a clear terminological distinction between the different classes of techniques described in the paper. Test suite minimization techniques seek to reduce the size of a test suite by eliminating redundant test cases from the test suite. Minimization is sometimes also called 'test suite reduction', meaning that the elimination is permanent. However, these two concepts are essentially interchangeable because all reduction techniques can be used to produce a temporary subset of the test suite, whereas any minimization techniques can be used to permanently eliminate test cases. More formally, following Rothermel *et al.* [8], the test suite minimization is defined as follows:

Definition 1 (*Test Suite Minimization Problem*)
Given: A test suite, T, a set of test requirements $\{r_1, \ldots, r_n\}$, that must be satisfied to provide the desired 'adequate' testing of the program, and subsets of T, $T_1, \ldots, T_n$, one associated with each of the r_is such that any one of the test cases t_j belonging to T_i can be used to achieve requirement r_i.

Problem: Find a representative set, T', of test cases from T that satisfies all r_is.

The testing criterion is satisfied when every test requirement in $\{r_1, \ldots, r_n\}$ is satisfied. A test requirement, r_i, is satisfied by any test case, t_j, that belongs to the T_i, a subset of T. Therefore, the representative set of test cases is the hitting set of the T_is. Furthermore, in order to maximize the effect of minimization, T' should be the minimal hitting set of the T_is. The minimal hitting set problem is an NP-complete problem as is the dual problem of the minimal set cover problem [9].

DOI: 10.1002/stvr

While test case selection techniques also seek to reduce the size of a test suite, the majority of selection techniques are *modification-aware*. That is, the selection is not only temporary (i.e. specific to the current version of the program), but also focused on the identification of the modified parts of the program. Test cases are selected because they are relevant to the changed parts of the SUT, which typically involves a white-box static analysis of the program code. Throughout this survey, the meaning of 'test case selection problem' is restricted to this modification-aware problem. It is also often referred to as the RTS problem. More formally, following Rothermel and Harrold [1], the selection problem is defined as follows (refer to Section 4 for more details on how the subset T' is selected):

Definition 2 (*Test Case Selection Problem*)
Given: The program, P, the modified version of P, P' and a test suite, T.
Problem: Find a subset of T, T', with which to test P'.

Finally, test case prioritization concerns ordering test cases for early maximization of some desirable properties, such as the rate of fault detection. It seeks to find the optimal permutation of the sequence of test cases. It does not involve selection of test cases, and assumes that all the test cases may be executed in the order of the permutation it produces, but that testing may be terminated at some arbitrary point during the testing process. More formally, the prioritization problem is defined as follows:

Definition 3 (*Test Case Prioritization Problem*)
Given: A test suite, T, the set of permutations of T, PT, and a function from PT to real numbers, $f : PT \rightarrow \mathbb{R}$.
Problem: To find $T' \in PT$ such that $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$.

This survey focuses on papers that consider one of these three problems. Throughout the paper, these three techniques will be collectively referred to as 'regression testing techniques'.

2.3. Classification of test cases

Leung and White present the first systematic approach to regression testing by classifying types of regression testing and test cases [10]. Regression testing can be categorized into *progressive* regression testing and *corrective* regression testing. Progressive regression testing involves changes of specifications in P', meaning that P' should be tested against S'. On the other hand, corrective regression testing does not involve changes in specifications, but only in design decisions and actual instructions. It means that the existing test cases can be reused without changing their input/output relation.

Leung and White categorize test cases into five classes. The first three classes consist of test cases that already exist in T.

- *Reusable*: Reusable test cases only execute the parts of the program that remain unchanged between two versions, i.e. the parts of the program that are common to P and P'. It is unnecessary to execute these test cases in order to test P'; however, they are called *reusable* because they may still be retained and reused for the regression testing of the future versions of P.
- *Retestable*: Retestable test cases execute the parts of P that have been changed in P'. Thus, retestable test cases should be re-executed in order to test P'.
- *Obsolete*: Test cases can be rendered obsolete because (1) their input/output relation is no longer correct due to changes in specifications, (2) they no longer test what they were designed to test due to modifications to the program or (3) they are 'structural' test cases that no longer contribute to structural coverage of the program.

The remaining two classes consist of test cases that have yet to be generated for the regression testing of P'.

- *New-structural*: New-structural test cases test the modified program constructs, providing structural coverage of the modified parts in P'.

DOI: 10.1002/stvr

- *New-specification*: New-specification test cases test the modified program specifications, testing the new code generated from the modified parts of the specifications of P'.

Leung and White go on to propose a retest strategy, in which a *test plan* is constructed based on the identification of changes in the program and classification of test cases. Although the definition of a test plan remains informal, it provides a basis for the subsequent literature. It is of particular importance to regression test case selection techniques, since these techniques essentially concern the problem of identifying *retestable* test cases. Similarly, test suite minimization techniques concern the identification of *obsolete* test cases. Test case prioritization also can be thought of as a more sophisticated approach to the construction of a test plan.

It should be noted that the subsequent literature focusing on the idea of selecting and reusing test cases for regression testing is largely concerned with corrective regression testing only. For progressive regression testing, it is very likely that new test cases are required in order to test the new specifications. So far, this aspect of the overall regression testing picture has been a question mainly reserved for test data generation techniques. However, the early literature envisages a 'complete' regression testing strategy that should also utilize test data generation techniques.

3. TEST SUITE MINIMIZATION

Test suite minimization techniques aim to identify redundant test cases and to remove them from the test suite in order to reduce the size of the test suite. The minimization problem described by Definition 1 can be considered as the minimal hitting set problem.

Note that the minimal hitting set formulation of the test suite minimization problem depends on the assumption that each r_i can be satisfied by a single test case. In practice, this may not be true. For example, suppose that the test requirement is functional rather than structural and, therefore, requires more than one test case to be satisfied. The minimal hitting set formulation no longer applies. In order to apply the given formulation of the problem, the functional granularity of test cases needs to be adjusted accordingly. The adjustment process may be either that a higher level of abstraction would be required so that each test case requirement can be met with a single test scenario composed of relevant test cases, or that a 'large' functional requirement needs to be divided into smaller sub-requirements that will correspond to individual test cases.

3.1. Heuristics

The NP-completeness of the test suite minimization problem encourages the application of heuristics; previous work on test case minimization can be regarded as the development of different heuristics for the minimal hitting set problem [11–14].

Horgan and London applied linear programming to the test case minimization problem in their implementation of a data-flow-based testing tool, ATAC [13, 15]. Harrold *et al.* presented a heuristic for the minimal hitting set problem with the worst-case execution time of $O(|T| * \max(|T_i|))$ [12]. Here $|T|$ represents the size of the original test suite, and $\max(|T_i|)$ represents the cardinality of the largest group of test cases among $T_1, \ldots, T_n$.

Chen and Lau applied GE and GRE heuristics and compared the results to that of the Harrold–Gupta–Soffa (HGS) heuristic [11]. The GE and GRE heuristics can be thought of as variations of the greedy algorithm that is known to be an effective heuristic for the set cover problem [16]. Chen *et al.* defined *essential* test cases as the opposite of *redundant* test cases. If a test requirement r_i can be satisfied by one and only one test case, the test case is an essential test case. On the other hand, if a test case satisfies only a subset of the test requirements satisfied by another test case, it is a redundant test case. Based on these concepts, the GE and GRE heuristics can be summarized as follows:

- *GE heuristic*: first select all essential test cases in the test suite; for the remaining test requirements, use the additional greedy algorithm, i.e. select the test case that satisfies the maximum number of unsatisfied test requirements.

DOI: 10.1002/stvr

Table I. Example test suite taken from Tallam and Gupta [18]. The early selection made by the greedy approach, t_1, is rendered redundant by subsequent selections, $\{t_2, t_3, t_4\}$.

	Testing requirements					
Test case	r_1	r_2	r_3	r_4	r_5	r_6
t_1	x	x	x			
t_2	x			x		
t_3		x			x	
t_4			x			x
t_5					x	

- *GRE heuristic*: first remove all redundant test cases in the test suite, which may make some test cases essential; then perform the GE heuristic on the reduced test suite.

Their empirical comparison suggested that no single technique is better than the other. This is a natural finding, because the techniques concerned are heuristics rather than precise algorithms.

Offutt *et al.* also treated the test suite minimization problem as the dual of the minimal hitting set problem, i.e. the set cover problem [14]. Their heuristics can also be thought of as variations of the greedy approach to the set cover problem. However, they adopted several different test case orderings, instead of the fixed ordering in the greedy approach. Their empirical study applied their techniques to a mutation-score-based test case minimization, which reduced sizes of test suites by over 30%.

Whereas other minimization approaches primarily considered code-level structural coverage, Marré and Bertolino formulated test suite minimization as a problem of finding a spanning set over a graph [17]. They represented the structure of the SUT using a *decision-to-decision* graph (ddgraph). A ddgraph is a more compact form of the normal CFG since it omits any node that has one entering edge and one exiting edge, making it an ideal representation of the SUT for branch coverage. They also mapped the result of data-flow analysis onto the ddgraph for testing requirements such as *def–use* coverage. Once testing requirements are mapped to entities in the ddgraph, the test suite minimization problem can be reduced to the problem of finding the minimal spanning set.

Tallam and Gupta developed the greedy approach further by introducing the *delayed greedy* approach, which is based on the Formal Concept Analysis of the relation between test cases and testing requirements [18]. A potential weakness of the greedy approach is that the early selection made by the greedy algorithm can eventually be rendered redundant by the test cases subsequently selected. For example, consider the test suite and testing requirements depicted in Table I, taken from Tallam and Gupta [18]. The greedy approach will select t_1 first as it satisfies the maximum number of testing requirements, and then continues to select t_2, t_3 and t_4. However, after the selection of t_2, t_3 and t_4, t_1 is rendered redundant. Tallam *et al.* tried to overcome this weakness by constructing a concept lattice, a hierarchical clustering based on the relation between test cases and testing requirements. Tallam *et al.* performed two types of reduction on the concept lattice. First, if a set of requirements covered by t_i is a superset of the set of requirements covered by t_j, then t_j is removed from the test suite. Second, if a set of test cases that cover requirement r_i is a subset of the set of test cases that cover requirement r_j, requirement r_i is removed. The concept lattice is a natural representation that supports the identification of these test cases. Finally, the greedy algorithm is applied to the transformed set of test cases and testing requirements. In the empirical evaluation, the test suites minimized by this 'delayed greedy' approach were either the same size or smaller than those minimized by the classical greedy approach or by the HGS heuristic.

Jeffrey and Gupta extended the HGS heuristic so that certain test cases are selectively retained [19, 20]. This 'selective redundancy' is obtained by introducing a secondary set of testing requirements. When a test case is marked as redundant with respect to the first set of testing requirements, Jeffrey and Gupta considered whether the test case is also redundant with respect to the second set of testing requirements. If it is not, the test case is still selected, resulting in a certain level of redundancy with respect to the first set of testing requirements. The empirical evaluation used

DOI: 10.1002/stvr

branch coverage as the first set of testing requirements and *all-uses* coverage information obtained by data-flow analysis. The results were compared to two versions of the HGS heuristic, based on branch coverage and *def–use* coverage. The results showed that, while their technique produced larger test suites, the fault-detection capability was better preserved compared to single-criterion versions of the HGS heuristic.

Whereas the selective redundancy approach considers the secondary criterion only when a test case is marked as being redundant by the first criterion, Black *et al.* considered a bi-criteria approach that takes into account both testing criteria [21]. They combined the *def–use* coverage criterion with the past fault-detection history of each test case using a weighted-sum approach and used Integer Linear Programming (ILP) optimization to find subsets. The weighted-sum approach uses weighting factors to combine multiple objectives. For example, given a weighting factor α and two objectives o_1 and o_2, the new and combined objective, o', is defined as follows:

$$o' = \alpha o_1 + (1 - \alpha) o_2$$

Consideration of a secondary objective using the weighted-sum approach has been used in other minimization approaches [22] and prioritization approaches [23]. Hsu and Orso also considered the use of an ILP solver with multi-criteria test suite minimization [22]. They extended the work of Black *et al.* by comparing different heuristics for a multi-criteria ILP formulation: the weighted-sum approach, the prioritized optimization and a hybrid approach. In prioritized optimization, the human user assigns a priority to each of the given criteria. After optimizing for the first criterion, the result is added as a constraint, while optimizing for the second criterion, and so on. However, one possible weakness shared by these approaches is that they require additional input from the user of the technique in the forms of weighting coefficients or priority assignment, which might be biased, unavailable or costly to provide.

Contrary to these approaches, Yoo and Harman treated the problem of time-aware prioritization as a multi-objective optimization problem [24]. Instead of using a fitness function that combines selection and prioritization, they used a Pareto-efficient multi-objective evolutionary algorithm to simultaneously optimize for multiple objectives. You and Harman argued that the resulting Pareto-frontier not only provides solutions but also allows the tester to observe trade-offs between objectives, providing additional insights.

McMaster and Memon proposed a test suite minimization technique based on *call-stack coverage*. A test suite is represented by a set of unique maximum depth call stacks; its minimized test suite is a subset of the original test suite whose execution generates the same set of unique maximum depth call stacks. Note that their approach is different from simply using function coverage for test suite minimization. Consider two test cases, t_1 and t_2, respectively, producing call stacks $c_1 = \langle f_1, f_2, f_3 \rangle$ and $c_2 = \langle f_1, f_2 \rangle$. With respect to function coverage, t_2 is rendered redundant by t_1. However, McMaster and Memon regard c_2 to be unique from c_1. For example, it may be that t_2 detects a failure that prevents the invocation of function f_3. Once the call-stack coverage information is collected, the HGS heuristic can be applied. McMaster and Memon later applied the same approach to Graphical User Interface (GUI) testing [25]. It was also implemented for object-oriented systems by Smith *et al.* [26].

While most test suite minimization techniques are based on some kind of coverage criteria, there do exist interesting exceptions. Harder *et al.* approached test suite minimization using operational abstraction [27]. An operational abstraction is a formal mathematical description of program behaviour. While it is identical to formal specifications in form, an operational abstraction expresses dynamically observed behaviour. Harder *et al.* use the widely studied Daikon dynamic invariant detector [28] to obtain operational abstractions. Daikon requires executions of test cases for the detection of possible program invariants. Test suite minimization is proposed as follows: if the removal of a test case does not change the detected program invariant, it is rendered redundant. They compared the operational abstraction approach to branch coverage-based minimization. While their approach resulted in larger test suites, it also maintained higher fault-detection capability. Moreover, Harder *et al.* also showed that test suites minimized for coverage adequacy can often be improved by considering operational abstraction as an additional minimization criterion.

DOI: 10.1002/stvr

Leitner *et al.* propose a somewhat different version of the minimization problem [29]. They start from the assumption that they already have a *failing test case*, which is too complex and too long for the human tester to understand. Note that this is often the case with randomly generated test data; the test case is often simply too complex for the human to establish the cause of failure. The goal of minimization is to produce a shorter version of the test case; the testing requirement is that the shorter test case should still reproduce the failure. This minimization problem is interesting because there is no uncertainty about the fault-detection capability; it is given as a testing requirement. Leitner *et al.* applied the widely studied *delta-debugging* technique [30] to reduce the size of the failing test case.

Schroeder and Korel proposed an approach of test suite minimization for black-box software testing [31]. They noted that the traditional approach of testing black-box software with combinatorial test suites may result in redundancy, since certain inputs to the software may not affect the outcome of the output being tested. They first identified, for each output variable, the set of input variables that can affect the outcome. Then, for each output variable, an individual combinatorial test suite is generated with respect to only those input variables that may affect the outcome. The overall test suite is a union of all combinatorial test suites for individual output variables. This has a strong connection to the concept of Interaction Testing, which is discussed in detail in Section 5.2.

Other work has focused on model-based test suite minimization [32, 33]. Vaysburg *et al.* introduced a minimization technique for model-based test suites that uses dependence analysis of Extended Finite State Machines (EFSMs) [32]. Each test case for the model is a sequence of transitions. Through dependence analysis of the transition being tested, it is possible to identify the transitions that affect the tested transition. In other words, testing a transition T can be thought of as testing a set of other transitions that affect T. If a particular test case tests the same set of transitions as some other test case, then it is considered to be redundant. Korel *et al.* extended this approach by combining the technique with automatic identification of changes in the model [33]. The dependence analysis-based minimization technique was applied to the set of test cases that were identified to execute the modified transitions. Chen *et al.* extended Korel's model-based approach to incorporate more complex representations of model changes [34].

A risk shared by most test suite minimization techniques is that a *discarded* test case may detect a fault. In some domains, however, test suite minimization techniques can enjoy the certainty of guaranteeing that discarding a test case will not reduce the fault-detection capability. Anido *et al.* investigated test suite minimization for testing Finite State Machines (FSMs) in this context [35]. When only some components of the SUT need testing, the system can be represented as a composition of two FSMs: *component* and *context*. The context is assumed to be fault-free. Therefore, certain transitions of the system that concern only the context can be identified to be redundant. Under the 'testing in context' assumption (i.e. the context is fault-free), it follows that it is possible to guarantee that a discarded test case cannot detect faults.

Kaminski and Ammann investigated the use of a logic criterion to reduce test suites while guaranteeing fault detection in testing predicates over Boolean variables [36]. From the formal description of fault classes, it is possible to derive a hierarchy of fault classes [37]. From the hierarchy, it follows that the ability to detect a class of faults may guarantee the detection of another class. Therefore, the size of a test suite can be reduced by executing only those test cases for the class of faults that subsume another class, whenever this is feasible.

3.2. Impact on fault-detection capability

Although the techniques discussed so far reported reduced test suites, there has been a persistent concern about the effect that the test suite minimization has on the fault-detection capability of test suites. Several empirical studies were conducted to investigate this effect [8, 38–40].

Wong *et al*. studied 10 common Unix programs using randomly generated test suites; this empirical study is often referred to as the WHLM study [39]. To reduce the size of the test suites, they used the ATAC testing tool developed by Horgan and London [13, 15]. First, a large pool of test cases was created using a random test data generation method. From this pool, several test suites with different total block coverage were generated. After generating test suites randomly,

DOI: 10.1002/stvr

artificial faults were seeded into the programs. These artificial faults were then categorized into four groups. Faults in Quartile-I can be detected by [0–25)% of the test cases from the original test suite; the percentage for Quartile-II, III and IV is [25–50)%, [50–75)% and [75–100]%, respectively. Intuitively, faults in Quartile-I are harder to detect than those in Quartile-IV. The effectiveness of the minimization itself was calculated as follows:

$$\left(1-\frac{\text{number of test cases in the reduced test suite}}{\text{number of test cases in the original test suite}}\right)*100\%$$

The impact of test suite minimization was measured by calculating the reduction in fault detection effectiveness as follows:

$$\left(1-\frac{\text{number of faults detected by the reduced test suite}}{\text{number of faults detected by the original test suite}}\right)*100\%$$

By categorizing the test suites (by different levels of block coverage) and test cases (by difficulty of detection), they arrived at the following observation. First, the reduction in size is greater in test suites with a higher block coverage in most cases. This is natural considering that test suites with higher block coverage will require more test cases in general. The average size reduction for test suites with (50–55)%, (60–65)%,(70–75)%,(80–85)% and (90–95)% block coverage was 1.19, 4.46, 7.78, 17.44, 44.23%, respectively. Second, the fault-detection effectiveness was decreased by test case reduction, but the overall decrease in fault-detection effectiveness is not excessive and could be regarded as worthwhile for the reduced cost. The average effectiveness reduction for test suites with (50–55)%, (60–65)%,(70–75)%,(80–85)% and (90–95)% block coverage was 0, 0.03, 0.01, 0.38, 1.45%, respectively. Third, test suite minimization did not decrease the fault-detection effectiveness for faults in Quartile-IV at all, meaning that all faults in Quartile-IV had been detected by the reduced test suite. The average decrease in fault-detection effectiveness for Quartile-I, II and III was 0.39, 0.66, and 0.098%, respectively. The WHLM study concluded that, if the cost of testing is directly related to the number of test cases, then the use of the reduction technique is recommended.

Wong *et al*. followed up on the WHLM study by applying the ATAC tool to test suites of another, bigger C program; this empirical study is often referred to as the WHLP study [40]. The studied program, `space`, was an interpreter for the Array Description Language (ADL) developed by the European Space Agency. In the WHLP study, test cases were generated, not randomly, but from the operational profiles of `space`. That is, each test case in the test case pool was generated so that it matches an example of real usage of `space` recorded in an operational profile. From the test case pool, different types of test suites were generated. The first group of test suites was constructed by randomly choosing a fixed number of test cases from the test case pool. The second group of test suites was constructed by choosing test cases from the test case pool until a predetermined block coverage target was met. The faults in the program were not artificial, but real faults that were retrieved from development logs.

The results of the WHLP study confirmed the findings of the WHLM study. As in the WHLM study, test suites with low initial block coverage (50, 55, 60 and 65%) showed no decrease in fault-detection effectiveness, after test suite minimization. For both the fixed size test suites and fixed coverage test suites, the application of the test case reduction technique did not affect the fault-detection effectiveness in any significant way; the average effectiveness reduction due to test suite minimization was less than 7.28%.

While both the WHLM and WHLP studies showed that the impact of test suite minimization on fault-detection capability was insignificant, other empirical studies produced radically different findings. Rothermel *et al.* also studied the impact of test suite minimization on the fault-detection capability [38]. They applied the HGS heuristics to the Siemens suite [41], and later expanded this to include `space` [8]. The results from these empirical studies contradicted the previous findings of the WHLM and WHMP studies.

For the study of the Siemens suite [38], Rothermel *et al.* constructed test suites from the test case pool provided by the Siemens suite so that the test suites include varying amounts of redundant

DOI: 10.1002/stvr

test cases that do not contribute to the decision coverage of the test suite. The effectiveness and impact of reduction was measured using the same metrics that were used in the WHLM study.

Rothermel *et al.* reported that the application of the test suite minimization technique produced significant savings in test suite size. The observed tendency in size reduction suggested a logarithmic relation between the original test suite size and the reduction effectiveness. The results of logarithmic regression confirmed this.

However, Rothermel *et al.* also reported that, due to the size reduction, the fault-detection capabilities of test suites were severely compromised. The reduction in fault-detection effectiveness was over 50% for more than half of 1000 test suites considered, with some cases reaching 100%. Rothermel *et al.* also reported that, unlike the size reduction effectiveness, the fault-detection effectiveness did not show any particular correlation with the original test suite size.

This initial empirical study was subsequently extended [8]. For the Siemens suite, the results of the HGS heuristic were compared to random reduction by measuring the fault-detection effectiveness of randomly reduced test suites. Random reduction was performed by randomly selecting, from the original test suite, the same number of test cases as in the reduced version of the test suite. The results showed that random reduction produced larger decreases in fault-detection effectiveness. To summarize the results for the Siemens suite, the test suite minimization technique produced savings in test suite size, but at the cost of decreased fault-detection effectiveness; however, the reduction heuristic showed better fault-detection effectiveness than the random reduction technique.

Rothermel *et al.* also expanded the previous empirical study by including the larger program, `space` [8]. The reduction in size observed in the test suites of `space` confirmed the findings of the previous empirical study of the Siemens suite; the size reduction effectiveness formed a logarithmic trend, plotted against the original test suite size, similar to the programs in the Siemens suite. More importantly, the reduction in fault-detection effectiveness was less than those of the Siemens suite programs. The average reduction in fault-detection effectiveness of test suites reduced by the HGS heuristic was 8.9%, whereas. that of test suites reduced by random reduction was 18.1%.

Although the average reduction in fault-detection effectiveness is not far from that reported for the WHLP study in the case of `space`, those of the Siemens suite differed significantly from both the WHLP study and the WHLM study, which reported that the application of the minimization technique did not have significant impact on fault-detection effectiveness. Rothermel *et al.* [8] pointed out the following differences between these empirical studies as candidates for the cause(s) of the contradictory findings, which is paraphrased as follows:

1. *Different subject programs*: the programs in the Siemens suite are generally larger than those studied in both the WHLM and WHLP studies. Difference in program size and structure certainly could have impact on the fault-detection effectiveness.
2. *Different types of test suites*: the WHLM study used test suites that were not coverage-adequate and were much smaller compared to test suites used by Rothermel *et al.* The initial test pools used in the WHLM study also did not necessarily contain any minimum number of test cases per covered item. These differences could have contributed to less redundancy in test suites, which led to reduced likelihood that test suite minimization will exclude fault-revealing test cases.
3. *Different types of test cases*: the test suites used in the WHLM study contained a few test cases that detected all or most of the faults. When such strong test cases are present, reduced versions of the test suites may well show little loss in fault-detection effectiveness.
4 *Different types of faults*: the faults studied by Rothermel *et al.* were all Quartile-I faults according to the definition of the WHLM study, whereas only 41% of the faults studied in the WHLM study belonged to the Quartile-I group. By having more 'easy-to-detect' faults, the test suites used in the WHLM study could have shown less reduction in fault-detection effectiveness after test suite minimization.

Considering the two contradicting empirical results, it is natural to conclude that the question of evaluating the effectiveness of the test suite minimization technique is very hard to answer in general and for all testing scenarios. The answer depends on too many factors, such as the structure

DOI: 10.1002/stvr

of the SUT, the quality of test cases and test suites and the types of faults present. This proliferation of potential contributory factors makes it very difficult to generalize any empirical result.

The empirical studies from WHLM, WHLP and Rothermel *et al.* all evaluated the effectiveness of test suite minimization in terms of two metrics: percentage size reduction and percentage fault-detection reduction. McMaster and Memon noticed that neither metric considers the actual role each testing requirement plays on fault detection [42]. Given a set of test cases, TC, a set of known faults, KF and a set of testing requirements, CR, *fault correlation* for a testing requirement $i \in CR$ to fault $k \in KF$ is defined as follows:

$$\frac{|\{j \in TC | j \text{ covers } i\} \cap \{j \in TC | j \text{ detects } k\}|}{|\{j \in TC | j \text{ covers } i\}|}$$

The expected probability of finding a given fault k after test suite minimization is defined as the maximum fault correlation of all testing requirements with k. From this, the probability of detecting all known faults in KF is the product of expected probability of finding all $k \in KF$. Since CR is defined by the choice of minimization criterion, e.g. branch coverage or call-stack coverage, comparing the probability of detecting all known faults provides a systematic method of comparing different minimization criteria, without depending on a specific heuristic for minimization. The empirical evaluation of McMaster and Memon compared five different minimization criteria for the minimization of test suites for GUI-intensive applications: event coverage (i.e. each event is considered as a testing requirement), event interaction coverage (i.e. each pair of events is considered as a testing requirement), function coverage, statement coverage and call-stack coverage proposed in [43]. While call-stack coverage achieved the highest average probability of detecting all known faults, McMaster and Memon also found that different faults correlate more highly with different criteria. This analysis provides valuable insights into the selection of minimization criterion.

Yu *et al.* considered the effect of test suite minimization on fault localization [44]. They applied various test suite minimization techniques to a set of programs, and measured the impact of the size reduction on the effectiveness of coverage-based fault localization techniques. Yu *et al.* reported that higher reduction in test suite size, typically achieved by statement coverage-based minimization, tends to have a negative impact on fault localization, whereas minimization techniques that maintain higher levels of redundancy in test suites have negligible impact.

4. TEST CASE SELECTION

Test case selection, or the RTS problem, is essentially similar to the test suite minimization problem; both problems are about choosing a subset of test cases from the test suite. The key difference between these two approaches in the literature is whether the focus is upon the changes in the SUT. Test suite minimization is often based on metrics such as coverage measured from a single version of the program under test. By contrast, in RTS, test cases are selected because their execution is relevant to the changes between the previous and the current version of the SUT.

To recall Definition 2, T' ideally should contain all the *faults-revealing* test cases in T, i.e. the test cases that will reveal faults in P'. In order to define this formally, Rothermel and Harrold introduced the concept of a *modification-revealing* test case [45]. A test case t is modification-revealing for P and P' if and only if $P(t) \neq P'(t)$. Given the following two assumptions, it is possible to identify the fault-revealing test cases for P' by finding the modification-revealing test cases for P and P'.

- **P-Correct-for-T Assumption**: It is assumed that, for each test case $t \in T$, when P was tested with t, P halted and produced the correct output.
- **Obsolete-Test-Identification Assumption**: It is assumed that there exists an effective procedure for determining, for each test case $t \in T$, whether t is obsolete for P'.

From these assumptions, it is clear that every test case in T terminates and produces correct output for P, and is also supposed to produce the same output for P'. Therefore, a modification-revealing

DOI: 10.1002/stvr

test case t must be also fault-revealing. Unfortunately, it is not possible to determine whether a test case t is fault-revealing against P' or not because it is undecidable whether P' will halt with t. Rothermel considers a weaker criterion for the selection, which is to select all *modification-traversing* test cases in T. A test case t is modification-traversing for P and P' if and only if:

1. it executes new or modified code in P' or
2. it formerly executed code that had been deleted in P'.

Rothermel also introduced the third assumption, which is the Controlled-Regression-Testing assumption.

- *Controlled-Regression-Testing Assumption*: When P' is tested with t, all factors that might influence the output of P', except for the code in P', are kept constant with respect to their states when P was tested with t.

Given that the Controlled-Regression-Testing assumption holds, a non-obsolete test case t can thereby be modification-revealing only if it is also modification-traversing for P and P'. Now, if the P-Correct-for-T assumption and the Obsolete-Test-Identification assumption hold along with the Controlled-Regression-Testing assumption, then the following relation also holds between the subset of fault-revealing test cases, T_{fr}, the subset of modification-revealing test cases, T_{mr}, the subset of modification-traversing test cases, T_{mt}, and the original test suite, T:

$$T_{fr} = T_{mr} \subseteq T_{mt} \subseteq T$$

Rothermel and Harrold admitted that the Controlled-Regression-Testing assumption may not be always practical, since certain types of regression testing may make it impossible to control the testing environment, e.g. testing of a system ported to different operating system [1]. Other factors like non-determinism in programs and time dependencies are also difficult to control effectively. However, finding the subset of modification-traversing test cases may still be a useful approach in practice, because T_{mt} is the closest approximation to T_{mr} that can be achieved without executing all test cases. In other words, by finding T_{mt}, it is possible to exclude those test cases that are guaranteed not to reveal any fault in P'. The widely used term, *safe* RTS, is based on this concept [46]. A safe RTS technique is not safe from all possible faults; however, it is safe in a sense that, if there exists a modification-traversing test case in the test suite, it will definitely be selected.

Based on Rothermel's formulation of the problem, it can be said that test case selection techniques for regression testing focus on identifying the modification-traversing test cases in the given test suite. The details of the selection procedure differ according to how a specific technique defines, seeks and identifies modifications in the program under test. Various approaches have been proposed using different techniques and criteria including Integer Programming [47, 48], data-flow analysis [49–52], symbolic execution [53], dynamic slicing [54], CFG graph-walking [46, 55–57], textual difference in source code, [58, 59] SDG slicing [60], path analysis [61], modification detection [62], firewall [63–66], CFG cluster identification [67] and design-based testing [68, 69]. The following subsections describe these in more detail, highlighting their strengths and weaknesses.

4.1. Integer programming approach

One of the earliest approaches to test case selection was presented by Fischer *et al.*, who used Integer Programming (IP) to represent the selection problem for testing FORTRAN programs [47, 48]. Lee and He implemented a similar technique [70]. Fischer first defined a program segment as a single-entry, single-exit block of code whose statements are executed sequentially. Their selection algorithm relies on two matrices that describe the relation between program segments and test cases, as well as between different program segments.

For a program with m segments and n test cases, the IP formulation is given as the problem of finding the decision vector,$<x_1, \ldots, x_n>$, that minimizes the following objective function, Z:

$$Z = c_1 x_1 + c_2 x_2 + \cdots + c_n x_n$$

DOI: 10.1002/stvr

subject to:

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \geq b_1$$
$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \geq b_2$$
$$\vdots$$
$$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \geq b_m$$

The decision vector, $<x_1, \ldots, x_n>$, represents the subset of selected test cases; x_i is equal to 1 if the ith test case is included; 0 otherwise. The coefficients, $c1, \ldots, c_n$, represent the cost of executing each corresponding test case; Fischer *et al.* used the constant value of 1 for all coefficients, treating all test cases as being equally expensive. The test case dependency matrix, $a_{11}, \ldots, a_{mn}$ represents the relations between test cases and the program segments. The element a_{ij} is equal to 1 if the ith program segment is executed by the test case j; 0 otherwise.

After deriving the series of inequalities, the set of b_k values are determined by using a reachability matrix that describes the program segments that are reachable from other segments. Using this approach, when the modified segments are known, it is possible to calculate all the segments that are reachable from the modified segments, which thus need to be tested at least once. The integer programming formulation is completed by assigning 1 to the b values for all the segments that need to be tested. The inequality, $a_{i1}x_1 + \cdots + a_{in}x_n \geq b_i$, thus ensures that at least one included test case covers the program element reachable from a change.

Hartmann and Robson implemented and extended a version of Fischer's algorithm in order to apply the technique to C [71–73]. They treat subroutines as segments, achieving subroutine coverage rather than statement coverage.

One weakness in Fischer's approach is its inability to deal with control-flow changes in P'. The test case dependency matrix, $a_{11}, \ldots, a_{mn}$, depends on the control-flow structure of the program under test. If the control-flow structure changes, the test case dependency matrix can be updated only by executing all the test cases, which negates the point of applying the selection technique.

4.2. Data-flow analysis approach

Several test case selection techniques have been proposed based on data-flow analysis [49–52]. Data-flow analysis-based selection techniques seek to identify new, modified or deleted definition–use pairs in P', and select test cases that exercise these pairs.

Harrold and Soffa presented data-flow analysis as the testing criterion for an incremental approach to unit testing during the maintenance phase [50]. Taha *et al.* built upon this idea and presented a test case selection framework based on an incremental data-flow analysis algorithm [52]. Harrold and Soffa developed both intra-procedural and inter-procedural selection techniques [51, 74]. Gupta *et al.* applied program slicing techniques to identify definition–use pairs that are affected by a code modification [49]. The use of slicing techniques enabled identification of definition–use pairs that need to be tested without performing a complete data-flow analysis, which is often very costly. Wong *et al.* combined a data-flow selection approach with a coverage-based minimization and prioritization to further reduce the effort [75].

One weakness shared by all data-flow analysis-based test case selection techniques is the fact that they are unable to detect modifications that are unrelated to data-flow change. For example, if P' contains new procedure calls without any parameter, or modified output statements that contain no variable uses, data-flow techniques may not select test cases that execute these.

Fisher II *et al.* applied the Data-flow-based RTS approach for test re-use in spreadsheet programs [76]. Fisher II *et al.* proposed an approach called What-You-See-Is-What-You-Test (WYSIWYT) to provide incremental, responsive and visual feedback about the *testedness* of cells in spreadsheets. The WYSIWYT framework collects and updates data-flow information incrementally as the user of the spreadsheet makes modifications to cells, using Cell Relation Graph.

DOI: 10.1002/stvr

Interestingly, the data-flow analysis approach to re-test spreadsheets is largely free from the difficulties associated with data-flow testing of procedural code, because spreadsheet programs lack tricky semantic features such as aliasing and unstructured control flow. This makes spreadsheet programs better suited to a data-flow analysis approach.

4.3. Symbolic execution approach

Yau and Kishmoto presented a test case selection technique based on symbolic execution of the SUT [53]. In symbolic execution of a program, the variables' values are treated as symbols, rather than concrete values [77]. Yau and Kishimoto's approach can be thought of as an application of symbolic execution and input partitioning to the test case selection problem. First, the technique statically analyses the code and specifications to determine the input partitions. Next, it produces test cases so that each input partition can be executed at least once. Given information on where the code has been modified, the technique then identifies the edges in the control-flow graph (CFG) that lead to the modified code. While symbolically executing all test cases, the technique determines test cases that traverse edges that do not reach any modification. The technique then selects all test cases that reach new or modified code. For the symbolic test cases that reach modifications, the technique completes the execution; the real test cases that match these symbolic test cases should be retested.

While it is theoretically powerful, the most important drawback of the symbolic execution approach is the algorithmic complexity of the symbolic execution. Yau and Kishmoto acknowledge that symbolic execution can be very expensive. Pointer arithmetic can also present challenging problems for symbolic execution-based approaches.

4.4. Dynamic slicing-based approach

Agrawal *et al.* introduced a family of test case selection techniques based on different program slicing techniques [54]. An *execution slice* of a program with respect to a test case is what is usually referred to as an execution trace; it is the set of statements executed by the given test case. A *dynamic slice* of a program with respect to a test case is the set of statements in the execution slice that have an influence on an output statement. Since an execution slice may contain statements that do not affect the program output, a dynamic slice is a subset of an execution slice. For example, consider the program shown in Figure 1. It contains two faults in line `S3` and `S11`, respectively. The execution slice of the program with respect to test case T_3 in Table II is shown in column `ES` of Figure 1. The dynamic slice of the program with respect to test case T_1 in Table II is shown in column `DS` of Figure 1.

In order to make the selection more precise, Agrawal *et al.* proposed two additional slicing criteria: a *relevant slice* and an *approximate relevant slice*. A relevant slice of a program with respect to a test case is the dynamic slice with respect to the same test case together with all the predicate statements in the program that, if evaluated differently, could have caused the program to produce a different output. An approximated relevant slice is a more conservative approach to include predicates that could have caused a different output; it is the dynamic slice with all the predicate statements in the execution slice. By including all the predicates in the execution slice, an approximated relevant slice caters for the indirect references via pointers. For example, consider the correction of `S3` in the program shown in Figure 1. The dynamic slice of T_4 does not include `S3` because the `class` value of T_4 is not affected by any of the lines between `S3` and `S8`. However, the relevant slice of T_4, shown in column `DS` of Figure 1, does include `S3` because it could have affected the output when evaluated differently.

The test suite and the previous version of the program under test are preprocessed using these slicing criteria; each test case is connected to a slice, *sl*, constructed by one of the four slicing criteria. After the program is modified, test cases for which *sl* contains the modified statement should be executed again. For example, assume that the fault in `S11`, detected by T_5, is corrected. The program should be retested with T_5. However, T_3 need not be executed because the execution slice of T_3, shown in column `ES` of Figure 1, does not contain `S11`. Similarly, assume that the fault in `S3`, detected by T_6, is corrected. The program should be retested with T_6. The execution

DOI: 10.1002/stvr

```
Slices
ES DS RS
x  x  x    S1: read(a,b,c);
x     x    S2: class := scalene;
x     x    S3: if a = b or b = a
             S4:      class := isosceles;
x     x    S5: if a * a = b * b + c * c
x          S6:      class := right
x  x  x    S7: if a = b and b = c
   x       S8:      class := equilateral
x  x  x    S9: case class of:
x         S10:      right        : area = b * c / 2;
   x      S11:      equilateral  : area = a * 2 * sqrt(3) / 4;
      x   S12:      otherwise    : s := (a + b + c) / 2;
      x   S13:                     area := sqrt(s * (s-a) * (s-b) * (s-c));
x         S14: end;
x         S15: write(class, area);
```

Figure 1. Example triangle classification program taken from Agrawal *et al.* [54]. Note that it is assumed that the input vector is sorted in descending order. It contains two faults. In `S3`, `b = a` should be `b = c`. In `S11`, `a * 2` should be `a * a`. The column `ES` represents the statements that belong to the execution slice with respect to test case T_3 in Table II. Similarly, column `DS` represents the dynamic slice with respect to test case T_1 in Table II and column `RS` the relevant slice with respect to test case T_4 in Table II.

Table II. Test cases used with the program shown in Figure 1, taken from Agrawal *et al.* [54]. Note that T_5 detects the fault in `S11`, because the value for area should be 3.90. Similarly, T_6 detects the fault in `S3`, because the value for class should be isosceles.

	Input			Output	
Test case	a	b	c	Class	Area
T_1	2	2	2	equilateral	1.73
T_2	4	4	3	isosceles	5.56
T_3	5	4	3	right	6.00
T_4	6	5	4	scalene	9.92
T_5	3	3	3	equilateral	2.60
T_6	4	3	3	scalene	4.47

slice technique selects all six test cases, T_1–T_6, after the correction of the fault in `S3` because the execution slices of all six test cases include `S3`. However, it is clear that T_1 and T_3 are not affected by the correction of `S3`; their `class` values are overwritten after the execution of `S3`. The dynamic slicing technique overcomes this weakness. The dynamic slice of T_1 is shown in column `DS` of Figure 1. Since `S3` does not affect the output of T_1, it is not included in the dynamic slice. Therefore, the modification of `S3` does not necessitate the execution of T_1.

Agrawal *et al.* first built their technique on cases in which modifications are restricted to those that do not alter the CFG of the program under test. As long as the CFG of the program remains the same, their technique is safe and can be regarded as an improvement over Fischer's integer programming approach. Slicing removes the need to formulate the linear programming problem, reducing the effort required from the tester. Agrawal *et al.* later relaxed the assumption about static CFGs in order to cater for modifications in the CFGs of the SUT. If a statement s is added to P, now the slice sl contains all the statements in P that uses the variables defined in s. Similarly, if a predicate p is added to P, the slice sl contains all the statements in P that are control-dependent on p. This does cater for the changes in the CFG to some degree, but it is not complete. For example, if the added statement is a simple output statement that does not define or use any variable, then this statement can still be modification-revealing. However, since the new statement does not contain any variable, its addition will not affect any of the existing slices, resulting in an empty selection.

DOI: 10.1002/stvr

4.5. Graph-walk approach

Rothermel and Harrold presented regression test case selection techniques based on graph-walking of Control Dependence Graphs (CDGs), Program Dependence Graphs (PDGs), System Dependence Graphs (SDGs) and CFGs [46, 56, 57, 78]. The CDG is similar to PDG but lacks data dependence relations. By performing a depth-first traversal of the CDGs of both P and P', it is possible to identify points in a program through which the execution trace reaches the modifications [46]. If a node in the CDG of P is not lexically equivalent to the corresponding node in the CDG of P', the algorithm selects all the test cases that execute the control-dependence predecessors of the mismatching node. The CDG-based selection technique does not cater for inter-procedural regression test case selection; Rothermel and Harrold recommend application of the technique at the individual procedural level.

Rothermel and Harrold later extended the graph-walking approach to use PDGs for intra-procedural selection, and SDGs for inter-procedural selection [56]. A weakness of the CDG-based technique is that, due to the lack of data dependence, the technique will select test cases that execute modified definitions but not the actual uses of a variable. If the modified definition of a variable is never used, it cannot contribute to any different output, and therefore its inclusion is not necessary for safe regression testing. PDGs contain data dependence for a single procedure; SDGs extend this to a complete program with multiple procedures. By using these graphs, Rothermel and Harrold's algorithm is able to check whether a modified definition of a variable is actually used later.

Rothermel and Harrold later presented the graph-walking approach based on CFGs [57]. The CFG-based technique essentially follows the approach introduced for the CDG-based technique, but on CFGs rather than on CDGs. Since CFG is a much simpler representation of the structure of a program, the CFG-based technique may be more efficient. However, the CFG lacks data dependence information, so the CFG-based technique may select test cases that are not capable of producing different outputs from the original programs, as explained above. The technique has been evaluated against various combinations of subject programs and test suites [79]. Ball improved the precision of the graph-walk approach with respect to branch coverage [80].

Rothermel *et al.* extended the CFG-based graph-walk approach for object-oriented software using the Inter-procedural Control-Flow Graph (ICFG) [81]. The ICFG connects methods using *call* and *return* edges. Harrold *et al.* adopted a similar approach for test case selection for Java software, using the Java Interclass Graph as representation (JIG) [82]. Xu and Rountev later extended this technique to consider AspectJ programs by incorporating the interactions between methods and advices at certain join points into the CFG [83]. Zhao *et al.* also considered a graph representation of AspectJ programs to apply a graph-walk approach for RTS [84]. Beydeda and Gruhn extended the graph-walk approach by adding black-box data-flow information to the Class Control-Flow Graph (CCFG) to test object-oriented software [85].

Orso *et al.* considered using different types of graph representation of the system to improve the scalability of graph-walk approach [86]. Their approach initially relies on a high-level graph representation of SUT to identify the parts of the system to be further analysed. Subsequently, the technique uses more detailed graph representation to perform more precise selection.

One strength of the graph-walk approach is its generic applicability. For example, it has been successfully used in black-box testing of re-usable classes [87]. Martins and Vieira captured the behaviours of a re-usable class by constructing a directed graph called the Behavioural Control-Flow Graph (BCFG) from the Activity Diagram (AD) of the class. The BCFG is a directed graph, $G = (V, E, s, x)$, with vertices V, edges E, a unique entry vertex s and an exit vertex x. Each vertex contains a label that specifies the signature of a method; each edge is also labelled according to the corresponding guards in AD. A path in G from s to x represents a possible life history of an object. By mapping changes made to the object to its BCFG and applying the graph-walking algorithm, it is possible to select test cases based on the behavioural difference between two versions of the same object. This approach requires traceability between the behavioural model and the actual test cases, because test cases are selected, not based on their structural coverage, but based on their behavioural coverage measured on BCFG. ADs have also been directly used for RTS by Chen *et al.* [88].

DOI: 10.1002/stvr

Orso *et al.* used a variation of the graph-walk approach to consider an RTS technique based on meta-data and specifications obtained from software components [89, 90]. They presented two different techniques based on meta-data: code-based RTS using component meta-data and specification-based RTS using component meta-data. For code-based RTS, it was assumed that each software component was capable of providing structural coverage information, which was fed into the graph-walk algorithm. For specification-based RTS, the component specification was represented in UML state-chart diagrams, which were used by the graph-walk algorithm.

The graph-walk algorithm has also been applied to test web services, despite the challenges that arise from the distributed nature of web services [91–95]. Several different approaches have been introduced to overcome these challenges. Lin *et al.* adopted the JIG-based approach after transforming the web services to a single-JVM local application [91]. Ruth *et al.* collected a coarse-grained CFG from developers of each web service that forms a part of the entire application [92, 93, 95]. Finally, Tarhini *et al.* utilized Timed Labelled Transition System (TLTS), which is a coarse-grained representation of web services that resemble a labelled state machine [94].

4.6. Textual difference approach

Volkolos and Frankl proposed a selection technique based on the textual difference between the source code of two versions of SUT [58, 59]. They identified modified parts of SUT by applying the `diff` Unix tool to the source code of different versions. The source code was pre-processed into canonical forms to remove the impact of cosmetic differences. Although their technique operates on a different representation of SUT, its behaviour is essentially very similar to that of the CFG-based graph-walk approach.

4.7. SDG slicing approach

Bates and Horwitz proposed test case selection techniques based on program slices from Program Dependency Graphs (PDGs) [60]. Bates and Horwitz approach the RTS problem in two stages. First, all the test cases that can be reused for P' need to be identified. Bates and Horwitz introduce the definition of an equivalent execution pattern. If statements s and s' belong to P and P', respectively, s and s' have *equivalent execution patterns* if and only if all of the following hold:

1. For any input file on which both P and P' terminate normally, s and s' are exercised the same number of times.
2. For any input file on which P terminates normally but P' does not, s' is exercised at most as many times as s is exercised.
3. For any input file on which P' terminates normally but P does not, s is exercised at most as many times as s' is exercised.

Using program slicing, Bates and Horwitz categorize statements into execution classes. Statement s from P and s' from P' belong to the same execution class if and only if any test that exercises s will also exercise s'.

Now, a statement s' in P' is *affected* by the modification if and only if one of the following holds:

1. There is no corresponding statement s in P.
2. The behaviour of s' is not equivalent to the corresponding statement s in P.

Equivalent behaviour is determined by PDG slice isomorphism; if the PDG slices of two statements are isomorphic, then those statements share an equivalent behaviour. For each affected statement in P', reusable test cases are selected based on the information retrieved from the identification stage.

While Bates and Horwitz's technique selects test cases for modified or newly added statements in P', it does not select tests that exercise statements that are deleted from P, and therefore is not safe.

Binkley [96, 97] presented a technique based on SDG slicing, which extends Bates and Horwitz's intra-procedural selection technique to inter-procedural test case selection. Binkley introduced the

DOI: 10.1002/stvr

concept of *common execution patterns*, which corresponds to the equivalent execution patterns of Bates and Horwitz, to capture the multiple invocations of a procedure.

4.8. Path analysis

Benedusi *et al.* applied path analysis for test case selection [61]. They construct *exemplar paths* from P and P' expressed in an algebraic expression. By comparing two sets of exemplar paths, they classified paths in P' as new, modified, cancelled or unmodified. Test cases and the paths they execute in P are known; therefore, they selected all the test cases that will traverse modified paths in P'.

One potential weakness of the path analysis approach of Benedusi *et al.* lies not in path analysis itself, but in the potentially over-specific definition of 'modification' used in the post-analysis selection phase. No test cases are selected for the paths that are classified as new or cancelled. However, new or cancelled paths denote modifications that represent differences between P and P'; test cases that execute new or cancelled paths in P' may be modification-revealing. As presented, therefore, the path analysis approach is not safe.

4.9. Modification-based technique

Chen *et al.* introduced a testing framework called TestTube, which utilizes a modification-based technique to select test cases [62]. TestTube partitions the SUT into *program entities*, and monitors the execution of test cases to establish connections between test cases and the program entities that they execute. TestTube also partitions P' into program entities, and identifies program entities that are modified from P. All the test cases that execute the modified program entities in P should be re-executed.

TestTube can be thought of as an extended version of the graph-walk approach. Both techniques identify modifications by examining the program source code, and select test cases that will execute the modified parts. TestTube extends the CDG-based graph-walk technique by introducing program entities that include both functions and entities that are not functions, i.e. variables, data types and pre-processor macros. Any test case that executes modified functions will be selected. Therefore, TestTube is a safe test case selection technique.

One weakness of TestTube is pointer handling. By including variable and data types as program entities, TestTube requires that all value creations and manipulations in a program can be inferred from source code analysis. This is only valid for languages without pointer arithmetic and type coercion. As a result, TestTube makes assumptions; for example, it assumes that all pointer arithmetics are well-bounded. If these assumptions do not hold then safety cannot be guaranteed.

4.10. Firewall approach

Leung and White introduced and later implemented what they called a *firewall* technique for regression testing of system integration [63–66]. The main concept is to draw a firewall around the modules of the system that need to be retested. They categorize modules into the following categories:

- *No Change*: Module A has not been modified, $NoCh(A)$.
- *Only Code Change*: Module A has the same specification but its code has been modified, $CodeCh(A)$.
- *Spec Change*: module A has modified specifications, $SpecCh(A)$.

If a module A calls a module B, there exist 9 possible pairings between the states of A and B. The integration between A and B can be ignored for regression testing if $NoCh(A) \wedge NoCh(B)$, leaving 8 pairings. If both A and B are modified either in code or in specifications, the integration tests between A and B should be executed again as well as the unit tests of A and B; this accounts for 4 of the remaining 8 pairings. The other 4 pairings are cases in which an unchanged module calls a changed module, or vice versa; these pairs form the boundary for integration testing, i.e. the so-called firewall.

DOI: 10.1002/stvr

By considering modules as the atomic entities, Leung and White maintained a very conservative approach to test case selection. If a module has been modified, any test case that tests the integration of the modified module should be selected. Therefore, all modification-traversing test cases will be selected. However, their technique may also select other test cases that execute the modified module, but are not modification-traversing in any way. Leung and White also noted that, in practice, the test suite for system integration is often not very reliable. The low reliability means that it is more likely that there may still exist a fault-revealing test case that does not belong to the test suite, and therefore cannot be selected. Note that it is always a risk that a fault-revealing test case exists outside the given test suite in any type of testing, not only in integration testing. What Leung and White pointed out was that such a risk can be higher in system integration testing due to the generally low quality of test suites.

The Firewall approach has been applied to Object-oriented programs [98–100] and GUIs [101]. Firewall approach has also been successfully applied to RTS for black-box Commercial Off-the-Shelf components. Zheng *et al.* applied the firewall technique of Leung and White based on the information extracted from the deployed binary code [102–105]. Skoglund and Runeson applied the firewall approach to a large-scale banking system [106].

4.11. Cluster identification

Laski and Szemer presented a test case selection technique based on analysis of the CFG of the program under test [67]. Their technique identifies single-entry, single-exit subgraphs of CFG called clusters. Given a program P and its modified version P',

- each cluster in P encapsulates some modifications to P,
- there is a unique cluster in P' that corresponds to the cluster in P, and
- when clusters in each graph are replaced by single nodes, there is a one-to-one correspondence between nodes in both graphs.

The CFGs of the original program and the modified program are reduced using a set of operators such as node collapse and node removal. During the process, if the counterpart of a node from the CFG of the original program cannot be found in the CFG of the modified program, this node is labelled as 'MOD', indicating a modification at the node. Eventually, all the modifications will be enclosed in one or more MOD cluster nodes. As with other test case selection techniques, their technique requires that the tester records the execution history of each test case in the test suite. Once clustering is completed, test case selection is performed by selecting all the test cases for which the corresponding execution path enters any of the MOD clusters.

The strength of the cluster identification technique is that it guarantees to select all modification-traversing test cases regardless of the type of the modification, i.e. addition or deletion of statements and control structures. However, since the clusters can encapsulate much larger areas of the SUT than the scope of actual modification, the technique may also select test cases that are not modification-traversing. In this sense the approach sacrifices precision in order to achieve safety.

4.12. Design-based approach

Briand *et al.* presented a black-box, design level RTS approach for UML-based designs [68, 69]. Assuming that there is traceability between the design and regression test cases, it is possible to perform RTS of code-level test cases from the impact analysis of UML design models. Briand *et al.* formalized possible changes in UML models, and classified the relevant test cases into the categories defined by Leung and White [10]: obsolete, retestable and reusable. They implemented an automated impact analysis tool for UML and empirically evaluated it using both student projects and industrial case studies.

The results showed that the changes made to a model can have a widely variable impact on the resulting system, which, in turn, yields varying degrees of reduction of effort in terms of the number of selected test cases. However, Briand *et al.* noted that the automated impact analysis itself can be valuable, especially for very large systems, such as the cruise control and monitoring system they studied. The UML use-cases of the model of the system had 323 614 corresponding

DOI: 10.1002/stvr

test cases. UML-based models also have been considered by Dent *et al.* [107], Pilskalns *et al.* [108] and Farooq *et al.* [109] for RTS; Le Traon *et al.* [110] and Wu and Offutt [111] considered the use of UML models in the wider context of regression testing in general. Muccini *et al.* considered the RTS problem at the software architecture level, although they did not use UML for the representation [112, 113].

5. TEST CASE PRIORITIZATION

Test case prioritization seeks to find the ideal ordering of test cases for testing, so that the tester obtains maximum benefit, even if the testing is prematurely halted at some arbitrary point. The approach was first mentioned by Wong *et al.* [39]. However, in that work it was only applied to test cases that were already selected by a test case selection technique. Harrold and Rothermel [114, 115] proposed and evaluated the approach in a more general context.

For example, consider the test suite described in Table III. Note that the example depicts an ideal situation in which fault-detection information is known. The goal of prioritization is to maximize early fault detection. It is obvious that the ordering A-B-C-D-E is inferior to B-A-C-D-E. In fact, any ordering that starts with the execution of C-E is superior to those that do not, because the subsequence C-E detects faults as early as possible; should testing be stopped prematurely, this ensures that the maximum possible fault coverage will have been achieved.

Note that the problem definition concerns neither versions of the program under test, nor exact knowledge of modifications. Ideally, the test cases should be executed in the order that maximizes early fault detection. However, fault-detection information is typically not known until the testing is finished. In order to overcome the difficulty of knowing which tests reveal faults, test case prioritization techniques depend on surrogates, hoping that early maximization of a certain chosen surrogate property will result in maximization of earlier fault detection. In a controlled-regression-testing environment, the result of prioritization can be evaluated by executing test cases according to the fault-detection rate.

5.1. Coverage-based prioritization

Structural coverage is a metric that is often used as the prioritization criterion [115–121]. The intuition behind the idea is that early maximization of structural coverage will also increase the chance of early maximization of fault detection. Therefore, while the goal of test case prioritization remains that of achieving a higher fault-detection rate, prioritization techniques actually aim to maximize early coverage.

Rothermel *et al.* reported empirical studies of several prioritization techniques [115, 121]. They applied the same algorithm with different fault-detection rate surrogates. The considered surrogates were: branch-total, branch-additional, statement-total, statement-additional, Fault Exposing Potential (FEP)-total and FEP-additional.

The branch-total approach prioritizes test cases according to the number of branches covered by individual test cases, while branch-additional prioritizes test cases according to the additional

Table III. Example test suite with fault-detection information, taken from Elbaum *et al.* [116]. It is clearly beneficial to execute test case C first, followed by E.

	Fault revealed by test case									
Test case	1	2	3	4	5	6	7	8	9	10
A	x				x					
B	x				x	x	x			
C	x	x	x	x	x	x	x			
D					x					
E								x	x	x

DOI: 10.1002/stvr

number of branches covered by individual test cases. The statement-total and statement-additional approaches apply the same idea to program statements, rather than branches. Algorithmically, 'total' approaches are essentially instances of greedy algorithms, whereas 'additional' approaches are essentially instances of additional greedy algorithms.

The FEP of a test case is measured using program mutation. Program mutation introduces a simple syntactic modification to the program source, producing a mutant version of the program [122]. This mutant is said to be *killed* by a test case if the test case reveals the difference between the original program and the mutant. Given a set of mutants, the mutation score of a test case is the ratio of mutants that are killed by the test case to the total kill-able mutants. The FEP-total approach prioritizes test cases according to the mutation score of individual test cases, while the FEP-additional approach prioritizes test cases according to the additional increase in mutation score provided by individual test cases. Note that FEP criterion can be constructed to be at least as *strong* as structural coverage; to kill a mutant, a test case not only needs to achieve the coverage of the location of mutation but also to execute the mutated part with a set of test inputs that can kill the mutant. In other words, coverage is necessary but not sufficient to kill the mutants.

It is important to note that all the 'additional' approaches may reach 100% realization of the utilized surrogate before every test case is prioritized. For example, achieving 100% branch coverage may not require all the test cases in the test suite, in which case none of the remaining test cases can increase the branch coverage. Rothermel *et al.* reverted to the 'total' approach once such a condition is met.

The results were evaluated using the Average Percentage of Fault-Detection (APFD) metric. Higher APFD values denote faster fault-detection rates. When plotting the percentage of detected faults against the number of executed test cases, APFD can be calculated as the area below the plotted line. More formally, let T be the test suite containing n test cases and let F be the set of m faults revealed by T. For ordering T', let TF_i be the order of the first test case that reveals the ith fault. The APFD value for T' is calculated as follows [123]:

$$APFD = 1 - \frac{TF_1 + \cdots + TF_m}{nm} + \frac{1}{2n}$$

Note that, while APFD is commonly used to evaluate test case prioritization techniques, it is not the aim of test case prioritization techniques to maximize APFD. Maximization of APFD would be possible only when every fault that can be detected by the given test suite is already known. This would imply that all test cases have been already executed, which would nullify the need to prioritize. APFD is computed after the prioritization only to evaluate the performance of the prioritization technique.

Rothermel *et al.* compared the proposed prioritization techniques to random prioritization, optimal prioritization and no prioritization, using the Siemens suite programs. Optimal prioritization is possible because the experiment was performed in a controlled environment, i.e. the faults were already known. The results show that all the proposed techniques produce higher APFD values than random or no prioritization. The surrogate with the highest APFD value differed between programs, suggesting that there is no single best surrogate. However, on average across the programs, the FEP-additional approach performed most effectively, producing APFD value of 74.5% compared to the 88.5% of the optimal approach. It should still be noted that these results are dependent on many factors, including the types of faults used for evaluation and types of mutation used for FEP, limiting the scope for generalization.

Elbaum *et al.* extended the empirical study of Rothermel *et al.* by including more programs and prioritization surrogates [116]. Among the newly introduced prioritization surrogates, function-coverage and function-level FEP enabled Elbaum *et al.* to study the effects of granularity on prioritization. Function-coverage of a test case is calculated by counting the number of functions that the test case executes. Function-level FEP is calculated, for each function f and each test case t, by summing the ratio of mutants in f killed by t. Elbaum *et al.* hypothesized that approaches with coarser granularity would produce lower APFD values, which was confirmed statistically.

Jones and Harrold applied the greedy-based prioritization approach to Modified Condition/Decision Coverage (MC/DC) criterion [124]. MC/DC is a 'stricter form' of branch coverage; it

DOI: 10.1002/stvr

requires execution coverage at condition level. A condition is a Boolean expression that cannot be factored into simpler Boolean expressions. By checking each condition in decision predicates, MC/DC examines whether each condition independently affects the outcome of the decision [125]. They presented an empirical study that contained only an execution time analysis of the prioritization technique and not an evaluation based on fault-detection rate.

Srivastava and Thiagarajan combined the greedy-based prioritization approach with RTS [126]. They first identified the modified code blocks in the new version of the SUT by comparing its binary code to that of the previous version. Once the modified blocks are identified, test case prioritization is performed using greedy-based prioritization, but only with respect to the coverage of modified blocks.

Do and Rothermel applied coverage-based prioritization techniques to the JUnit testing environment, a popular unit testing framework [127]. The results showed that prioritized execution of JUnit test cases improved the fault-detection rate. One interesting finding was that the random prioritization sometimes resulted in an APFD value higher than the *untreated* ordering, i.e. the order of creation. When executed in the order of creation, newer unit tests are executed later. However, it is the newer unit tests that have a higher chance of detecting faults. The empirical results showed that random prioritization could exploit this weakness of untreated ordering in some cases.

Li *et al.* applied various meta-heuristics for test case prioritization [128]. They compared random prioritization, a hill climbing algorithm, a genetic algorithm, a greedy algorithm, the additional greedy algorithm and a two-optimal greedy algorithm. The greedy algorithm corresponds to the *total* approaches outlined above, whereas the additional greedy algorithm corresponds to the *additional* approaches outlined above. The two-optimal greedy is similar to the greedy algorithm except that it considers two candidates at the same time rather than a single candidate for the next order. They considered the Siemens suite programs and the program `space`, and evaluated each technique based on Average Percentage of Block Coverage (APBC) instead of APFD. The results showed that the additional greedy algorithm is the most efficient in general.

5.2. Interaction testing

Interaction testing is required when the SUT involves multiple combinations of different components. A common example would be configuration testing, which is required to ensure that the SUT executes correctly on different combinations of environment, such as different operating systems or hardware options. Each component that can be changed is called a *factor*; the number of choices for each factor is called the *level* of the corresponding factor. As the number of factors and levels of each factor increase, exhaustive testing of all possible combinations of factors becomes infeasible as it requires an exponentially large test suite.

Instead of testing exhaustively, *pairwise* interaction testing requires only that every individual pair of interactions between different factors are included at least once in the testing process. The reduction grows larger as more factors and levels are involved. More formally, the problem of obtaining interaction testing combinations can be expressed as the problem of obtaining a *covering array*, $CA(N; t, k, v)$, which is an array with N rows and k columns; v is the number of levels associated with each factor, and t is the *strength* of the interaction coverage (2 in the case of pairwise interaction testing).

One important research direction in interaction testing is how to efficiently generate an interaction test suite with high interaction coverage. This shares the same basic principles of test case prioritization. For example, the greedy approach aims to find, one by one, the 'next' test case that will most increase the k-way interaction coverage [129, 130], which resembles the greedy approach to test case prioritization. However, the similarities are not just limited to the generation of interaction test suites. Bryce and Colbourn assume that testers may value certain interactions higher than others [131, 132]. For example, an operating system with a larger user base may be more important than one with a smaller user base. After weighting each level value for each factor, they calculate the combined benefit of a given test by adding the weights of each level value selected for the test. They present a Deterministic Density Algorithm (DDA) that prioritizes interaction

DOI: 10.1002/stvr

tests according to their combined benefit. Qu *et al.* compared different weighting schemes used for prioritizing covering arrays [133, 134].

Bryce and Memon also applied the principles of interaction coverage to the test case prioritization of Event-Driven Software (EDS) [135]. EDS takes sequences of events as input, changes state and outputs new event sequences. A common example would be GUI-based programs. Bryce and Memon interpreted t-way interaction coverage as sequences that contain different combinations of events over t unique GUI windows. Interaction coverage-based prioritization of test suites was compared to different prioritization techniques such as unique event coverage (the aim is to cover as many unique events as possible, as early as possible), longest to shortest (execute the test case with the longest event sequence first) and shortest to longest (execute the test case with the shortest event sequence first). The empirical evaluation showed that interaction coverage-based testing of EDS can be more efficient than the other techniques, provided that the original test suite contains higher interaction coverage. Note that Bryce and Memon did not try to generate additional test cases to improve interaction coverage; they only considered permutations of existing test cases.

5.3. Prioritization approaches based on other criteria

While the majority of existing prioritization literature concerns structural coverage in some form or another, there are prioritization techniques based on other criteria [136–139].

Distribution-based Approach: Leon and Podgurski introduced distribution-based filtering and prioritization [136]. Distribution-based techniques minimize and prioritize test cases based on the distribution of the profiles of test cases in the multi-dimensional profile space. Test case profiles are produced by the dissimilarity metric, a function that produces a real number representing the degree of dissimilarity between two input profiles. Using this metric, test cases can be clustered according to their similarities. The clustering can reveal some interesting information. For example:

- Clusters of similar profiles may indicate a group of redundant test cases
- Isolated clusters may contain test cases inducing unusual conditions that are perhaps more likely to cause failures
- Low density regions of the profile space may indicate uncommon usage behaviours

The first point is related to reduction of effort; if test cases in a cluster are indeed very similar, it may be sufficient to execute only one of them. The second and third points are related to fault-proneness. Certain unusual conditions and uncommon behaviours may tend to be harder to reproduce than more common conditions and behaviours. Therefore, the corresponding parts of the program are likely to be tested less than other, more frequently used parts of the program. Assigning a high priority to test cases that execute these unusual behaviours may increase the chance of early fault detection. A good example might be exception handling code.

Leon and Podgurski developed new prioritization techniques that combine coverage-based prioritization with distribution-based prioritization. This hybrid approach is based on the observation that *basic coverage maximization* performs reasonably well compared to *repeated coverage maximization*. Repeated coverage maximization refers to the prioritization technique of Elbaum *et al.* [116], which, after realizing 100% coverage, repeatedly prioritizes test cases starting from 0% coverage again. In contrast, basic coverage maximization stops prioritizing when 100% coverage is achieved. Leon and Podgurski observed that the fault-detection rate of repeated coverage maximization is not as high as that of basic coverage maximization. This motivated them to consider a hybrid approach that first prioritizes test cases based on coverage, then switches to distribution-based prioritization once the basic coverage maximization is achieved. They considered two different distribution-based techniques. The one-per-cluster approach samples one test case from each cluster, and prioritizes them according to the order of cluster creation during the clustering. The failure-pursuit approach behaves similarly, but it adds the k closest neighbours of any test case that finds a fault. The results showed that the distribution-based prioritization techniques could outperform repeated coverage maximization.

Human-based Approach: Tonella *et al.* combined Case-Based Reasoning (CBR) with test case prioritization [137]. They utilized a machine learning technique called *boosting*, which is a

DOI: 10.1002/stvr

framework to combine simple learners into a single, more general and effective learner [140]. They adopted a boosting algorithm for ranking learning called *Rankboost* [141]. The algorithm takes a test suite, T, an initial prioritization index, f, and a set of pairwise priority relations between test cases, Φ, as input. The pairwise priority relation is obtained from comparisons of test cases made by the human tester. The output is a ranking function $H : T \rightarrow \mathbb{R}$ such that, with test cases t_1 and t_2, $t_1 \prec t_2$ if $H(t_1) > H(t_2)$. The ranking function H is then used to prioritize test cases.

They used the statement coverage metric and the cyclomatic complexity computed for the functions executed by test cases as the initial prioritization index. The test suite of the `space` program was considered. In order to measure the human effort required for the learning process, different test suite sizes were adopted, ranging from 10 to 100 test cases. The results were compared to other prioritization techniques including optimal ordering, random prioritization, statement coverage prioritization and additional statement coverage prioritization (the latter two correspond to statement-total and statement-additional respectively).

The results showed that, for all test suite sizes, the CBR approach was outperformed only by the optimal ordering. The number of pairwise relations entered manually showed a linear growth against the size of test suites. Tonella *et al.* reported that for test suites of `space` with fewer than 60 test cases, the CBR approach can be more efficient than other prioritization techniques with limited human effort. Note that empirical evaluation was performed based on an *ideal* user model, i.e. it was assumed that the human tester always makes the correct decision when comparing test cases. One notable weakness of this approach was that it did not scale well. The input from the human tester becomes inconsistent beyond a certain number of comparisons, which in turn limits the size of the learning samples for CBR.

Yoo *et al.* tried to improve the scalability of human-based prioritization approaches by combining pairwise comparisons of test cases with a clustering technique [138]. While the prioritization is still based on the comparisons made by the human tester, the tester is presented with clusters of similar test cases instead of individual test cases. The prioritization between clusters (*inter-cluster* prioritization) is, therefore, performed by the tester. However, the prioritization within each cluster (*intra-cluster* prioritization) is performed based on coverage. After both layers of prioritization are complete, the final ordering of test cases is determined by selecting the test case with the highest priority, determined by intra-cluster prioritization, from the next cluster in the order determined by inter-cluster prioritization, until all the clusters are empty. This is called the Interleaved Clusters Prioritization (ICP) technique.

With the use of clustering, Yoo *et al.* were able to reduce the size of the prioritization problem so that they could apply a more expensive pairwise approach called Analytic Hierarchy Process (AHP). AHP is a pairwise comparison technique developed by the Operations Research community [142] and has been successfully applied to Requirements Engineering [143]. The combination of AHP and ICP has been empirically evaluated for programs and test suites of various sizes, using a more realistic user model (with errors). The results showed that this combination of techniques can be much more effective than coverage-based prioritization. One surprising finding was that sometimes, an error rate higher than 50%, i.e. the human tester making wrong comparisons half the time, did not prevent this technique from achieving higher APFD than coverage-based prioritization. Yoo *et al.* explained this unexpected finding by showing that a certain amount of improvement derived from the effect of clustering. This confirms the argument of Leon and Podgurski about the benefits of distribution-based approach [136]; the clustering sometimes enables the early execution of a fault-revealing test case that would have been assigned low priority due to its low contribution to code coverage.

Probabilistic Approach: Kim and Porter proposed a history-based approach to prioritize test cases that are already selected by RTS [144]. If the number of test cases selected by an RTS technique is still too large, or if the execution costs are too high, then the selected test cases may have to be further prioritized. Since the relevance to the recent change in SUT is assumed by the use of an RTS technique, Kim *et al.* focus on the execution history of each test case, borrowing from statistical quality control. They define the probabilities of each test case tc to be selected at time t as $P_{tc,t}(H_{tc}, \alpha)$, where H_{tc} is a set of t timed observations $\{h_1, \ldots, h_t\}$ drawn from previous

DOI: 10.1002/stvr

runs of tc and α is a smoothing constant. Then the probability $P_{tc,t}(H_{tc}, \alpha)$ is defined as follows:

$$P_0 = h_1$$
$$P_k = \alpha h_k + (1-\alpha) P_{k-1} \quad (0 \leq \alpha \leq 1, k \geq 1)$$

Different definitions of H_{tc} result in different prioritization approaches. For example, Kim *et al.* define Least Recently Used (LRU) prioritization by using test case execution history as H_{tc} with α value that is as close to 0 as possible. The empirical evaluation showed that the LRU prioritization approach can be competitive in a severely constrained testing environment, i.e. when it is not possible to execute all test cases selected by an RTS technique.

Mirarab and Tahvildari took a different probabilistic approach to test case prioritization using Bayesian Networks [145]. The Bayesian Network model is built upon changes in program elements, fault proneness of program elements and probability of each test case to detect faults. Mirarab and Tahvildari extended the approach by adding a feedback route to update the Bayesian Network as prioritization progresses [146]. For example, if a test case covers a set of program elements, the probability of selecting other test cases that cover the same elements will be lowered. Note that this corresponds to the 'additional' approach described by Rothermel *et al.* [115, 121].

History-based Approach: Sherriff *et al.* presented a prioritization technique based on association clusters of software artefacts obtained by a matrix analysis called singular value decomposition [147]. The prioritization approach depends on three elements: association clusters, relationship between test cases and files and a modification vector. Association clusters are generated from a change matrix using SVD; if two files are often modified together as a part of a bug fix, they will be clustered into the same association cluster. Each file is also associated with test cases that affect or execute it. Finally, a new system modification is represented as a vector in which the value indicates whether a specific file has been modified. Using the association clusters and the modification vector, it is then possible to assign each file with a priority that corresponds to how closely the new modification matches each test case. One novel aspect of this approach is that any software artefact can be considered for prioritization. Sherriff *et al.* noted that the faults that are found in non-source files, such as media files or documentation, can be as severe as those found in source code.

Requirement-based Approach: Srikanth *et al.* presented requirement-based test case prioritization [139]. Test cases are mapped to software requirements that are tested by them, and then prioritized by various properties of the mapped requirements, including customer-assigned priority and implementation complexity. One potential weakness of this approach is the fact that requirement properties are often estimated and subjective values. Krishnamoorthi and Sahaaya developed a similar approach with additional metrics [148].

Model-based Approach: Korel *et al.* introduced a model-based prioritization approach [149–151]. Their initial approach was called *selective* prioritization, which was strongly connected to RTS [149]. Test cases were classified into a high priority set, TS_H, and a low priority set, TS_L. They defined and compared different definitions of high and low priority test cases, but essentially a test case is assigned high priority if it is relevant to the modification made to the model. The initial selective prioritization process consists of the random prioritization of TS_H followed by the random prioritization of TS_L. Korel *et al.* developed more sophisticated heuristics based on the dependence analysis of the models [150, 151].

Other Approaches: The use of mutation score for test case prioritization has been analysed by Rothermel *et al.* along with other structural coverage criteria [115, 121]. Hou *et al.* considered interface-contract mutation for the regression testing of component-based software and evaluated it with the *additional* prioritization technique [152].

Sampath *et al.* presented the prioritization of test cases for web applications [153]. The test cases are, in this case, recorded user sessions from the previous version of the SUT. Session-based test cases are thought to be ideal for testing web applications because they tend to reflect the actual usage patterns of real users, thereby making for realistic test cases. They compared different criteria for prioritization such as the number of HTTP requests per test case, coverage of parameter values,

DOI: 10.1002/stvr

frequency of visits for the pages recorded in sessions and the number of parameter values. The empirical evaluations showed that prioritized test suites performed better than randomly ordered test suites, but also that there is not a single prioritization criterion that is always best. However, the 2-way parameter-value criterion, the prioritization criterion that orders tests to cover all pairwise combinations of parameter-values between pages as soon as possible, showed the highest APFD value for 2 out of 3 web applications that were studied.

Fraser and Wotawa introduced a model-based prioritization approach [154]. Their prioritization technique is based on the concept of *property relevance* [155]. A test case is relevant to a model property if it is theoretically possible for the test case to violate the property. The relevance relation is obtained by the use of a model-checker, which is used as the input to the greedy algorithm. While they showed that property-based prioritization can outperform coverage-based prioritization, they noted that the performance of property-based prioritization is heavily dependent on the quality of the model specification.

A few techniques and analyses used for test suite minimization or RTS problem have also been applied to test case prioritization. Rummel *et al.* introduced a prioritization technique based on data-flow analysis by treating each *du* pair as a testing requirement to be covered [156]. Smith *et al.* introduced a prioritization technique based on a call-tree model, which they also used for test suite minimization [26]. They prioritized test cases according to the number of call-tree paths covered by each test case. Jeffrey and Gupta prioritized test cases using relevant slices [157], which was also used for RTS [54]. Each test case was associated with output statements, from which relevant slices were calculated. Then test cases were prioritized according to the sum of two elements: the size of the corresponding relevant slice and the number of statements that are executed by the test case but do not belong to the relevant slice. Both elements were considered to correlate to the chance of revealing a fault introduced by a recent change.

5.4. Cost-aware test case prioritization

Unlike test suite minimization and RTS, the basic definition of test case prioritization does not involve filtering out test cases, i.e. it is assumed that the tester executes the entire test suite following the order given by the prioritization technique. This may not be feasible in practice due to limited resources. A number of prioritization techniques addressed this problem of the need to be cost-aware [23, 24, 118, 158].

With respect to cost-awareness, the basic APFD metric has two limitations. First, it considers all faults to be equally severe. Second, it assumes that every test case costs the same in resources. Elbaum *et al.* extended the basic APFD metric to APFD_c so that the metric incorporates not just the rate of fault detection but also the severity of detected faults and the expense of executing test cases [118]. An ordering of test cases according to the APFD_c metric detects more severe faults at a lower cost. More formally, let T be the set of n test cases with costs $t_1, \ldots, t_n$, and let F be the set of m faults with severity values $f_1, \ldots, f_m$. For ordering T', let TF_i be the order of the first test case that reveals the ith fault. APFD_c of T' is calculated as following:

$$APFD_c = \frac{\sum_{i=1}^{m} (f_i \times (\sum_{j=TF_i}^{n} t_j - \frac{1}{2} t_{TF_i}))}{\sum_{i=1}^{n} t_i \times \sum_{i=1}^{m} f_i}$$

Elbaum *et al.* applied random ordering, additional statement coverage prioritization, additional function coverage prioritization and additional fault index prioritization techniques to `space`, which contains faults discovered during the development stage. They adopted several different models of test case cost and fault severity, including uniform values, random values, normally distributed values and models taken from the Mozilla open source project. The empirical results were achieved by synthetically adding cost severity models to `space`. This enabled them to observe the impact of different severity and cost models. They claimed two practical implications. With respect to test case cost, they proposed the use of many small test cases rather than a few large test cases. Clearly the number of possible prioritizations is higher with a test suite that contains many small test cases, compared to one with a small number of large test cases. It was also claimed that having different models of fault severity distribution can also impact the efficiency of testing.

DOI: 10.1002/stvr

This is true only when the prioritization technique considers the fault-detection history of previous tests.

Elbaum *et al.* compared two different severity distribution models: linear and exponential. In the linear model, the severity values grow linearly as the severity of faults increases, whereas they grow exponentially in the exponential model. If the previous fault-detection history correlates to the fault detection capability of the current iteration of testing, the exponential model ensures that test cases with a history of detecting more severe faults are executed earlier.

Walcott *et al.* presented a time-aware prioritization technique [23]. Time-aware prioritization does not prioritize the entire test suite; it aims to produce a subset of test cases that are prioritized and can be executed within the given time budget. More formally, it is defined as follows:

Given: A test suite, T, the collection of all permutations of elements of the power set of permutations of T, $perms(2^T)$, the time budget, $t_{\max}$, a time function $time: perms(2^T) \rightarrow \mathbb{R}$ and a fitness function $fit: perms(2^T) \rightarrow \mathbb{R}$:

Problem: Find the test tuple $\sigma_{\max} \in perms(2^T)$ such that $time(\sigma_{\max}) \leq t_{\max}$ and $\forall \sigma' \in perms(2^T)$, where $\sigma_{\max} \neq \sigma'$ and $time(\sigma') \leq t_{\max}$, $fit(\sigma_{\max}) > fit(\sigma')$.

Intuitively, a time-aware prioritization technique selects and prioritizes test cases at the same time so that the produced ordered subset yields higher fault-detection rates within the given time budget. Walcott *et al.* utilized a genetic algorithm, combining selection and prioritization into a single fitness function. The selection component of the fitness function is given higher weighting so that it dominates the overall fitness value produced. The results of the genetic algorithm were compared to random ordering, reverse ordering and the optimal ordering. The results showed that time-aware prioritization produces higher rates of fault detection compared to random, initial and reverse ordering. However, Walcott *et al.* did not compare the time-aware prioritization to the existing, non-time-aware prioritization techniques. Note that non-time-aware prioritization techniques can also be executed in 'time-aware' manner by stopping the test when the given time budget is exhausted.

While Yoo and Harman studied test suite minimization [24], their multi-objective optimization approach is also relevant to the cross-cutting concern of cost-awareness. By using multi-objective optimization heuristics, they obtained a Pareto-frontier which represents the trade-offs between the different criteria, including cost. When there is a constraint on cost, the knowledge of Pareto-frontier can provide the tester with more information to achieve higher coverage. The tester can then prioritize the subset selected by observing the Pareto-frontier.

The cost-constraint problem has also been analysed using ILP [159, 160]. Hou *et al.* considered the cost-constraint in web service testing [159]. Users of web services are typically assigned with a usage quota; testing a system that uses web services, therefore, has to consider the remaining quota for each web service. The ILP approach was later analysed in more generic context using execution time of each test as a cost factor [160].

Do and Rothermel studied the impact of time constraints on the cost-effectiveness of existing prioritization techniques [158]. In total, six different prioritization approaches were evaluated: original order, random order, total block coverage, additional block coverage, Bayesian Network approach without feedback, Bayesian Network approach with feedback. They considered four different time constraints, each of which allows $\{25\%, 50\%, 75\%, 100\%\}$ of time required for the execution of all test cases. Each prioritization approach was evaluated under these constraints using a cost-benefit model. The results showed that, although time constraints affect techniques differently, it is always beneficial to adopt some prioritization when under time constraints. The original ordering was always affected the most severely.

6. META-EMPIRICAL STUDIES

Recently, the meta-empirical study of regression testing techniques has emerged as a separate subject in its own right. It addresses cross-cutting concerns such as cost-benefit analysis of regression testing techniques and the studies of evaluation methodology for these techniques. Both studies

DOI: 10.1002/stvr

seek to provide more confidence in efficiency and effectiveness of regression testing techniques. Work in these directions is still in the early stages compared to the bodies of work available for minimization, selection or prioritization techniques. However, it is believed that these studies will make significant contributions towards the technology transfer.

Empirical evaluation of any regression testing technique is inherently a *post-hoc* process that assumes the knowledge of a set of known faults. Without the *a priori* knowledge of faults, it would not be possible to perform a controlled experiment comparing different regression testing techniques. This poses a challenge to the empirical evaluation of techniques, since the availability of fault data tends to be limited [161].

Andrews *et al.* performed an extensive comparison between real faults and those seeded by mutation [161]. One concern when using mutation faults instead of real faults is that there is no guarantee that the detection of mutation faults can be an accurate predictor of the detection of real faults. After considering various statistical data such as the ratio and distribution of fault detection, Andrews *et al.* concluded that mutation faults can indeed provide a good indication of the fault detection capability of the test suite, assuming that mutation operators are carefully selected and equivalent mutants are removed. However, they also note that, while mutation faults were not easier to detect than real faults, they were also not harder to detect. Do and Rothermel extended this study by focusing the comparison on the result of test case prioritization techniques [162, 163]. Here, they considered whether evaluating prioritization techniques against mutation faults and seeded faults differs. Based on the comparison of these two evaluation methods, it was concluded that mutation faults can be safely used in place of real or hand-seeded faults.

Although it was not their main aim, Korel *et al.* made an important contribution to the empirical evaluation methodology of regression testing techniques through the empirical evaluation of their prioritization techniques [149–151]. They noted that, in order to compare different prioritization techniques in terms of their rate of fault detection, they need to be evaluated using all possible prioritized sequences of test cases that may be generated by each technique. Even deterministic prioritization algorithms, such as the greedy algorithm, can produce different results for the same test suite if some external factors change; for example, if the ordering of the initial test suite changes, there is a chance that the greedy algorithm will produce a different prioritization result. Korel *et al.* argued, therefore, that the rate of fault detection should be measured in average across all possible prioritized sequences. They introduced a new metric called Most Likely average Position, which measures the average relative position of the first test case that detects a specific fault.

Elbaum *et al.* extended the empirical studies of prioritization techniques with the Siemens suite and `space` [116] by performing statistical analysis of the variance in APFD [117]. The APFD values were analysed against various program, change and test metrics. Program metrics included mean number of executable statements, mean function size across all functions, etc. Change metrics included number of functions with at least one changed statement, number of statements inserted or deleted, etc. Test metrics included number of tests in the test suite, percentage of tests reaching a changed function, mean number of changed functions executed by a test over a test suite, etc. The aim was to identify the source of variations in results. Elbaum *et al.* reported that the metrics that reflected normalized program characteristics (such as mean function size across the program) and characteristics of test suites in relation to programs (such as mean percentage of functions executed by a test over a test suite) were the primary contributors to the variances in prioritization. While they reported that this finding was not the anticipated one, it showed that the prioritization results are the product of closely coupled interactions between programs under test, changes and test suites.

Empirical evaluation of different techniques can benefit from a shared evaluation framework. Rothermel and Harrold presented a comparison framework for RTS techniques [45], which was used to compare different RTS techniques [1]. While minimization and prioritization techniques lack such a framework, certain metrics have been used as a *de facto* standard evaluation framework. Rate of reduction in size and rate of reduction in fault-detection capability have been widely used to evaluate test suite minimization techniques [12, 18–21, 25, 26, 38–40, 43]. Similarly, APFD [116] has been widely used to evaluate prioritization techniques [23, 116, 117, 120, 121, 123, 127, 136–138, 145, 146, 154, 156, 157, 159, 160, 162–165].

DOI: 10.1002/stvr

Rothermel *et al.* studied the impact of test suite granularity and test input grouping on the cost-effectiveness of regression testing [120, 165]. They first introduced the concept of *test grains*, which is the smallest unit of test input that is executable and checkable. Test cases are constructed by grouping test grains. Based on this, they defined test suite granularity as the number of test grains in a test case, and test input grouping as the way test grains are added to each test case, e.g. randomly or grouped by their functionality. They reported that having a coarse-grained test suite did not significantly compromise the fault-detection capability of the test suite, but resulted in decreased total execution time. The savings in execution time can be explained by the fact that a coarse-grained test suite contains fewer test cases, thereby reducing the set-up time and other overheads that occur between execution of different test cases. However, they did not consider the cost of the test oracle. It is not immediately obvious whether the cost of a test oracle would increase or decrease as the test suite granularity increases. This oracle cost could affect the overall cost-effectiveness.

Kim *et al.* studied the impact of test application frequency on the cost-effectiveness of RTS techniques [166, 167]. Their empirical studies showed that the frequency of regression test application has a significant impact on the cost-effectiveness of RTS techniques. They reported that RTS techniques tend to be more cost-effective when the frequency of test application is high. It implies that only a small amount of changes are made between tests, which makes RTS more effective. However, as intervals between tests grow, changes are accumulated and RTS techniques tend to select more and more test cases, resulting in low cost-effectiveness. One interesting finding is that, as intervals between tests grow, random re-testing tends to work very well. With small testing intervals, the random approach fails to focus on the modification. As testing intervals increase, more parts of SUT need to be re-tested, improving the effectiveness of the random approach. Elbaum *et al.* studied the impacts of changes in terms of the quantitative nature of modifications [164]. They investigated how the cost-effectiveness of selection and prioritization techniques is affected by various change metrics such as percentage of changed lines of code (LOC), average number of LOC changed per function, etc. Their empirical analysis confirmed that the differences in these metrics can make a significant impact on the cost-effectiveness of techniques. However, they also reported that simple size of change, measured in LOC, was not a predominant factor in determining the cost-effectiveness of techniques. Rather, it was the distribution of changes and the ability of test cases to reach these changes.

Elbaum *et al.* also presented a technique for selecting the most cost-effective prioritization technique [168]. They applied a set of prioritization techniques to the same set of programs, and analysed the resulting APFD metric values. Different techniques perform best for different programs; they applied the classification tree technique to predict the best-suited technique for a program. Note that the term 'cost-effectiveness' in this work means the efficiency of a prioritization technique measured by the APFD metric; the computational cost of applying these techniques was not considered.

Rosenblum and Weyuker introduced a coverage-based cost-effective predictor for RTS techniques [169]. Their analysis is based on the coverage relation between test cases and program entities. If each program entity has a uniformly distributed probability of being changed in the next version, it is possible to predict the average number of test cases to be selected by a safe RTS technique using coverage relation information. They evaluated their predictor with the TestTube RTS tool [62], using multiple versions of the `KornShell` [170] and an I/O library for Unix, `SFIO` [171], as subjects. Their predictor was reasonably accurate; for example, it predicted an average of 87.3% of the test suite to be selected for `KornShell`, when TestTube selected 88.1%. However, according to the cost model of Leung and White [172], the cost of coverage analysis for RTS per test case was greater than the cost of execution per test case, indicating that TestTube was not cost-effective. Harrold *et al.* introduced an improved version of the cost-effective predictor of Rosenblum *et al.* for more accurate cost-effectiveness prediction of version-specific RTS [173]. They evaluated their predictor using TestTube and another RTS tool, DejaVu [57].

Modelling the cost-effectiveness of regression testing techniques has emerged as a research topic. This is motivated by the observation that any analysis of cost-effectiveness should depend

DOI: 10.1002/stvr

on some model. Leung and White introduced an early cost-model for regression testing strategies and compared the cost models of the *retest-all* strategy and the selective retesting strategy [172]. Malishevsky *et al.* presented detailed models of cost-benefit trade-offs for regression testing techniques [119]. They applied their models to the regression testing of `bash`, a popular Unix shell [174], with different ratio values of $f/(e+c)$, where f is the cost of omitting one fault, e is the additional cost per test and c is the result-validation cost per test. The results implied that if a regression testing technique does not consider f, it may overestimate the cost-effectiveness of a given technique. The cost model of Malichevsky *et al.* has been extended and evaluated against the prioritization of JUnit test cases [175]. Smith and Kapfhammer studied the impact of the incorporation of cost into test suite minimization [176]. Existing minimization heuristics including HGS [12], delayed greedy [18] and 2-optimal greedy algorithm [128] were extended to incorporate the execution cost of each test case. Do and Rothermel considered the impact of time constraints on selection and prioritization techniques across multiple consecutive versions of subject programs to incorporate software life-cycle factors into the study [177].

Reflecting the complexity of regression testing process, cost-effectiveness models often need to be sophisticated in order to incorporate multiple variables [119, 175–177]. However, complexity can be a barrier to uptake. Do and Rothermel introduced an approach based on statistical sensitivity analysis to simplify complicated cost models [178]. Their approach fixed certain cost factors that were deemed to be the least significant by the sensitivity analysis. The empirical evaluation showed that, while certain levels of simplification can still preserve the accuracy of the model, over-simplification may be risky.

7. SUMMARY & DISCUSSION

7.1. Analysis of current global trends in the literature

This paper has produced a survey of 159 papers on test suite minimization, RTS and test case prioritization. This number includes papers on methodologies of empirical evaluation and comparative studies. Data summarizing the results in these papers are shown in Tables AI–AIV in the Appendix. Note that the last category consists of papers on cross-cutting concerns for empirical studies, such as methodologies of empirical evaluation and analyses of cost-effectiveness, as well as purely comparative studies and surveys. Figure 2 plots the number of surveyed papers for each year since 1977, when Fischer published his paper on RTS using a linear programming approach [47]. The observed trend in the number of publications shows that the field continues to grow.

Figure 3 shows the chronological trend in the number of studies for each of the topics in this paper. In this figure, the papers have been classified into four different categories. The first three

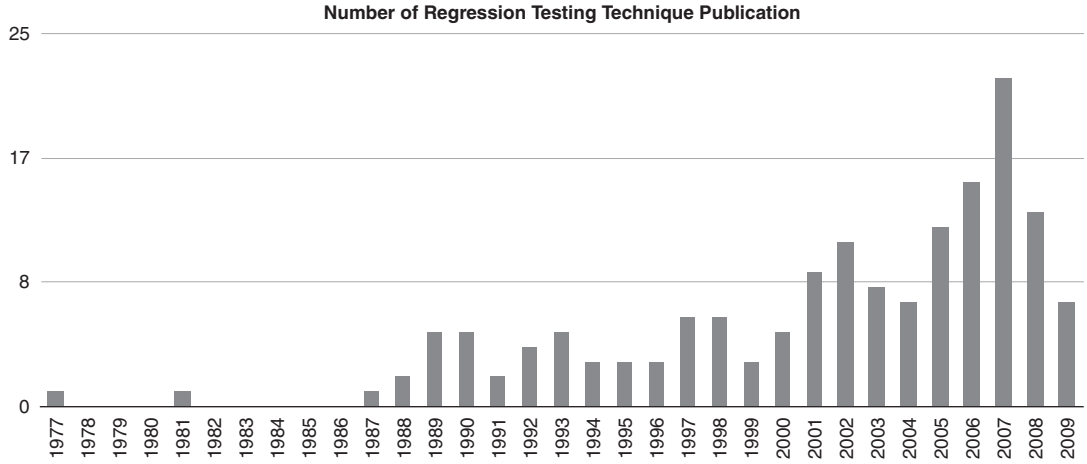

Figure 2. Number of surveyed papers in each year since 1977. The field is still growing.

DOI: 10.1002/stvr

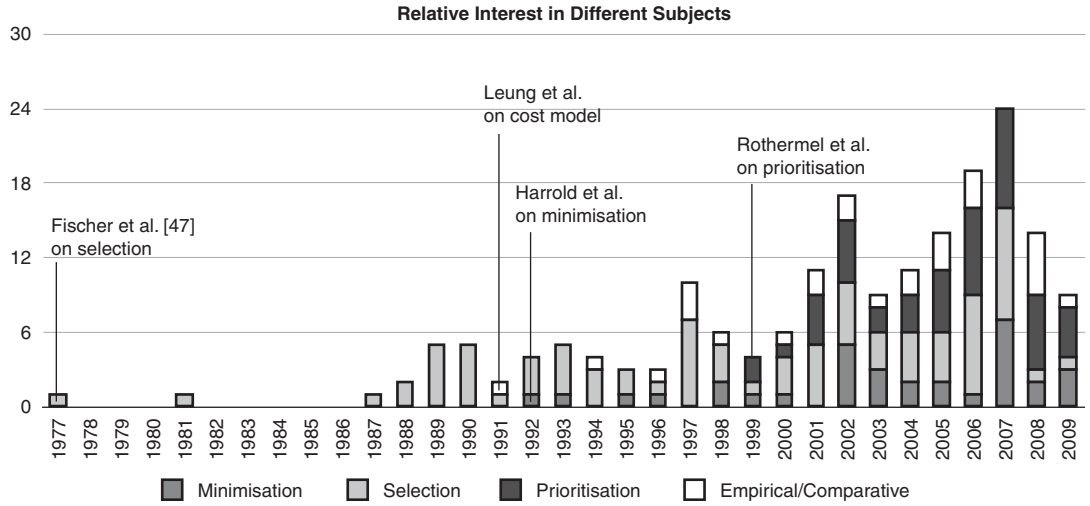

Figure 3. Relative research interest in each subject. Papers that consider more than one subject were counted multiple times.

categories contain papers on minimization, selection and prioritization, respectively. The fourth category contains papers on empirical evaluation and comparative studies, including previous surveys. Papers that consider more than one subject are represented in each category for which they are relevant; for example, a survey on RTS [1] is counted in both the selection category and the comparative studies category. Therefore, while the graph closely resembles Figure 2, it is not a representation of the number of publications. Rather, the figure should be read as a guide to the trends in study topics over time.

Considering that many different RTS approaches were introduced in the late 80s and 90s, recent research on RTS techniques has been mostly concerned with the application and evaluation of the graph-walk approach [83, 84, 87, 90–95]. As the figure reveals, the interest in test case prioritization has been steadily growing since the late 90s. In Figures 2 and 3, the data for 2009 is, of course, partial.

Whereas most of the early papers on RTS were theoretical (Table AII), empirical evaluation of regression testing techniques has recently received a burgeoning interest. Not only are there more publications on pure empirical/comparative studies (as can be observed in Figure 3), but recent studies of regression testing techniques tend to evaluate the suggested techniques empirically, as can be observed in Tables AI–AIII.

However, the scale of empirical studies seems to remain limited. Figure 4 shows the maximum size of SUTs (measured in LOC) and test suites (measured as the number of test cases) studied empirically in the literature. For both data, only the empirical studies that explicitly note the size of subject SUTs and test suites have been included. When only the average size of test suites is given, the maximum average size of studied test suites has been used. For the maximum size of SUTs, only the empirical studies that use source code as test subjects have been included; for example, studies of regression testing of UML models are not included. For about 60% of empirical studies, the largest SUT studied is smaller than 10 000 LoC. For about 70% of empirical studies, the largest test suite studied contains fewer than 1000 test cases.

Figure 5 shows the origins of subject SUTs studied in the literature. For detailed information about the classification, refer to the Appendix. Programs available from the Software Infrastructure Repository (SIR) [179] account for over 50% of subjects of empirical studies of regression testing techniques. The predominant programming language is C, followed by Java. Considering that the first paper appeared in 2002, model-based techniques have shown significant growth. Although there are several papers on RTS for web applications [91–95], empirical evaluation of these techniques remains limited.

DOI: 10.1002/stvr

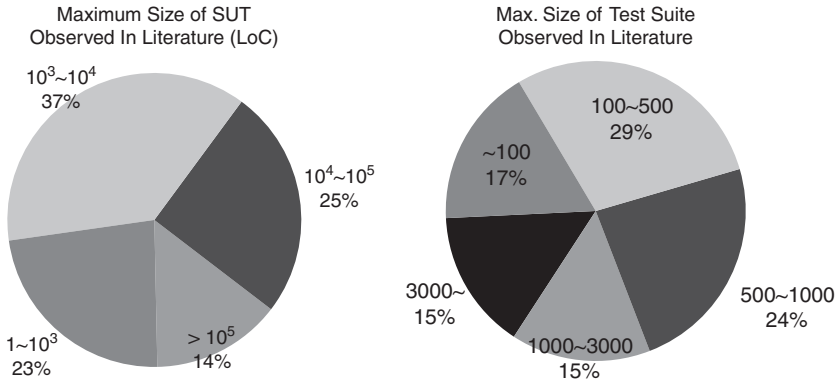

Figure 4. Maximum size of SUT and test suites studied in the literature.

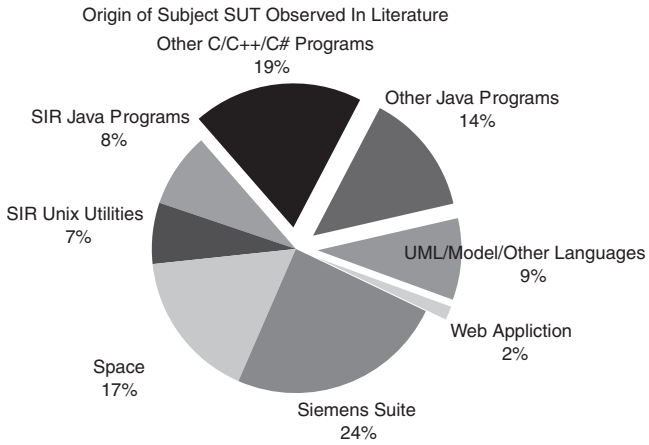

Figure 5. Origins of subject SUTs observed in the literature.

7.2. State-of-the-art, trends and issues

7.2.1. State-of-the-art Among the class of RTS techniques, the graph-walk approach seems to be the predominant technique in the literature. Although it was originally conceived for CDGs and PDGs [46, 56], the most widely used form of graph-walk approach works on CFGs [57, 79]. Its popularity can be observed from the fact that the approach has been applied to many forms of graph representation of SUT that are not CFGs [82–84, 87]. Indeed, the strength of the graph-walk approach lies not only in the fact that it is intuitive but also in the generic applicability of the technique to any graph representation of systems.

By studying the literature, it becomes clear that two ideas played essential roles in the development of RTS techniques: Leung and White's early idea of regression testing and test case classification [10], and Rothermel and Harrold's definition of a *safe* RTS [45]. Collectively, these two ideas provided a profound theoretical framework that can be used to evaluate RTS techniques.

The existence of such a theoretical framework is what differentiate RTS techniques from test suite minimization and test case prioritization. For RTS techniques, it is possible to define what a *safe* technique should do because the RTS problem is specifically focused on the modifications between two versions of SUT. Minimization and prioritization techniques, on the other hand, are forced to rely on surrogate metrics for real fault-detection capability. Therefore, it may not be clear what 'safe' minimization or prioritization techniques would mean. A minimization technique would not be safe unless the surrogate metric perfectly captures the fault-detection capability of the test suite; however, the empirical studies so far have shown that there is no such single metric.

DOI: 10.1002/stvr

For prioritization, the concept of 'safe' and 'non-safe' does not apply directly since the aim of the technique is to permute the test suite and not to select a subset out of it.

Naturally, the history of minimization and prioritization literature is an on-going exploration of different heuristics and surrogate metrics. It is interesting to note that the greedy algorithm, a good approximation for the set cover problem and, therefore, test suite minimization problem, is also an efficient heuristic for test case prioritization precisely because of its *greedy* nature. In other words, *as much as possible*, *as soon as possible*. As a result of this, the greedy approach and its variations have a strong presence in the literature on both test suite minimization [11, 14, 18] and test case prioritization [115, 116, 121]. Recently, there are other approaches to test suite minimization and test case prioritization that aim to overcome the uncertainty of surrogates, e.g. the use of multiple minimization criteria [21, 22, 24] and the use of expert knowledge for test case prioritization [137, 138].

7.2.2. Trends Emphasis on Models: Whereas most of the early regression testing techniques concerned code-based, white-box regression testing, the model-based regression testing approach has been of more recent growing interest [32–34, 68, 69, 88, 107–111]. UML models and EFSMs are often used.

New Domains: While the majority of existing literature on regression testing techniques concerns stand-alone programs written in C and Java, there are a growing number of other domains that are being considered for regression testing. For example, spreadsheets [76], GUIs [25, 180] and web applications [91–95, 153] have been considered.

Multi-criteria Regression Testing: In minimization and prioritization, it is known that there is no single surrogate metric that correlates to fault prediction capability for all programs [8, 38–40, 115, 121]. One potential way of overcoming this limitation is to consider multiple surrogates simultaneously [21, 22] using classical multi-objective optimization techniques, such as the weighted-sum approach [21] or the prioritized approach [22]. Expert domain knowledge has also been used in addition to software metrics for test case prioritization [137, 138]. Finally, Multi-Objective Evolutionary Algorithms (MOEAs) have been used to deal with multiple objectives [24].

Cost-awareness: The cost of regression testing is a cross-cutting concern for all three classes of regression testing techniques. Studies of cost-aware regression testing can be classified into two categories. First, there is work that aims to incorporate the cost of testing directly into regression testing techniques at technical level; for example, cost-aware test case prioritization [23, 118] or minimization techniques that provide the tester with a series of alternative subsets of the original test suites that can be executed in different amounts of time [24]. Second, there are empirical evaluations of regression testing techniques that consider whether the application of these techniques is indeed cost-effective in the wider context of the overall software life-cycle [120, 158, 165, 169, 177].

7.2.3. Issues Limited Subjects: In Figure 5, subject programs from SIR [179] account for almost 60% of the subjects of empirical studies observed in the regression testing technique literature. While this is certainly evidence that SIR has been of tremendous value to the research community, it also means that many regression testing techniques are being evaluated against a limited set of programs and test suites. By no means is this a criticism of SIR itself or work that is based on subjects from SIR; rather, the dependency on SIR shows how time consuming and difficult it is to collect multiple versions of program source code, their test suites and associated fault information, let alone to make it publicly available. The SIR commonality also supports and facilitates comparisons. However, the community faces a possible risk of 'over-fitting' the research of these techniques to those programs that are easily available. Open source software projects are often suggested as an alternative source of data, but from the results of this survey analysis it seems that their use for research on regression testing techniques is limited.

Two potential approaches to the issue of limited subjects can be envisioned. The first approach is to design a method that will allow a realistic simulation of real software faults. There is early work in this direction. Andrews *et al.* considered whether mutation faults are similar to real faults [161];

DOI: 10.1002/stvr

Do and Rothermel studied the same question especially in the context of regression testing [163]. Jia and Harman used a search-based approach to obtain higher-order mutation faults that are more 'subtle' and, therefore, potentially harder to detect than first-order mutation faults [181].

The second approach is to engage more actively with industry and open source communities. This is not an easy task, because the information about testing and software faults tends to be sensitive, particularly to commercial organizations. The research community faces the challenge of convincing the wider practitioner community of the cost-effectiveness and usefulness of these techniques. This is closely related to the issue of technology transfer, to which the paper now turns.

Technology Transfer: A detailed observation of the literature suggests that the community may have reached a stage of maturity that, in order to progress to the next level of achievement, technology transfer to industry will play an important role. While the research community may not have found the 'silver bullet', most empirical evaluation of the proposed techniques suggests that application of minimization, selection and prioritization techniques does make a difference from (1) uncontrolled regression testing, i.e. retest-all approach and un-prioritized regression testing, and (2) random approaches.

However, empirical evaluation and application of regression testing techniques at industrial level seems to remain limited [182]. Out of the 159 papers listed in Tables AI–AIV, only 31 papers list a member of industry as an author or a co-author. More importantly, only 12 papers consider industrial software artefacts as a subject of the associated empirical studies [68, 69, 88, 99–102, 104–106, 126, 147]. This suggests that a large-scale industrial uptake of these techniques has yet to occur.

8. FUTURE DIRECTIONS

This section discusses some of the possible future directions in the field of regression testing techniques. While it is not possible to predict the future direction that a field of study will follow, it was possible to identify some trends in literature, which may suggest and guide the direction of future research.

8.1. Orchestrating regression testing techniques with test data generation

Automatic test data generation has made advances in both functional and non-functional testing [183]. Superficially test data generation is the counterpart to regression testing techniques; it creates test cases while regression testing seeks to mange them. However, *because* these two activities are typically located at the opposite ends of the testing process, they may become close when the testing process repeats.

Orchestrating regression testing with test data generation has a long heritage. From the classification of test cases by Leung and White [10], it follows that regression testing involves two different needs that are closely related to test data generation: repairing obsolete test cases for corrective regression testing and generating additional test data for progressive regression testing. The second problem, in particular, has been referred to as the test suite augmentation problem [184, 185]. There was early work on both problems. Memon and Soffa considered automatic repair of GUI test cases that were made obsolete by the changes in the GUI [180], which was later extended by Memon [186]. Similarly, Alshahwan and Harman focused on the use of user session data for regression testing of web applications, and how the test cases can be automatically repaired for the next version [187]. Apiwattanapong *et al.* identified the testing requirements that are needed to test the new and modified parts of an SUT so that additional test data can be generated automatically [184, 185].

However, it is believed that there are additional areas that may be synergetic. For example, test data generation might possibly refer to the test cases selected and unselected during the last iteration in order to identify the part of the SUT to focus on. Similarly, regression testing techniques can use the additional information provided by test data generation techniques in order to make regression testing more efficient and effective. For example, there are test data generation

DOI: 10.1002/stvr

techniques that target a specific concern in the SUT, such as detection of the presence of a memory leak. The additional information about the *intention* behind each test case could be used to enrich the minimization, selection and prioritization process.

8.2. Multi-objective regression testing

Regression testing is a complex and costly process that may involve multiple objectives and constraints. For example, the cost of executing a test case is usually measured as the time taken to execute the test case. However, there may be a series of different costs involved in executing a test case, such as setting up the environment or preparing a test input, each of which may be subject to a different constraint. Existing techniques also assume that test cases can be executed in any given order without any change to the cost of execution, which seems unrealistic. Test cases may have dependency relations between them. It may also be possible to lower the cost of execution by grouping test cases that share the same test environment, thereby saving set-up time.

Considering the complexity of real-world regression testing, existing representations of problems in regression testing may be over-simplistic. Indeed, most of the published empirical studies rely on relatively small-scale academic examples. Even when real-world programs are studied, they tend to be individual programs, not a software system as a whole. Larger software systems do not simply entail larger problem size; they may denote a different level of complexity.

It is believed that, in order to cope with the complexity of regression testing, regression testing techniques may need to become multi-objective. There is existing, preliminary work that seeks to represent the cost of test case execution as an additional constraint using evolutionary algorithms [23, 24]. Multi-objective optimization heuristics may provide the much-needed flexibility that is required for the representation of problems with high complexity.

Another benefit of moving to a multi-objective paradigm is the fact that it provides additional insight into the regression testing problem by allowing the testers to observe the inherent trade-offs between multiple constraints. This is not possible with the so-called *classical* approaches to multi-objective problems that either consider one objective at a time [22] or conflate multiple objectives into a single objective using weighting [21, 23]; these approaches may be based on *multiple criteria*, but they are not truly multi-objective in the sense that they all produce a single solution. The result of a multi-objective optimization is often a set of solutions that do not dominate each other, thereby forming the trade-offs between constraints. The insight into the trade-offs may provide additional information that is hard to obtain manually.

8.3. Problem of test oracle and its cost

Test oracles present a set of challenging problems for software testing. It is difficult to generate them automatically, they often require human efforts to verify and the cost of this effort is hard to estimate and measure. The oracle cost has been considered as a part of cost models [178], but has not been considered as a part of the process of minimization, selection and prioritization itself. Since regression testing techniques seek to efficiently re-use existing test cases, information about the cost of verifying the output observed with the existing test suite may be collected across versions. This can be incorporated into the existing regression testing techniques.

While a test oracle and its cost may be seen as yet another additional objective that can be considered using a multi-objective approach, it can be argued that the test oracle will present many interesting and exciting research questions in the context of regression testing and, thus, deserves a special treatment in its own right. This is because, compared to other testing cost such as the physical execution time of test cases, the test oracle cost is closely related to the *quality* of testing. Moreover, unlike some costs that can be reduced by using more advanced hardware, the cost of oracle verification derives from human effort and is, therefore, harder to reduce. These characteristics make the issues related to test oracles challenging but interesting research subjects.

DOI: 10.1002/stvr

8.4. Consideration of other domains

The majority of regression testing techniques studied in this survey concern white-box structural regression testing of code-level software artefacts. However, other domains are emerging as new and exciting research subjects.

Recently, Service-Oriented Architectures (SOAs) have been of keen interest both for academic researchers and industrialists. In the SOA paradigm, software system is built, or *composed*, by orchestrating a set of web services, each of which takes charge of a specific task. Several approaches have been introduced to address the issue of regression testing of web services, most of which seek to apply the same technique developed for traditional applications to web services [91–95]. However, the inherently distributed nature of an SOA system presents several challenges that are alien to traditional regression testing techniques.

Web services often reside in remote locations and are developed by a third-party, making it hard to apply the traditional white-box regression testing techniques that require analysis of source code. Modifications can happen across multiple services, which can make fault localization difficult. High interactivity in web applications may result in complex test cases that may involve human interaction. Finally, distributed systems often contain concurrency issues. Traditional regression testing techniques assume that the program produces deterministic output. This may not be adequate for testing applications with concurrency. Answers to these specific issues in regression testing of web applications are still in the early stage of development.

Model-based regression testing techniques have also received growing interests [68, 69, 107–111]. It is believed that the model-based regression testing techniques will be of crucial importance in the future for the following reasons:

- *Higher level regression testing*: These techniques can act as a medium between requirement/specification and testing activities, bringing regression testing from the structural level to functional level.
- *Scalability*: in dealing with software systems of industrial scale, model-based techniques will scale up better than code-based techniques.

However, there are a few open research questions. First, there is the well-known issue of traceability. Unless the traceability from requirements and specifications to code-level artefacts and test cases is provided, the role of model-based regression testing techniques will be severely limited. Second, there is the issue of test adequacy: if a test adequacy A is appropriate for a model M, which test adequacy should be used to test the program P that has been automatically generated from M? Does A still apply to P? If so, does it follow that M being adequate for A means P will be adequate for A as well?

There are also other interesting domains to consider. Testing of GUIs has received growing interest, not only in the context of regression testing [25, 135, 180], but also in the context of testing in general [188, 189]. Regression testing GUIs present a different set of challenges to code-based structural regression testing since GUIs are usually generated in a visual programming environment; they are often subject to frequent changes and, not being well-typed, do not readily facilitate static analysis.

8.5. Non-functional testing

A majority of existing regression testing techniques rely upon structural information about the SUT, such as data-flow analysis, CFG analysis, program slices and structural coverage. The impact that non-functional property testing will have on regression testing techniques has not been fully studied. Existing techniques were able to map the problems in the regression testing to well-formed abstract problems using the properties of structural information. For example, test suite minimization could be mapped to the minimal hitting set problem or the set coverage problem, precisely because the techniques were based on the concept of ‘coverage’. Similarly, graph-walking approaches to test case selection were made possible because the changes between different versions were defined by structural differences in CFGs.

DOI: 10.1002/stvr

Imagine regression testing techniques for non-functional properties. What would be the minimized test suite that can test the power consumption of an embedded system? How would test cases be prioritized to achieve an efficient and effective stress testing of a web application? These questions remain largely unanswered and may require approaches that are significantly different from existing paradigms.

8.6. Tool support

Closely related to the issue of technology transfer is the issue of tool support. Without readily available tools that implement regression testing techniques, practical adoption will remain limited. One potential difficulty of providing tool support is the fact that, unlike unit testing for which there exists a series of frameworks based on the xUnit architecture, there is not a common framework for the regression testing process in general. The closest to a common ground for regression testing would be an Integrated Development Environment (IDE), such as Eclipse, with which the xUnit architecture is already integrated successfully. A good starting point for regression testing techniques may be the management framework of unit test cases, built upon xUnit architecture and IDEs.

9. CONCLUSION

This paper provides both a survey and a detailed analysis of trends in regression test case selection, minimization and prioritization. The paper shows how the work on these three topics is closely related and provides a survey of the landscape of work on the development of these ideas, their applications, empirical evaluation and open problems for future work.

The analysis of trends reported in the paper reveals some interesting properties. There is evidence to suggest that the topic of test case prioritization is of increasing importance, judging by the shift in emphasis towards it that is evident in the literature. It is also clear that the research community is moving towards assessment of the complex trade-offs and balances between different concerns, with an increase in work that considers the best way in which to incorporate multiple concerns (cost and value for instance) and to fully evaluate regression testing improvement techniques.

This focus on empirical methodology is one tentative sign that the field is beginning to mature. The trend analysis also indicates a rising profile of publication, providing evidence to support the claim that the field continues to attract growing attention from the wider research community, which is a positive finding for those working on regression test case selection and minimisation and, in particular those working on prioritization problems.

Our survey also provides evidence to indicate that there is a preponderance of empirical work that draws upon a comparatively small set of subjects (notably those available through the SIR repository). This is a testament to the importance of this source of case study material. It is valuable because it allows for cross comparison of results and replication, which is essential for the development of any science. However, it may potentially suggest a risk of over-fitting.

APPENDIX A

The following tables contain detailed information about publications on each class of technique: test suite minimization in Table AI, RTS in Table AII, test case prioritization in Table AIII and empirical/comparative studies in Table AIV. Note that publications that consider more than one class of techniques appear in multiple tables for ease of reference. For example, a survey of Regression Test Selection [1] appears in both Tables AII and AIV.

Information on the maximum size of SUT (measured in LOC) and the maximum size of test suites were collected from papers only when they were explicitly noted. Some papers that considered multiple test suites for a single program contained only the average, in which case the size of the test suite with largest average size was recorded. When the studied SUT cannot be measured in

DOI: 10.1002/stvr

Table AI. Summary of publications on test suite minimization.

Reference	Year	Max. SUT size (LoC)	Max. Test suite size	Siemens suite	`space`	SIR `Unix` utilities	SIR Java Programs	Other C/ C++/C#	Other Java programs	Models and other languages	Web applications
Horgan *et al.* [13]	1992	1000	26					•			
Harrold *et al.* [12]	1993	N/A	19					•			
Offutt *et al.* [14]	1995	48	36.8					•			
Chen *et al.* [11]	1996	N/A*	—								
Rothermel *et al.* [38]	1998	516	260	•							
Wong *et al.* [39]	1998	842	33					•			
Wong *et al.* [40]	1999	6218	200		•						
Schroeder and Korel [31]	2000	Theory	—								
Korel *et al.* [33]	2002	Theory	—								
Malishevsky *et al.* [119]	2002	65 632	1168			•					
Rothermel *et al.* [120]	2002	68 782	1985			•		•			
Rothermel *et al.* [8]	2002	516	260	•							
Vaysburg *et al.* [32]	2002	Theory	—								
Anido *et al.* [35]	2003	Theory	—								
Harder *et al.* [27]	2003	6218	169	•	•						
Marré and Bertolino [17]	2003	516	5542	•							
Black *et al.* [21]	2004	512	5542	•							
Rothermel *et al.* [165]	2004	68 782	1985			•		•			
Jeffrey and Gupta [19]	2005	516	135	•							
McMaster and Memon [43]	2005	6218	4712		•						
Tallam anf Gupta [18]	2006	6218	539	•	•						
Chen *et al.* [34]	2007	Theory	—								
Hou *et al.* [152]	2007	5500	183						•		
Jeffrey and Gupta [20]	2007	6218	1560	•	•						
Leitner *et al.* [29]	2007	N/A†	—								
McMaster and Memon [42]	2007	11 803	1500						•		
Smith *et al.* [26]	2007	1455	N/A						•		
Yoo *et al.* [24]	2007	6218	169	•	•						

DOI: 10.1002/stvr

Table AI. *Continued.*

Reference	Year	Max. SUT size (LoC)	Max. Test suite size	Siemens suite	`space`	SIR `Unix` utilities	SIR Java Programs	Other C/C++/C#	Other Java programs	Models and other languages	Web applications
McMaster and Memon [25]	2008	11 803	1500						•		
Yu *et al.* [44]	2008	6218	13 585	•	•						
Zhong *et al.* [6]	2008	26 824	N/A	•				•			
Hsu *et al.* [22]	2009	1 892 226	5542	•		•			•		
Kaminski *et al.* [36]	2009	N/A‡	—								
Smith and Kapfhammer [176]	2009	6822	110						•		

*Chen *et al.* [11][1] evaluated their heuristics using simulation rather than real data.
†Leitner *et al.* [29][2] minimized the length of a unit test case, not a test suite.
‡Kaminski *et al.* [36][3] applied logical reduction to a set of 19 boolean predicates taken from the avionic software.

DOI: 10.1002/stvr

Table AII. Summary of publications on Regression Test Selection (RTS).

Reference	Year	Max. SUT size (LoC)	Max. Test suite size	Siemens suite	`space`	SIR `Unix` utilities	SIR Java Programs	Other C/ C++/C#	Other Java programs	Models and other languages	Web applications
Fischer [47]	1977	Theory	—								
Fischer *et al.* [48]	1981	Theory	—								
Yau and Kishimoto [53]	1987	Theory	—								
Benedusi *et al.* [61]	1988	Theory	—								
Harrold and Soffa [50]	1988	Theory	—								
Harrold and Soffa [74]	1989	Theory	—								
Harrold and Soffa [51]	1989	Theory	—								
Hartmann and Robson [71]	1989	Theory	—								
Leung and White [10]	1989	Theory	—								
Taha *et al.* [52]	1989	Theory	—								
Hartmann and Robson [73]	1990	Theory	—								
Hartmann and Robson [72]	1990	Theory	—								
Lee and He [70]	1990	Theory	—								
Leung and White [63]	1990	Theory	—								
Leung and White [64]	1990	550	235							•	
Horgan and London [15]	1991	1000	26					•			
Gupta *et al.* [49]	1992	Theory	—								
Laski and Zermer [67]	1992	Theory	—								
White and Leung [65]	1992	Theory	—								
Agrawal *et al.* [54]	1993	Theory	—								
Bates Horwitz [60]	1993	Theory	—								
Rothermel and Harrold [46]	1993	Theory	—								
White *et al.* [66]	1993	Theory	—								
Chen *et al.* [62]	1994	11 000	39					•			
Rothermel and Harrold [56]	1994	Theory	—								
Rothermel and Harrold [45]	1994	Theory	—								
Binkley [96]	1995	Theory	—								
Kung *et al.* [98]	1995	N/A*	N/A								
Rothermel and Harrold [1]	1996	Survey	—								
Rothermel [55]	1996	516	5542	•				•			

DOI: 10.1002/stvr

Table AII. *Continued.*

Reference	Year	Max. SUT size (LoC)	Max. Test suite size	Siemens suite	space	SIR Unix utilities	SIR Java Programs	Other C/ C++/C#	Other Java programs	Models and other languages	Web applications
Binkley [97]	1997	623	—†					•			
Rosenblum and Rothermel [5]	1997	49 316	1033	•				•			
Rosenblum and Weyuker [169]	1997	N/A‡	N/A								
Rothermel and Harrold [57]	1997	516	5542	•							
Rothermel and Harrold [78]	1997	512	5542	•							
Wong *et al.* [75]	1997	6218	1000		•						
Vokolos and Frankl [58]	1997	N/A§	N/A					•			
Ball [80]	1998	Theory	—								
Graves *et al.* [7]	1998	516	398	•							
Rothermel and Harrold [79]	1998	516	5542	•							
Vokolos and Frankl [59]	1998	6218	100		•						
Harrold [114]	1999	Theory	—								
Kim *et al.* [166]	2000	6218	4361	•	•						
Le Traon *et al.* [110]	2000	N/A ¶	N/A								
Rothermel *et al.* [81]	2000	24 849	317					•			
Beydeda and Gruhn [85]	2001	Theory	—								
Bible *et al.* [4]	2001	49 316	1033	•	•			•			
Harrold *et al.* [82]	2001	N/A	189				•		•		
Harrold *et al.* [173]	2001	516	19	•							
Jones and Harrold [124]	2001	6218	4712	•	•						
Orso *et al.* [89]	2001	6035	138				•				
Briand *et al.* [68]	2002	N/A‖	596							•	
Chen *et al.* [88]	2002	N/A	306						•		
Fisher II *et al.* [76]	2002	N/A**	493							•	
Malishevsky *et al.* [119]	2002	65 632	1168			•					
Rothermel *et al.* [120]	2002	68 782	1985			•		•			
Elbaum *et al.* [164]	2003	65 632	1168			•					
Wu and Offutt [111]	2003	Theory	—								
White *et al.* [101]	2003	N/A††	N/A							•	
Deng *et al.* [107]	2004	Theory	—								
Rothermel *et al.* [165]	2004	68 782	1985			•		•			

DOI: 10.1002/stvr

Table AII. *Continued.*

Reference	Year	Max. SUT size (LoC)	Max. Test suite size	Siemens suite	`space`	SIR `Unix` utilities	SIR Java Programs	Other C/C++/C#	Other Java programs	Models and other languages	Web applications
Orso *et al.* [86]	2004	532 000	707						•		
White and Robinson [99]	2004	N/A[††]	N/A						•		
Kim *et al.* [167]	2005	6218	4361	•	•						
Martins and Vieira [87]	2005	902	N/A					•			
Muccini *et al.* [112]	2005	N/A[‖]								•	
Skoglund and Runeson [106]	2005	1 200 000	N/A						•		
Do and Rothermel [177]	2006	80 400	1533				•				
Lin *et al.* [91]	2006	N/A[‡‡]	N/A								
Pilskalns *et al.* [108]	2006	N/A	52							•	
Tarhini *et al.* [94]	2006	Theory	—								
Zhao *et al.* [84]	2006	Theory	—								
Zheng *et al.* [102]	2006	757 000	592					•			
Zheng *et al.* [103]	2006	757 000	592					•			
Muccini *et al.* [113]	2006	N/A[‖]	N/A							•	
Farooq *et al.* [109]	2007	Theory	—								
Orso *et al.* [90]	2007	6035	567				•			•	
Ruth and Tu [92]	2007	Theory	—								
Ruth *et al.* [95]	2007	Theory	—								
Ruth and Tu [93]	2007	Theory	—								
Sherriff *et al.* [147]	2007	Theory	—								
Xu and Rountev [83]	2007	3423	63						•		

DOI: 10.1002/stvr

Table AII. *Continued*.

Reference	Year	Max. SUT size (LoC)	Max. Test suite size	Siemens suite	`space`	SIR `Unix` utilities	SIR Java Programs	Other C/ C++/C#	Other Java programs	Models and other languages	Web applications
Zheng *et al.* [104]	2007	757 000	592					•			
Zheng *et al.* [105]	2007	757 000	31					•			
Fahad and Nadeem [3]	2008	Survey	—								
White *et al.* [100]	2008	Over 1MLoC	N/A					•			
Briand *et al.* [69]	2009	N/A‖	323 614							•	

*Kung *et al.* applied their technique to InterView C++ library, which contains 147 files and over 140 classes [98].

†Binkley [97] applied his technique to five C programs (the largest of which had 623 statements), but only identified the statements that require retesting without considering test suites.

‡Rosenblum and Weyuker [169] evaluated their cost-effectiveness predictor using 31 versions of the `KornShell` and a single version of the `SFIO` (Unix library), but exact versions, sizes of SUT and sizes of test suites were not specified.

§Vokolos and Frankl evaluated their textual difference selection technique using a small C function in addmon family of tools, `power` [58].

¶Le Traon *et al.* [110] presented a case study of a model of packet-switched data transport service, the size of which was not specified.

‖Briand *et al.* studied UML models rather than real systems, the biggest of which contained either 9 classes with 70 methods [68] or 32 classes with 64 methods [69]. Muccini *et al.* studied an RTS technique at software architecture level and presented case studies for the architecture model of an elevator system and a cargo router system [112, 113].

**Fisher II *et al.* [76] evaluated their retesting strategy for spreadsheets with a spreadsheet containing 48 cells, 248 expressions and 100 predicates.

††White *et al.* applied the firewall approach to GUI program with 246 GUI objects [101]. White and Robinson applied the firewall approach to a real time system developed by ABB [99].

‡‡ Lin *et al.* [91] applied their technique to a Java Interclass Graph (JIG) with over 100 nodes.

DOI: 10.1002/stvr

Table AIII. Summary of publications on test case prioritization.

Reference	Year	Max. SUT size (LoC)	Max. Test suite size	Siemens suite	`space`	SIR `Unix` utilities	SIR Java Programs	Other C/ C++/C#	Other Java programs	Models and other languages	Web applications
Harrold [114]	1999	Theory	—								
Rothermel *et al.* [115]	1999	516	19	•							
Elbaum *et al.* [116]	2000	6218	169	•	•						
Elbaum *et al.* [117]	2001	6218	169	•	•						
Elbaum *et al.* [118]	2001	6218	166		•						
Jones *et al.* [124]	2001	6218	4712	•	•						
Rothermel *et al.* [121]	2001	6218	169	•	•						
Elbaum *et al.* [123]	2002	6218	169	•	•						
Kim *et al.* [144]	2002	6218	226	•	•						
Malishevsky *et al.* [119]	2002	65 632	1168			•					
Rothermel *et al.* [120]	2002	68 782	1985			•		•			
Srivastava *et al.* [126]	2002	18 000 000*	3128					•			
Elbaum *et al.* [164]	2003	65 632	1168			•					
Leon *et al.* [136]	2003	N/A†	3333					•	•		
Do *et al.* [127]	2004	80 400	877				•				
Elbaum *et al.* [168]	2004	68 000	1985.32			•		•			
Rothermel *et al.* [165]	2004	68 782	1985			•		•			
Bryce *et al.* [131]	2005	N/A‡	—								
Do *et al.* [162]	2005	80 400	877								
Korel *et al.* [149]	2005	800	980							•	
Rummel *et al.* [156]	2005	N/A§	21						•		
Srikanth *et al.* [139]	2005	2500	50						•		
Bryce *et al.* [132]	2006	N/A‡	—								
Do *et al.* [177]	2006	80 400	1533				•				
Do *et al.* [175]	2006	80 400	877				•				
Do *et al.* [163]	2006	80 400	877				•				
Jeffrey *et al.* [157]	2006	516	N/A	•							
Tonella *et al.* [137]	2006	6218	169		•						

DOI: 10.1002/stvr

Table AIII. *Continued.*

Reference	Year	Max. SUT size (LoC)	Max. Test suite size	Siemens suite	`space`	SIR `Unix` utilities	SIR Java Programs	Other C/C++/C#	Other Java programs	Models and other languages	Web applications
Walcott *et al.* [23]	2006	1808	53						•		
Bryce *et al.* [135]	2007	N/A‡	—								
Fraser *et al.* [154]	2007	N/A§	246								•
Hou *et al.* [152]	2007	5500	183						•		
Korel *et al.* [150]	2007	1416	1000							•	
Li *et al.* [128]	2007	11 148	4350	•	•						
Mirarab *et al.* [145]	2007	124 000	105				•				
Qu *et al.* [133]	2007	17 155	796			•					
Smith *et al.* [26]	2007	1455	N/A						•		
Do *et al.* [158]	2008	80 400	912				•				
Hou *et al.* [159]	2008	N/A¶	1000								
Korel *et al.* [151]	2008	1416	1439	•						•	
Mirarab *et al.* [146]	2008	80 400	912				•				
Qu *et al.* [134]	2008	107 992	975			•					
Sampath *et al.* [153]	2008	9401	890								
Krishnamoorthi *et al.* [148]	2009	6000	N/A						•		
Smith *et al.* [176]	2009	6822	110						•		
Yoo *et al.* [138]	2009	122 169	1061		•	•					
Zhang *et al.* [160]	2009	5361	209						•		

*Srivastava and Thiagarajan [126] considered the biggest software system so far, an office productivity application with over 18 million LoC. However, the technique took the executable binary as input, not the source code. The compiled application was 8.8 Mb in size, with a 22 Mb symbol table.

†Leon *et al.* [136] considered three compilers: `javac`, `jikes` and `gcc`. The sizes of the source code were not specified.

‡Bryce *et al.* [131, 132, 135] studied interaction coverage prioritization, for which the LoC metric is not appropriate.

§These papers considered models rather than real software systems. Rummel *et al.* [156] applied their technique to a model with 3 classes and 21 methods. Fraser *et al.* [154] did not specify the size of the studied models and test suites.

¶Hou *et al.* [159] evaluated their technique using a web application composed of 12 web services.

DOI: 10.1002/stvr

Table AIV. Summary of publications on empirical evaluation and comparative studies.

Reference	Year	Max. SUT size (LoC)	Max. Test suite size	Siemens suite	space	SIR Unix utilities	SIR Java Programs	Other C/ C++/C#	Other Java programs	Models and other languages	Web applications
Leung *et al.* [172]	1991	Theory	—								
Rothermel *et al.* [45]	1994	Theory	—								
Rothermel *et al.* [1]	1996	Survey	—								
Rosenblum *et al.* [5]	1997	49 316	1033	•				•			
Rosenblum *et al.* [169]	1997	N/A*	N/A								
Rothermel *et al.* [78]	1997	512	5542	•							
Graves *et al.* [7]	1998	516	398	•							
Kim *et al.* [166]	2000	6218	4361	•	•						
Bible *et al.* [4]	2001	49 316	1033	•	•			•			
Harrold *et al.* [173]	2001	516	19	•							
Malishevsky *et al.* [119]	2002	65 632	1168			•					
Rothermel *et al.* [120]	2002	68 782	1985			•		•			
Elbaum *et al.* [164]	2003	65 632	1168			•					
Elbaum *et al.* [168]	2004	68 000	1985.32			•		•			
Rothermel *et al.* [165]	2004	68 782	1985			•		•			
Do *et al.* [162]	2005	80 400	877								
Do *et al.* [179]	2005	Theory	—								
Kim *et al.* [167]	2005	6218	4361	•	•						
Do *et al.* [177]	2006	80 400	1533				•				
Do *et al.* [175]	2006	80 400	877				•				
Do *et al.* [163]	2006	80 400	877				•				
Do *et al.* [158]	2008	80 400	912				•				
Do *et al.* [178]	2008	80 400	912				•				
Engström *et al.* [2]	2008	Systematic Review	—								
Fahad *et al.* [3]	2008	Survey	—								
Zhong *et al.* [6]	2008	26 824	N/A	•				•			
Smith *et al.* [176]	2009	6822	110						•		

*Rosenblum *et al.* [169] evaluated their cost-effectiveness predictor using 31 versions of the `KornShell` and a single version of the `SFIO` (Unix library), but exact versions, sizes of SUT and sizes of test suites were not specified.

DOI: 10.1002/stvr

LOC, the detailed information was provided in footnotes. Tables also contain information about the origins of the studied SUT, which are classified as follows:

- *Siemens suite* [41]: All or part of the following set of C programs—`printtokens`, `printtokens2`, `schedule`, `schedule2`, `replace`, `tcas`, `totinfo`, available from SIR [179].
- `space`: An interpreter for ADL, developed by European Space Agency. Available from SIR.
- *Unix utilities in SIR*: All or part of the following set of C programs—`flex`, `grep`, `gzip`, `sed`, `vim`, `bash`, available from SIR.
- *Java programs in SIR*: All or part of the following set of Java programs—`siena`, `ant`, `jmeter`, `jtopas`, `xml-security`, `nanoxml`, available from SIR.
- *Other C/C++/C# programs*: Programs written in C/C++/C# that are not available from SIR.
- *Other Java programs*: Programs written in Java that are not available from SIR.
- *Models and programs in other languages*: models including state machines and UML diagrams. There are also a very few empirical studies that consider programs written in other languages, e.g. Pascal.
- *Web applications*: web applications and web services.

ACKNOWLEDGEMENTS

Shin Yoo is supported by the EPSRC SEBASE project (EP/D050863). Mark Harman is supported by EPSRC Grants EP/D050863, GR/S93684 & GR/T22872, by EU grant IST-33472 (EvoTest) and also by the kind support of DaimlerChrysler Berlin and Vizuri Ltd., London.

REFERENCES

1. Rothermel G, Harrold MJ. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering* 1996; **22**(8):529–551.
2. Emelie EE, Skoglund M, Runeson P. Empirical evaluations of regression test selection techniques: A systematic review. *ESEM '08*: *Proceedings of the Second ACM—IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM: New York, NY, U.S.A., 2008; 22–31.
3. Fahad M, Nadeem A. A survey of UML based regression testing. *Intelligent Information Processing* 2008; **288**:200–210.
4. Bible J, Rothermel G, Rosenblum DS. A comparative study of coarse- and fine-grained safe regression test-selection techniques. *ACM Transactions on Software Engineering and Methodology* 2001; **10**(2):149–183.
5. Rosenblum D, Rothermel G. A comparative study of regression test selection techniques. *Proceedings of the 2nd International Workshop on Empirical Studies of Software Maintenance*. IEEE Computer Society Press: Silver Spring, MD, 1997; 89–94.
6. Zhong H, Zhang L, Mei H. An experimental study of four typical test suite reduction techniques. *Information and Software Technology* 2008; **50**(6):534–546.
7. Graves T, Harrold MJ, Kim JM, Porter A, Rothermel G. An empirical study of regression test selection techniques. *Proceedings of the 20th International Conference on Software Engineering* (*ICSE 1998*). IEEE Computer Society Press: Silver Spring, MD, 1998; 188–197.
8. Rothermel G, Harrold M, Ronne J, Hong C. Empirical studies of test suite reduction. *Software Testing, Verification, and Reliability* 2002; **4**(2):219–249.
9. Garey MR, Johnson DS. *Computers and Intractability*: *A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company: New York, NY, 1979.
10. Leung HKN, White L. Insight into regression testing. *Proceedings of the International Conference on Software Maintenance* (*ICSM 1989*). IEEE Computer Society Press: Silver Spring, MD, 1989; 60–69.
11. Chen TY, Lau MF. Dividing strategies for the optimization of a test suite. *Information Processing Letters* 1996; **60**(3):135–141.
12. Harrold MJ, Gupta R, Soffa ML. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology* 1993; **2**(3):270–285.
13. Horgan J, London S. ATAC: A data flow coverage testing tool for c. *Proceedings of the Symposium on Assessment of Quality Software Development Tools*. IEEE Computer Society Press: Silver Spring, MD, 1992; 2–10.
14. Offutt J, Pan J, Voas J. Procedures for reducing the size of coverage-based test sets. *Proceedings of the 12th International Conference on Testing Computer Software*. ACM Press: New York, 1995; 111–123.

DOI: 10.1002/stvr

15. Horgan JR, London S. Data flow coverage and the C language. *Proceedings of the Symposium on Testing, Analysis, and Verification* (*TAV4*). ACM Press: New York, 1991; 87–97.
16. Papadimitriou CH, Steiglitz K. *Combinatorial Optimization*. Courier Dover Publications: Mineola, NY, 1998.
17. Marré M, Bertolino A. Using spanning sets for coverage testing. *IEEE Transactions on Software Engineering* 2003; **29**(11):974–984.
18. Tallam S, Gupta N. A concept analysis inspired greedy algorithm for test suite minimization. *SIGSOFT Software Engineering Notes* 2006; **31**(1):35–42.
19. Jeffrey D, Gupta N. Test suite reduction with selective redundancy. *Proceedings of the 21st IEEE International Conference on Software Maintenance 2005* (*ICSM'05*). IEEE Computer Society Press: Silver Spring, MD, 2005; 549–558.
20. Jeffrey D, Gupta N. Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Transactions on Software Engineering* 2007; **33**(2):108–123.
21. Black J, Melachrinoudis E, Kaeli D. Bi-criteria models for all-uses test suite reduction. *Proceedings of the 26th International Conference on Software Engineering* (*ICSE 2004*). ACM Press: New York, 2004; 106–115.
22. Hsu HY, Orso A. MINTS: A general framework and tool for supporting test-suite minimization. *Proceedings of the 31st International Conference on Software Engineering* (*ICSE 2009*). IEEE Computer Society: Silver Spring, MD, 2009; 419–429.
23. Walcott KR, Soffa ML, Kapfhammer GM, Roos RS. Time aware test suite prioritization. *Proceedings of the International Symposium on Software Testing and Analysis* (*ISSTA 2006*). ACM Press: New York, 2006; 1–12.
24. Yoo S, Harman M. Pareto efficient multi-objective test case selection. *Proceedings of the International Symposium on Software Testing and Analysis* (*ISSTA 2007*). ACM Press: New York, 2007; 140–150.
25. McMaster S, Memon A. Call-stack coverage for gui test suite reduction. *IEEE Transactions on Software Engineering* 2008; **34**(1):99–115.
26. Smith A, Geiger J, Kapfhammer GM, Soffa ML. Test suite reduction and prioritization with call trees. *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering* (*ASE 2007*). ACM Press: New York, 2007.
27. Harder M, Mellen J, Ernst MD. Improving test suites via operational abstraction. *Proceedings of the 25th International Conference on Software Engineering* (*ICSE 2003*). IEEE Computer Society: Silver Spring, MD, 2003; 60–71.
28. Ernst MD, Cockrell J, Griswold WG, Notkin D. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 2001; **27**(2):99–123.
29. Leitner A, Oriol M, Zeller A, Ciupa I, Meyer B. Efficient unit test case minimization. *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering* (*ASE 2007*). ACM Press: New York, 2007; 417–420.
30. Zeller A. Yesterday, my program worked. Today, it does not why? *SIGSOFT Software Engineering Notes* 1999; **24**(6):253–267.
31. Schroeder PJ, Korel B. Black-box test reduction using input-output analysis. *SIGSOFT Software Engineering Notes* 2000; **25**(5):173–177.
32. Vaysburg B, Tahat LH, Korel B. Dependence analysis in reduction of requirement based test suites. *Proceedings of the International Symposium on Software Testing and Analysis* (*ISSTA 2002*). ACM Press: New York, 2002; 107–111.
33. Korel B, Tahat L, Vaysburg B. Model based regression test reduction using dependence analysis. *Proceedings of the IEEE International Conference on Software Maintenance* (*ICSM 2002*). IEEE Computer Society: Silver Spring, MD, 2002; 214–225.
34. Chen Y, Probert RL, Ural H. Regression test suite reduction using extended dependence analysis. *Proceedings of the 4th International Workshop on Software Quality Assurance* (*SOQUA 2007*). ACM Press: New York, 2007; 62–69.
35. Anido R, Cavalli AR, Lima Jr LP, Yevtushenko N. Test suite minimization for testing in context. *Software Testing, Verification and Reliability* 2003; **13**(3):141–155.
36. Kaminski GK, Ammann P. Using logic criterion feasibility to reduce test set size while guaranteeing fault detection. *Proceedings of the International Conference on Software Testing, Verification, and Validation 2009* (*ICST 2009*). IEEE Computer Society: Silver Spring, MD, 2009; 356–365.
37. Lau MF, Yu YT. An extended fault class hierarchy for specification-based testing. *ACM Transactions on Software Engineering Methodology* 2005; **14**(3):247–276.
38. Rothermel G, Harrold MJ, Ostrin J, Hong C. An empirical study of the effects of minimization on the fault detection capabilities of test suites. *Proceedings of the International Conference on Software Maintenance* (*ICSM 1998*). IEEE Computer Press: Silver Spring, MD, 1998; 34–43.
39. Wong WE, Horgan JR, London S, Mathur AP. Effect of test set minimization on fault detection effectiveness. *Software Practice and Experience* 1998; **28**(4):347–369.
40. Wong WE, Horgan JR, Mathur AP, Pasquini A. Test set size minimization and fault detection effectiveness: A case study in a space application. *The Journal of Systems and Software* 1999; **48**(2):79–89.
41. Hutchins M, Foster H, Goradia T, Ostrand T. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. *Proceedings of the 16th International Conference on Software Engineering* (*ICSE 1994*). IEEE Computer Press: Silver Spring, MD, 1994; 191–200.

DOI: 10.1002/stvr

42. McMaster S, Memon AM. Fault detection probability analysis for coverage-based test suite reduction. *Proceedings of the 21st IEEE International Conference on Software Maintenance* (*ICSM'07*). IEEE Computer Society: Silver Spring, MD, 2007.
43. McMaster S, Memon AM. Call stack coverage for test suite reduction. *Proceedings of the 21st IEEE International Conference on Software Maintenance* (*ICSM'05*). IEEE Computer Society: Washington, DC, U.S.A., 2005; 539–548.
44. Yu Y, Jones JA, Harrold MJ. An empirical study of the effects of test-suite reduction on fault localization. *Proceedings of the International Conference on Software Engineering* (*ICSE 2008*). ACM Press: New York, 2008; 201–210.
45. Rothermel G, Harrold MJ. A framework for evaluating regression test selection techniques. *Proceedings of the 16th International Conference on Software Engineering* (*ICSE 1994*). IEEE Computer Press: Silver Spring, MD, 1994; 201–210.
46. Rothermel G, Harrold MJ. A safe, efficient algorithm for regression test selection. *Proceedings of the International Conference on Software Maintenance* (*ICSM 2003*). IEEE Computer Press: Silver Spring, MD, 1993; 358–367.
47. Fischer K. A test case selection method for the validation of software maintenance modifications. *Proceedings of the International Computer Software and Applications Conference*. IEEE Computer Press: Silver Spring, MD, 1977; 421–426.
48. Fischer K, Raji F, Chruscicki A. A methodology for retesting modified software. *Proceedings of the National Telecommunications Conference*. IEEE Computer Society Press: Silver Spring, MD, 1981; 1–6.
49. Gupta R, Harrold MJ, Soffa ML. An approach to regression testing using slicing. *Proceedings of the International Conference on Software Maintenance* (*ICSM 1992*). IEEE Computer Press: Silver Spring, MD, 1992; 299–308.
50. Harrold MJ, Soffa ML. An incremental approach to unit testing during maintenance. *Proceedings of the International Conference on Software Maintenance* (*ICSM 1998*). IEEE Computer Press: Silver Spring, MD, 1988; 362–367.
51. Harrold MJ, Soffa ML. Interprocedual data flow testing. *Proceedings of the 3rd ACM SIGSOFT Symposium on Software Testing*, *Analysis*, *and Verification* (*TAV3*). ACM Press: New York, 1989; 158–167.
52. Taha AB, Thebaut SM, Liu SS. An approach to software fault localization and revalidation based on incremental data flow analysis. *Proceedings of the International Computer Software and Applications Conference* (*COMPSAC 1989*). IEEE Computer Press: Silver Spring, MD, 1989; 527–534.
53. Yau SS, Kishimoto Z. A method for revalidating modified programs in the maintenance phase. *Proceedings of the International Computer Software and Applications Conference* (*COMPSAC 1987*). IEEE Computer Press: Silver Spring, MD, 1987; 272–277.
54. Agrawal H, Horgan JR, Krauser EW, London SA. Incremental regression testing. *Proceedings of the International Conference on Software Maintenance* (*ICSM 1993*). IEEE Computer Society: Silver Spring, MD, 1993; 348–357.
55. Rothermel G. Efficient, effective regression testing using safe test selection techniques. *PhD Thesis*, University of Clemson, May 1996.
56. Rothermel G, Harrold MJ. Selecting tests and identifying test coverage requirements for modified software. *Proceedings of the International Symposium on Software Testing and Analysis* (*ISSTA 1994*). ACM Press: New York, 1994; 169–184.
57. Rothermel G, Harrold MJ. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology* 1997; **6**(2):173–210.
58. Vokolos F, Frankl P. Pythia: A regression test selection tool based on text differencing. *Proceedings of the International Conference on Reliability Quality and Safety of Software Intensive Systems*. Chapman & Hall, 1997.
59. Vokolos F, Frankl P. Empirical evaluation of the textual differencing regression testing technique. *Proceedings of the IEEE International Conference on Software Maintenance* (*ICSM 1998*). IEEE Computer Press: Silver Spring, MD, 1998; 44–53.
60. Bates S, Horwitz S. Incremental program testing using program dependence graphs. *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press: New York, 1993; 384–396.
61. Benedusi P, Cmitile A, De Carlini U. Post-maintenance testing based on path change analysis. *Proceedings of the International Conference on Software Maintenance* (*ICSM 1988*). IEEE Computer Press: Silver Spring, MD, 1988; 352–361.
62. Chen YF, Rosenblum D, Vo KP. Testtube: A system for selective regression testing. *Proceedings of the 16th International Conference on Software Engineering* (*ICSE 1994*). ACM Press: New York, 1994; 211–220.
63. Leung HKN, White L. Insights into testing and regression testing global variables. *Journal of Software Maintenance* 1990; **2**(4):209–222.
64. Leung HKN, White L. A study of integration testing and software regression at the integration level. *Proceedings of the International Conference on Software Maintenance* (*ICSM 1990*). IEEE Computer Press: Silver Spring, MD, 1990; 290–301.
65. White LJ, Leung HKN. A firewall concept for both control-flow and data-flow in regression integration testing. *Proceedings of the International Conference on Software Maintenance* (*ICSM 1992*). IEEE Computer Press: Silver Spring, MD, 1992; 262–271.

DOI: 10.1002/stvr

66. White LJ, Narayanswamy V, Friedman T, Kirschenbaum M, Piwowarski P, Oha M. Test manager: A regression testing tool. *Proceedings of the International Conference on Software Maintenance* (*ICSM 1993*). IEEE Computer Press: Silver Spring, MD, 338–347.
67. Laski J, Szermer W. Identification of program modifications and its applications in software maintenance. *Proceedings of the International Conference on Software Maintenance* (*ICSM 1992*). IEEE Computer Press: Silver Spring, MD, 1992; 282–290.
68. Briand LC, Labiche Y, Buist K, Soccar G. Automating impact analysis and regression test selection based on UML designs. *Proceedings of the International Conference on Software Maintenance* (*ICSM 2002*). IEEE Computer Society: Silver Spring, MD, 2002; 252–261.
69. Briand LC, Labiche Y, He S. Automating regression test selection based on UML designs. *Journal of Information and Software Technology* 2009; **51**(1):16–30.
70. Lee JAN, He X. A methodology for test selection. *The Journal of Systems and Software* 1990; **13**(3):177–185.
71. Hartmann J, Robson DJ. Revalidation during the software maintenance phase. *Proceedings of the International Conference on Software Maintenance* (*ICSM 1989*). IEEE Computer Press: Silver Spring, MD, 1989; 70–80.
72. Hartmann J, Robson DJ. Retest-development of a selective revalidation prototype environment for use in software maintenance. *Proceedings of the International Conference on System Sciences*, vol. 2. IEEE Computer Press: Silver Spring, MD, 1990; 92–101.
73. Hartmann J, Robson DJ. Techniques for selective revalidation. *IEEE Software* 1990; **7**(1):31–36.
74. Harrold MJ, Soffa ML. An incremental data flow testing tool. *Proceedings of the 6th International Conference on Testing Computer Software* (*ICTCS 1989*). ACM Press: New York, 1989.
75. Wong WE, Horgan JR, London S, Bellcore HA. A study of effective regression testing in practice. *Proceedings of the 8th International Symposium on Software Reliability Engineering* (*ISSRE 1997*). IEEE Computer Society: Silver Spring, MD, 1997; 264–275.
76. Fisher II M, Jin D, Rothermel G, Burnett M. Test reuse in the spreadsheet paradigm. *Proceedings of the International Symposium on Software Reliability Engineering* (*ISSRE 2002*). IEEE Computer Society: Silver Spring, MD, 2002; 257–268.
77. *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Press, 10 December 1990.
78. Rothermel G, Harrold MJ. Experience with regression test selection. *Empirical Software Engineering*: *An International Journal* 1997; **2**(2):178–188.
79. Rothermel G, Harrold MJ. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering* 1998; **24**(6):401–419.
80. Ball T. On the limit of control flow analysis for regression test selection. *Proceedings of the International Symposium on Software Testing and Analysis* (*ISSTA 1998*). ACM Press: New York, 1998; 134–142.
81. Rothermel G, Harrold MJ, Dedhia J. Regression test selection for C++ software. *Software Testing*, *Verification and Reliability* 2000; **10**(2):77–109.
82. Harrold MJ, Jones JA, Li T, Liang D, Orso A, Pennings M, Sinha S, Spoon S. Regression test selection for Java software. *ACM Conference on Object-oriented Programming*, *Systems*, *Languages*, *and Applications* (*OOPSLA 2001*). ACM Press: New York, 2001; 312–326.
83. Xu G, Rountev A. Regression test selection for AspectJ software. *Proceedings of the 29th International Conference on Software Engineering* (*ICSE 2007*). IEEE Computer Society: Silver Spring, MD, 2007; 65–74.
84. Zhao J, Xie T, Li N. Towards regression test selection for aspect-oriented programs. *Proceedings of the 2nd Workshop on Testing Aspect-oriented Programs* (*WTAOP 2006*). ACM Press: New York, 2006; 21–26.
85. Beydeda S, Gruhn V. Integrating white- and black-box techniques for class-level regression testing. *Proceedings of the 25th IEEE International Computer Software and Applications Conference* (*COMPSAC 2001*). IEEE Computer Society Press: Silver Spring, MD, 2001; 357–362.
86. Orso A, Shi N, Harrold MJ. Scaling regression testing to large software systems. *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*FSE 2004*). ACM Press: New York, 2004; 241–251.
87. Martins E, Vieira VG. Regression test selection for testable classes. *Dependable Computing—EDCC 2005* (*Lecture Notes in Computer Science*, vol. 3463/2005). 2005; 453–470.
88. Chen Y, Probert RL, Sims DP. Specification-based regression test selection with risk analysis. *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research* (*CASCON 2002*). IBM Press, 2002; 1–14.
89. Orso A, Harrold MJ, Rosenblum DS, Rothermel G, Soffa ML, Do H. Using component metadata to support the regression testing of component-based software. *Proceedings of the IEEE International Conference on Software Maintenance* (*ICSM 2001*). IEEE Computer Society Press: Silver Spring, MD, 2001.
90. Orso A, Do H, Rothermel G, Harrold MJ, Rosenblum DS. Using component metadata to regression test component-based software: Research articles. *Software Testing*, *Verification*, *and Reliability* 2007; **17**(2):61–94.
91. Lin F, Ruth M, Tu S. Applying safe regression test selection techniques to java web services. *Proceedings of the International Conference on Next Generation Web Services Practices* (*NWESP 2006*). IEEE Computer Society: Silver Spring, MD, 2006; 133–142.
92. Ruth M, Tu S. A safe regression test selection technique for web services. *Proceedings of the 2nd International Conference on Internet and Web Applications and Services* (*ICIW 2007*). IEEE Computer Press: Silver Spring, MD, 2007; 47–47.

DOI: 10.1002/stvr

93. Ruth M, Tu S. Concurrency issues in automating rts for web services. *Proceedings of the IEEE International Conference on Web Services* (*ICWS 2007*). IEEE Computer Press: Silver Spring, MD, 1142–1143.
94. Tarhini A, Fouchal H, Mansour N. Regression testing web services-based applications. *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications* (*AICCSA 2006*). IEEE Computer Press: Silver Spring, MD, 2006; 163–170.
95. Ruth M, Oh S, Loup A, Horton B, Gallet O, Mata M, Tu S. Towards automatic regression test selection for web services. *Proceedings of the 31st International Computer Software and Applications Conference* (*COMPSAC 2007*). IEEE Computer Press: Silver Spring, MD, 2007; 729–736.
96. Binkley D. Reducing the cost of regression testing by semantics guided test case selection. *Proceedings of the International Conference on Software Maintenance* (*ICSM 1995*). IEEE Computer Society: Silver Spring, MD, 1995; 251–260.
97. Binkley
D. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering* 1997; **23**(8): 498–516.
98. Kung DC, Gao J, Hsia P, Lin J, Toyoshima Y. Class firewall, test order, and regression testing of object-oriented programs. *Journal of Object-oriented Programming* 1995; **8**(2):51–65.
99. White L, Robinson B. Industrial real-time regression testing and analysis using firewalls. *Proceedings of the 20th IEEE International Conference on Software Maintenance* (*ICSM 2004*). IEEE Computer Press: Silver Spring, MD, 2004; 18–27.
100. White L, Jaber K, Robinson B, Rajlich V. Extended firewall for regression testing: An experience report. *Journal of Software Maintenance and Evolution* 2008; **20**(6):419–433.
101. White L, Almezen H, Sastry S. Firewall regression testing of gui sequences and their interactions. *Proceedings of the IEEE International Conference on Software Maintenance* (*ICSM 2003*). IEEE Computer Press: Silver Spring, MD, 2003; 398–409.
102. Zheng J, Robinson B, Williams L, Smiley K. A lightweight process for change identification and regression test selection in using cots components. *Proceedings of the 5th International Conference on Commercial-off-the-shelf* (*COTS*)*-based Software Systems* (*ICCBSS 2006*). IEEE Computer Society: Silver Spring, MD, 2006; 137–146. DOI: http://dx.doi.org/10.1109/ICCBSS.2006.1.
103. Zheng J, Robinson B, Williams L, Smiley K. Applying regression test selection for COTS-based applications. *Proceedings of the 28th International Conference on Software Engineering* (*ICSE 2006*). ACM Press: New York, 2006; 512–522.
104. Zheng J, Williams L, Robinson B. Pallino: Automation to support regression test selection for COTS-based applications. *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering* (*ASE 2007*). ACM Press: New York, 2007; 224–233.
105. Zheng J, Williams L, Robinson B, Smiley K. Regression test selection for black-box dynamic link library components. *Proceedings of the 2nd International Workshop on Incorporating COTS Software into Software Systems*: *Tools and Techniques* (*IWICSS 2007*). IEEE Computer Society: Silver Spring, MD, 2007; 9–14.
106. Skoglund M, Runeson P. A case study of the class firewall regression test selection technique on a large scale distributed software system. *Proceedings of the International Symposium on Empirical Software Engineering* (*ISESE 2005*). IEEE Computer Society Press: Silver Spring, MD, 2005; 74–83.
107. Deng D, Sheu PY, Wang T. Model-based testing and maintenance. *Proceedings of the 6th IEEE International Symposium on Multimedia Software Engineering* (*MMSE 2004*). IEEE Computer Society Press: Silver Spring, MD, 2004; 278–285.
108. Pilskalns O, Uyan G, Andrews A. Regression testing uml designs. *Proceedings of the 22nd IEEE International Conference on Software Maintenance* (*ICSM 2006*). IEEE Computer Society Press: Silver Spring, MD, 2006; 254–264.
109. Farooq Qua, Iqbal MZZ, Malik ZI, Nadeem A. An approach for selective state machine based regression testing. *Proceedings of the 3rd International Workshop on Advances in Model-based Testing* (*A-MOST 2007*). ACM: New York, NY, U.S.A., 2007; 44–52.
110. Le Traon Y, Jeron T, Jezequel JM, Morel P. Efficient object-oriented integration and regression testing. *IEEE Transactions on Reliability* 2000; **49**(1):12–25.
111. Wu Y, Offutt J. Maintaining evolving component-based software with UML. *Proceedings of the 7th European Conference on Software Maintenance and Reengineering* (*CSMR 2003*). IEEE Computer Society Press: Silver Spring, MD, 2003; 133–142.
112. Muccini H, Dias M, Richardson DJ. Reasoning about software architecture-based regression testing through a case study. *Proceedings of the 29th International Computer Software and Applications Conference* (*COMPSAC 2005*), vol. 2. IEEE Computer Society: Silver Spring, MD, 2005; 189–195.
113. Muccini H, Dias M, Richardson DJ. Software-architecture based regression testing. *The Journal of Systems and Software* 2006; **79**(10):1379–1396.
114. Harrold MJ. Testing evolving software. *The Journal of Systems and Software* 1999; **47**(2–3):173–181.
115. Rothermel G, Untch RH, Chu C, Harrold MJ. Test case prioritization: An empirical study. *Proceedings of the International Conference on Software Maintenance* (*ICSM 1999*). IEEE Computer Press: Silver Spring, MD, 1999; 179–188.

DOI: 10.1002/stvr

116. Elbaum SG, Malishevsky AG, Rothermel G. Prioritizing test cases for regression testing. *Proceedings of the International Symposium on Software Testing and Analysis* (*ISSTA 2000*). ACM Press: New York, 2000; 102–112.
117. Elbaum S, Gable D, Rothermel G. Understanding and measuring the sources of variation in the prioritization of regression test suites. *Proceedings of the Seventh International Software Metrics Symposium* (*METRICS 2001*). IEEE Computer Press: Silver Spring, MD, 2001; 169–179.
118. Elbaum SG, Malishevsky AG, Rothermel G. Incorporating varying test costs and fault severities into test case prioritization. *Proceedings of the International Conference on Software Engineering* (*ICSE 2001*). ACM Press: New York, 2001; 329–338.
119. Malishevsky A, Rothermel G, Elbaum S. Modeling the cost-benefits tradeoffs for regression testing techniques. *Proceedings of the International Conference on Software Maintenance* (*ICSM 2002*). IEEE Computer Press: Silver Spring, MD, 2002; 230–240.
120. Rothermel G, Elbaum S, Malishevsky A, Kallakuri P, Davia B. The impact of test suite granularity on the cost-effectiveness of regression testing. *Proceedings of the 24th International Conference on Software Engineering* (*ICSE 2002*). ACM Press: New York, 2002; 130–140.
121. Rothermel G, Untch RJ, Chu C. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering* 2001; **27**(10):929–948.
122. Budd TA. Mutation analysis of program test data. *PhD Thesis*, Yale University, New Haven, CT, U.S.A., 1980.
123. Elbaum S, Malishevsky A, Rothermel G. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering* 2002; **28**(2):159–182.
124. Jones JA, Harrold MJ. Test-suite reduction and prioritization for modified condition/decision coverage. *Proceedings of the International Conference on Software Maintenance* (*ICSM 2001*). IEEE Computer Press: Silver Spring, MD, 2001; 92–101.
125. Chilenski J, Miller S. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal* 1994; **9**(5):193–200.
126. Srivastava A, Thiagarajan J. Effectively prioritizing tests in development environment. *Proceedings of the International Symposium on Software Testing and Analysis* (*ISSTA 2002*). ACM Press: New York, 2002; 97–106.
127. Do H, Rothermel G, Kinneer A. Empirical studies of test case prioritization in a junit testing environment. *Proceedings of the 15th International Symposium on Software Reliability Engineering* (*ISSRE 2004*). IEEE Computer Press: Silver Spring, MD, 2004; 113–124.
128. Li Z, Harman M, Hierons RM. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering* 2007; **33**(4):225–237.
129. Bryce RC, Colbourn CJ, Cohen MB. A framework of greedy methods for constructing interaction test suites. *Proceedings of the 27th International Conference on Software Engineering* (*ICSE 2005*). ACM Press: New York, 2005; 146–155.
130. Cohen MB, Dwyer MB, Shi J. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Transactions on Software Engineering* 2008; **34**(5):633–650.
131. Bryce RC, Colbourn CJ. Test prioritization for pairwise interaction coverage. *Proceedings of the ACM Workshop on Advances in Model-based Testing* (*A-MOST 2005*). ACM Press: New York, 2005; 1–7.
132. Bryce RC, Colbourn CJ. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Journal of Information and Software Technology* 2006; **48**(10):960–970.
133. Qu X, Cohen MB, Woolf KM. Combinatorial interaction regression testing: A study of test case generation and prioritization. *Proceedings of the IEEE International Conference on Software Maintenance* (*ICSM 2007*). IEEE Computer Press: Silver Spring, MD, 2007; 255–264.
134. Qu X, Cohen MB, Rothermel G. Configuration-aware regression testing: An empirical study of sampling and prioritization. *Proceedings of the ACM International Symposium on Software Testing and Analysis* (*ISSTA 2008*). ACM Press: New York, 2008; 75–86.
135. Bryce RC, Memon AM. Test suite prioritization by interaction coverage. *Proceedings of the Workshop on Domain Specific Approaches to Software Test Automation* (*DOSTA 2007*). ACM: New York, 2007; 1–7.
136. Leon D, Podgurski A. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. *Proceedings of the IEEE International Symposium on Software Reliability Engineering* (*ISSRE 2003*). IEEE Computer Press: Silver Spring, MD, 2003; 442–456.
137. Tonella P, Avesani P, Susi A. Using the case-based ranking methodology for test case prioritization. *Proceedings of the 22nd International Conference on Software Maintenance* (*ICSM 2006*). IEEE Computer Society: Silver Spring, MD, 2006; 123–133.
138. Yoo S, Harman M, Tonella P, Susi A. Clustering test cases to achieve effective & scalable prioritisation incorporating expert knowledge. *Proceedings of the International Symposium on Software Testing and Analysis* (*ISSTA 2009*). ACM Press: New York, 2009; 201–211.
139. Srikanth H, Williams L, Osborne J. System test case prioritization of new and regression test cases. *Proceedings of the International Symposium on Empirical Software Engineering*. IEEE Computer Society Press: Silver Spring, MD, 2005; 64–73.
140. Freund Y, Schapire R. A short introduction to boosting. *Journal of Japanese Society for Artificial Intelligence* 1999; **14**(5):771–780.

DOI: 10.1002/stvr

141. Freund Y, Iyer R, Schapire RE, Singer Y. An efficient boosting algorithm for combining preferences. *Proceedings of the 15th International Conference on Machine Learning* (*ICML 1998*), Shavlik JW (ed.). Morgan Kaufmann Publishers: Los Altos, CA, 1998; 170–178.
142. Saaty T. *The Analytic Hierarchy Process*, *Planning*, *Piority Setting*, *Resource Allocation*. McGraw-Hill: New York, NY, U.S.A., 1980.
143. Karlsson J, Wohlin C, Regnell B. An evaluation of methods for prioritizing software requirements. *Information and Software Technology* 1998; **39**(14–15):939–947.
144. Kim JM, Porter A. A history-based test prioritization technique for regression testing in resource constrained environments. *Proceedings of the 24th International Conference on Software Engineering* (*ICSE 2002*). ACM Press: New York, 2002; 119–129.
145. Mirarab S, Tahvildari L. A prioritization approach for software test cases based on Bayesian networks. *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering*. Springer: Berlin, 2007; 276–290.
146. Mirarab S, Tahvildari L. An empirical study on Bayesian network-based approach for test case prioritization. *Proceedings of the International Conference on Software Testing*, *Verification and Validation*. IEEE Computer Society: Silver Spring, MD, 2008; 278–287.
147. Sherriff M, Lake M, Williams L. Prioritization of regression tests using singular value decomposition with empirical change records. *Proceedings of the 18th IEEE International Symposium on Software Reliability* (*ISSRE 2007*). IEEE Computer Society: Washington, DC, U.S.A., 2007; 81–90.
148. Krishnamoorthi R, Sahaaya Arul Mary SA. Factor oriented requirement coverage based system test case prioritization of new and regression test cases. *Information and Software Technology* 2009; **51**(4):799–808.
149. Korel B, Tahat L, Harman M. Test prioritization using system models. *Proceedings of the 21st IEEE International Conference on Software Maintenance* (*ICSM 2005*), 2005; 559–568.
150. Korel B, Koutsogiannakis G, Tahat LH. Model-based test prioritization heuristic methods and their evaluation. *Proceedings of the 3rd International Workshop on Advances in Model-based Testing* (*A-MOST 2007*). ACM Press: New York, 2007; 34–43.
151. Korel B, Koutsogiannakis G, Tahat L. Application of system models in regression test suite prioritization. *Proceedings of the IEEE International Conference on Software Maintenance 2008* (*ICSM 2008*). IEEE Computer Society Press: Silver Spring, MD, 2008; 247–256.
152. Hou SS, Zhang L, Xie T, Mei H, Sun JS. Applying interface-contract mutation in regression testing of component-based software. *Proceedings of the 23rd IEEE International Conference on Software Maintenance* (*ICSM 2007*). IEEE Computer Society Press: Silver Spring, MD, 2007; 174–183.
153. Sampath S, Bryce RC, Viswanath G, Kandimalla V, Koru AG. Prioritizing user-session-based test cases for web applications testing. *Proceedings of the 1st International Conference on Software Testing Verification and Validation* (*ICST 2008*). IEEE Computer Society: Silver Spring, MD, 2008; 141–150.
154. Fraser G, Wotawa F. Test-case prioritization with model-checkers. *SE'07*: *Proceedings of the 25th Conference on IASTED International Multi-conference*. ACTA Press: Anaheim, CA, U.S.A., 2007; 267–272.
155. Fraser G, Wotawa F. Property relevant software testing with model-checkers. *SIGSOFT Software Engineering Notes* 2006; **31**(6):1–10.
156. Rummel M, Kapfhammer GM, Thall A. Towards the prioritization of regression test suites with data flow information. *Proceedings of the 20th Symposium on Applied Computing* (*SAC 2005*). ACM Press: New York, 2005.
157. Jeffrey D, Gupta N. Test case prioritization using relevant slices. *Proceedings of the 30th Annual International Computer Software and Applications Conference* (*COMPSAC 2006*). IEEE Computer Society: Washington, DC, U.S.A., 2006; 411–420.
158. Do H, Mirarab SM, Tahvildari L, Rothermel G. An empirical study of the effect of time constraints on the cost-benefits of regression testing. *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM Press: New York, 2008; 71–82.
159. Hou SS, Zhang L, Xie T, Sun JS. Quota-constrained test-case prioritization for regression testing of service-centric systems. *Proceedings of the IEEE International Conference on Software Maintenance* (*ICSM 2008*). IEEE Computer Society Press: Silver Spring, MD, 2008.
160. Zhang L, Hou SS, Guo C, Xie T, Mei H. Time-aware test-case prioritization using Integer Linear Programming. *Proceedings of the International Conference on Software Testing and Analysis* (*ISSTA 2009*). ACM Press: New York, 2009; 212–222.
161. Andrews JH, Briand LC, Labiche Y. *Proceedings of the 27th International Conference on Software Engineering* (*ICSE 2005*). ACM Press: New York, 2005; 402–411.
162. Do H, Rothermel G. A controlled experiment assessing test case prioritization techniques via mutation faults. *Proceedings of the 21st IEEE International Conference on Software Maintenance* (*ICSM 2005*). IEEE Computer Society Press: Silver Spring, MD, 2005; 411–420.
163. Do H, Rothermel G. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering* 2006; **32**(9):733–752.
164. Elbaum S, Kallakuri P, Malishevsky A, Rothermel G, Kanduri S. Understanding the effects of changes on the cost-effectiveness of regression testing techniques. *Software Testing*, *Verification*, *and Reliability* 2003; **13**(2):65–83.

DOI: 10.1002/stvr

165. Rothermel G, Elbaum S, Malishevsky A, Kallakuri P, Qiu X. On test suite composition and cost-effective regression testing. *ACM Transactions on Software Engineering and Methodology* 2004; **13**(3):277–331.
166. Kim JM, Porter A, Rothermel G. An empirical study of regression test application frequency. *Proceedings of the 22nd International Conference on Software Engineering* (*ICSE 2000*). ACM Press: New York, 2000; 126–135.
167. Kim JM, Porter A, Rothermel G. An empirical study of regression test application frequency. *Software Testing*, *Verification*, *and Reliability* 2005; **15**(4):257–279.
168. Elbaum S, Rothermel G, Kanduri S, Malishevsky AG. Selecting a cost-effective test case prioritization technique. *Software Quality Control* 2004; **12**(3):185–210.
169. Rosenblum D, Weyuker E. Using coverage information to predict the cost-effectiveness of regression testing strategies. *IEEE Transactions on Software Engineering* 1997; **23**(3):146–156.
170. Bolsky MI, Korn DG. *The New KornShell Command and Programming Language*. Prentice-Hall PTR: Upper Saddle River, NJ, U.S.A., 1995.
171. Korn D, phong Vo K. SFIO: Safe/Fast String/File IO. *Proceedings of the Summer Usenix Conference 1991*, 1991; 235–256.
172. Leung HKN, White L. A cost model to compare regression test strategies. *Proceedings of the International Conference on Software Maintenance* (*ICSM 1991*). IEEE Computer Press: Silver Spring, MD, 1991; 201–208.
173. Harrold MJ, Rosenblum DS, Rothermel G, Weyuker EJ. Empirical studies of a prediction model for regression test selection. *IEEE Transactions on Software Engineering* 2001; **27**(3):248–263.
174. Ramey C, Fox B. *Bash Reference Manual* (2.2 edn). O'Reilly and Associates: Sebastopol, CA, 1998.
175. Do H, Rothermel G, Kinneer A. Prioritizing junit test cases: An empirical assessment and cost-benefits analysis. *Empirical Software Engineering* 2006; **11**(1):33–70.
176. Smith AM, Kapfhammer GM. An empirical study of incorporating cost into test suite reduction and prioritization. *Proceedings of the 24th Symposium on Applied Computing* (*SAC 2009*). ACM Press: New York, 2009.
177. Do H, Rothermel G. An empirical study of regression testing techniques incorporating context and lifetime factors and improved cost-benefit models. *SIGSOFT '06/FSE-14*: *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM Press: New York, 2006; 141–151.
178. Do H, Rothermel G. Using sensitivity analysis to create simplified economic models for regression testing. *Proceedings*
of the ACM SIGSOFT International Symposium on Software Testing and Analysis (*ISSTA 2008*). ACM Press: New York, 2008; 51–61.
179. Do H, Elbaum SG, Rothermel G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering* 2005; **10**(4):405–435.
180. Memon AM, Soffa ML. Regression testing of GUIs. *ESEC/FSE-11*: *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM: New York, NY, U.S.A., 2003; 118–127.
181. Jia Y, Harman M. Constructing subtle faults using higher order mutation testing. *Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation* (*SCAM 2008*). IEEE Computer Society Press: Silver Spring, MD, 2008; 249–258.
182. Harrold M, Orso A. Retesting software during development and maintenance. *Frontiers of Software Maintenance* (*FoSM 2008*). IEEE Computer Society Press: Silver Spring, MD, 2008; 99–108.
183. McMinn P. Search-based software test data generation: A survey. *Software Testing*, *Verification and Reliability* 2004; **14**(2):105–156.
184. Apiwattanapong T, Santelices R, Chittimalli PK, Orso A, Harrold. MJ, Matrix: Maintenance-oriented testing requirements identifier and examiner. *Proceedings of the Testing*: *Academic and Industrial Conference on Practice And Research Techniques*. IEEE Computer Society: Los Alamitos, CA, U.S.A., 2006; 137–146.
185. Santelices RA, Chittimalli PK, Apiwattanapong T, Orso A, Harrold MJ. Test-suite augmentation for evolving software. *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE: New York, 2008; 218–227.
186. Memon AM. Automatically repairing event sequence-based gui test suites for regression testing. *ACM Transactions on Software Engineering Methodology* 2008; **18**(2):1–36.
187. Alshahwan N, Harman M. Automated session data repair for web application regression testing. *Proceedings of the 2008 International Conference on Software Testing*, *Verification*, *and Validation*. IEEE Computer Society: Los Alamitos, CA, U.S.A., 2008; 298–307.
188. Bertolini C, Peres G, d'Amorim M, Mota A. An empirical evaluation of automated black box testing techniques for crashing guis. *Proceedings of the 2nd International Conference on Software Testing Verification and Validation* (*ICST 2009*). IEEE Computer Press: Silver Spring, MD, 2009; 21–30.
189. Fu C, Grechanik M, Xie Q. Inferring types of references to gui objects in test scripts. *Proceedings of the 2nd International Conference on Software Testing Verification and Validation* (*ICST 2009*). IEEE Computer Press: Silver Spring, MD, 2009; 1–10.

DOI: 10.1002/stvr