SCIENCE CHINA
Information Sciences

• REVIEW •

July 2016, Vol. 59 071101:1–071101:19
doi: 10.1007/s11432-016-5572-2

High-confidence software evolution

Qing GAO[1], Jun LI[1], Yingfei XIONG[1]*, Dan HAO[1], Xusheng XIAO[2], Kunal TANEJA[3], Lu ZHANG[1] & Tao XIE[4]

[1]*Key Laboratory of High Confidence Software Technologies, MoE Institute of Software, School of Electronics Engineering and Computer Science, Peking University, Beijing* 100871*, China;*
[2]*NEC Laboratories America, Inc., Princeton, NJ* 08540*, USA;*
[3]*Accenture Technology Labs, San Jose, CA* 95113*, USA;*
[4]*Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL* 61801*, USA*

Received January 2, 2016; accepted March 29, 2016; published online June 13, 2016

Abstract Software continues to evolve due to changing requirements, platforms and other environmental pressures. Modern software is dependent on frameworks, and if the frameworks evolve, the software has to evolve as well. On the other hand, the software may be changed due to changing requirements. Therefore, in high-confidence software evolution, we must consider both framework evolution and client evolution, each of which may incur faults and reduce software quality. In this article, we present a set of approaches to address some problems in high-confidence software evolution. In particular, to support framework evolution, we propose a history-based matching approach to identify a set of transformation rules between different APIs, and a transformation language to support automatic transformation. To support client evolution for high-confidence software, we propose a path-exploration-based approach to generate tests efficiently by pruning paths irrelevant to changes between versions, several coverage-based approaches to optimize test execution, and approaches to locate faults and fix memory leaks automatically. These approaches facilitate high-confidence software evolution from various aspects.

Keywords software evolution, high confidence, software quality, software development, program analysis

Citation Gao Q, Li J, Xiong Y F, et al. High-confidence software evolution. Sci China Inf Sci, 2016, 59(7): 071101, doi: 10.1007/s11432-016-5572-2

1 Introduction

Software systems continuously evolve and become more complicated, because they have to respond to evolving requirements, platforms, and other environmental pressures [1–3]. Software evolution is the dynamic behavior of programming systems as they are maintained and enhanced over their lifetime [2]. During software evolution, the software is modified due to changed requirements, platforms, etc., and may incur extra faults. Therefore, it is necessary to facilitate high-confidence software evolution.

Modern software is dependent on frameworks. If the frameworks evolve, the software as the client has to evolve as well. On the other hand, the software may be changed due to changing requirements. Therefore, in high-confidence software evolution, we must consider both framework evolution and client

* Corresponding author (email: xiongyf@pku.edu.cn)

https://engine.scichina.com/doi/10.1007/s11432-016-5572-2

© Science China Press and Springer-Verlag Berlin Heidelberg 2016

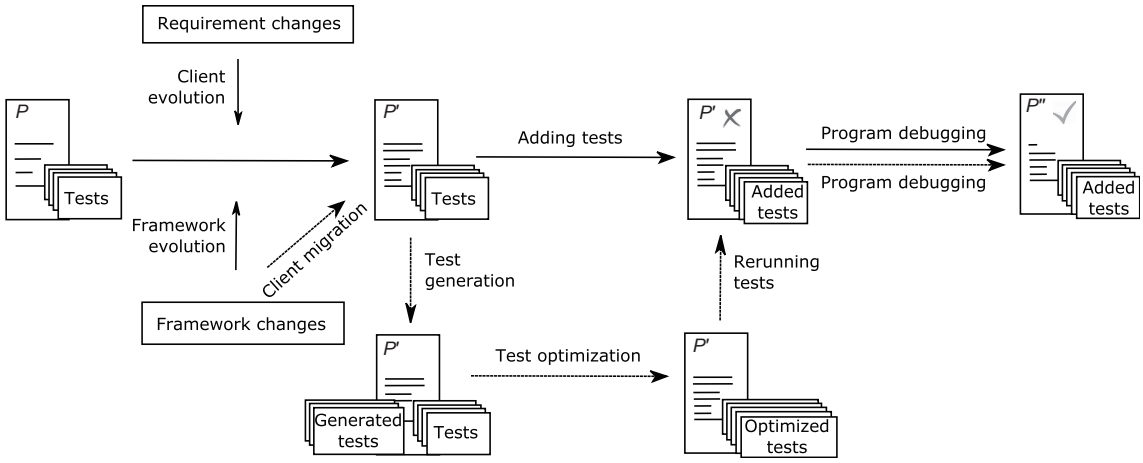


Figure 1 Overview of high-confidence software evolution

evolution. In particular, framework evolution refers to the evolution occurring on the program in the frameworks, whereas client evolution refers to the evolution occurring on the program of the client.

In practice, the API frameworks that a software client is dependent on may often change because these APIs are updated or are replaced by others. For example, due to fault fixing or other requirements, an old API is updated to a new version. Sometimes, developers want to use another API to replace the current one because the platform changes (e.g., from the Android platform to the iOS platform). The former is usually called API update, whereas the latter is called API switching. These API changes may introduce incompatibilities into the client programs, and thus it is important to provide tool support to adapt the client programs between different APIs in framework evolution. Besides framework evolution, client programs may also be modified due to new requirements such as adding new functionalities or conducting performance improvement, classified as client evolution. In either framework evolution or client evolution, the software modifications may incur faults. That is, the software quality may become worse during evolution. Therefore, it is necessary to assure the quality of modified software. In summary, high-confidence software evolution addresses at least two main issues. The first one is tool support for software evolution, and the second one is quality assurance of software evolution.

In the literature, there is a lot of research on software evolution [4–13]. In this article, we present our recent research on high-confidence software evolution, which addresses the preceding two issues. To address the first issue, we present client migration approaches to migrate the old client program to the new APIs with tool support and ensure safety. To address the second issue, we present some testing and debugging approaches to improve the software quality in evolution. In particular, we present a path-exploration-based approach, which generates tests efficiently by pruning paths irrelevant to changes between versions. To optimize test execution, we present an on-demand approach to reduce the number of tests by maintaining their fault-detection capability and several prioritization approaches to schedule the execution order of tests based on their coverage information. To actually improve the software quality, we further present an approach to locate faults by mapping edits to mutants, and a static approach to automatically fix memory leaks through inter-procedural pointer analysis.

Figure 1 shows an overview of high-confidence software evolution. Solid lines show the process of software evolution. A program (denoted as P) may evolve due to framework evolution or client evolution. Developers add tests to the original test suite, and run tests to find whether there is a failure. If a failure is found, they debug the program, modify the program from P' to P'', and rerun the tests, until all the tests pass. The dashed lines in Figure 1 show our research in the corresponding steps for tool support for software evolution or quality assurance of software evolution.

This article makes the following main contributions:

https://engine.scichina.com/doi/10.1007/s11432-016-5572-2

• We identify two scenarios that may compromise software quality, i.e., framework evolution and client evolution, and present an overview of our recent research that helps assure high quality of the software

as the software evolves.

• We show in detail how our approaches can be combined together to help find faults and repair memory leaks introduced in software evolution. The proposed client migration techniques are effective in assuring high quality of framework evolution, and test generation, test optimization, and program debugging techniques are effective in assuring high quality of client evolution.

The rest of the article is organized as follows. Section 2 describes our research on client migration. Section 3 presents our research on regression test generation. Section 4 describes our research on test optimization, including test-suite reduction and test prioritization. Section 5 presents our research on automated debugging, i.e., fault localization and repair. Section 6 presents a demonstrative example and Section 7 concludes the article.

2 Client migration

Modern programs depend on specific frameworks. If a framework evolves, its client programs have to evolve as well. To provide tool support to evolve client programs and ensure the correctness during client migration, we first need to identify the transformation rules between different releases of the framework, and then apply the changes to the client program. To address the first issue, we propose a history-based matching approach, which considers all the revisions between the two releases in the history of the API evolution. To address the second issue, we design a programming language that describes the transformation rules, and automatically applies the changes to client code.

2.1 Transformation rule identification

In practice, it is typical that a software framework evolves independently from its client programs. For various reasons, it is difficult for framework developers to always ensure backward compatibility when evolving frameworks. Thus, automatic framework-evolution identification, which aims to recover the mapping relationships between the APIs of one framework release and those of another framework release, can benefit client developers.

Some approaches (e.g., CatchUP! [7] and MolhadoRef [5]) record change operations performed on the framework with a specialized integrated development environment (IDE) and identify framework evolution by replaying the recorded operations. Since such change operations can accurately characterize framework evolution, these approaches can be very accurate. However, recording and maintaining the operations along the evolution history of the framework could be a serious burden for framework developers. Thus, although some IDEs have already provided mechanisms to record change operations, framework developers are typically reluctant to use these mechanisms. Therefore, it is not very common for a framework to have recorded operations during its evolution history.

Some other approaches try to identify framework evolution via directly matching the two releases of the framework based on some heuristics. Two basic heuristics are text similarity [14] and structure similarity [15]. In particular, given two releases (denoted as R1 and R2), a method in R1 is regarded as being evolved to a method in R2, if the two methods are similar in text or structure. Another effective heuristic is based on comparing the two call graphs of the two releases [16]. The intuition of this heuristic is that, if the set of callers of a method in R1 is very similar to that of a method in R2, there would typically be a mapping relationship between the two methods. Generally speaking, given method A in R1 and method B in R2, if we can define a metric (denoted as M) such that the closeness between M(A) and M(B) indicates the likelihood that A evolves to B, such a metric can be used for identifying the mapping relationships between R1 and R2. As a result, Weissgerber and Diehl [17] propose a series of syntax-level metrics to facilitate framework-evolution identification. Furthermore, researchers have demonstrated that using more than one heuristic simultaneously would typically lead to identifying more useful transformation rules than using just one heuristic. In fact, various recent approaches [18, 19] are based on multiple heuristics. To better balance precision and recall when using multiple heuristics, Wu et al. [11] propose iterative matching to better utilize the strengths and better avoid the weaknesses of

https://engine.scichina.com/doi/10.1007/s11432-016-5572-2

different heuristics. Since the differences between the two releases are typically quite large, it would be intrinsically difficult for these approaches to achieve both high precision and high recall without utilizing the evolution history between the two releases.

We aim to bridge these two categories of approaches. The basic idea of our approach [20] is to consider all the revisions between the two releases in the history of the framework evolution. In particular, we first identify the transformation rules between each pair of adjacent revisions in the evolution history and then aggregate a series of sets of transformation rules to form the final set of transformation rules between the two releases. When identifying the transformation rules between a pair of revisions, we use natural language processing to acquire change information from comments submitted with the second revision as well as several existing heuristics used by previous matching-based approaches. Compared with approaches based on replaying change operations, the set of transformation rules between each pair of adjacent revisions plays a role similar to change operations, and the aggregation process is similar to the replaying process. However, our approach is still based on matching. Thus, it would not bring an extra burden to framework developers. Empirical evidence also demonstrates that our approach can vastly outperform all previous matching-based approaches in both precision and recall.

2.2 Transformation rule presentation

For various reasons, programmers often need to migrate programs between different APIs. For example, when an old API framework is upgraded to a new version and the new version is not backward-compatible, the programmers have to modify the client programs in order to use the new API. As another example, when we need to port a program to a new platform, we have to migrate the program so that it can work with the API on the new platform. It would be beneficial for API vendors to provide automated tools to assist the migration of client programs. When an API vendor upgrades its API, providing tools to automatically migrate the client programs to the new version could prevent the potential loss of users. Furthermore, the API vendors of one platform could provide tools to migrate client programs from other platforms, attracting more users to their platform. For example, Python has provided the 2to3 script to transform (or migrate) programs from Python 2.x to 3.x. Also, RIM has provided a tool to transform Android applications into Blackberry applications. However, despite the many benefits of providing migration tools, it is very difficult to provide such migration tools. As a result, such tools are rarely provided in practice.

2.2.1 *Our approach*

To address such issue, we design a novel programming language to support migrating programs between different APIs [21, 22]. Using the language, the API providers can easily describe the client programs changes resulting from the API upgrade as a transformation program. We develop a tool that accepts client code and the transformation program as input and then outputs new client code automatically that adapts to the new API. The approach reduces the difficulties of providing migration tools by API providers. A key feature of the approach is to support type safety: given any well typed transformation program written in the language, if the client code is well-typed, the transformed code is also well-typed. As a result, we can ensure that the transformation will never introduce any compilation error in the client code. If a migration program itself correctly captures the mapping relations between the old API and the new API, we can ensure that the resulting program is correctly migrated, i.e., being a semantics-preserving transformed program.

It is challenging to ensure type safety in transformation. We can expect two challenges here. First, type is one of the most complex components in programming language design. While transforming client code, even if all the individual code snippets are well typed after transformation, the whole client code may not be well typed. There are complex relations among different types, and the transformation program should deal with these relations carefully. Second, the completeness is hard to ensure. Completeness means the transformation program should cover all the type changes in API upgrade. Even the transformation script published by the Python group, i.e., the 2to3 script, does not deal with completeness well. For

https://engine.scichina.com/doi/10.1007/s11432-016-5572-2

example, client code using method `file()` in Python 2.x cannot be compatible with Python 3.x after it is transformed by the 2to3 script.

To show a flavor of the approach, we use the following simple example client-migration task obtained from a Java SDK upgrade, where class `Hashtable` is replaced by `HashMap`. Given the following piece of code using class `Hashtable`,

```
void processTable(Hashtable t) {
  Enumeration e = t.elements():
  ...
}
```

we would like to replace it with the following piece of code since the class `Hashtable` is replaced by `HashMap`:

```
void processTable(HashMap t) {
  Iterator e = t.values().iterator():
  ...
}
```

To describe this transformation in the language, we need to write the following piece of code using the transformation language:

```
(t : Hashtable ->> HashMap)
[ (t.elements()) -> (t.values().iterator()) ]
```

This piece of code declares two things. First, the type `Hashtable` in the old API corresponds to `HashMap` in the new API. Second, any call to the `elements()` method of `Hashtable` should be converted to a call to `values().iterator()`. Given this piece of code, the approach automatically performs the transformation. As we can see from the preceding piece of code, the transformation code is mainly in plain Java, and on top of plain Java there is only a construct for direct replacement. In this way, it is easy for the programmers to learn this transformation language. Furthermore, an existing study [23] has shown that this simple form is able to handle many client migration cases in practice. This example gives an intuitive sense of the transformation language, and next we present more details.

A transformation program includes a set of transformation rules. Each rule has a form as follows:

$$
\begin{array}{l}
(x_1 : C_1 \hookrightarrow D_1, x_2 : C_2 \hookrightarrow D_2, \cdots, x_n : C_n \hookrightarrow D_n) \\
\ [\\
\qquad JavaExpr_1 \rightarrow JavaExpr_2 \\
\ ]
\end{array}
$$

The first part of a rule is type declarations of meta variables. Each variable is bound by Java expression $JavaExpr_1$, and the type declaration of a variable x_i shows the type change of this variable from Java expression $JavaExpr_1$ to $JavaExpr_2$. Thus, notation C_i denotes the type of variable x_i in $JavaExpr_1$, and D_i denotes the type of variable x_i in $JavaExpr_2$. The semantics of this rule means (1) $JavaExpr_1$ first matches a Java expression in client code in terms of the type of each variable; (2) during expression matching, the meta variables will bind with a sub expression; (3) the matched expression will be replaced by $JavaExpr_2$, and each meta variable will be replaced by a concrete sub expression.

However, the preceding semantics are not sufficient to ensure the type safety. There are four conditions to sufficiently ensure type safety, as described below.

First, for each code snippet introduced by a transformation rule, the code snippet itself should be well-typed. In the preceding rule, this condition requires that $JavaExpr_1$ and $JavaExpr_2$ should be well typed. Since the code snippets will be directly replaced by the code, any type error in the transformation rules will be brought into the code.

https://engine.scichina.com/doi/10.1007/s11432-016-5572-2

Second, each type in the source API should be mapped to at most one type in the target API. Using the preceding rule as an example, if type C_i maps to type D_j and D_k, this condition requires that D_j should be equal to D_k. Such requirement is mainly for the execution of the transformation rules. If one type is mapped to more than one type in the target API, we would not know which type we should replace with.

Third, the mapping between the source types and the target types must preserve the subtyping relation. Let C_i and C_j be two types in the source API and m be the type mapping, $C_i <: C_j \implies m(C_i) <: m(C_j)$. The rationale of this condition can be seen from the following example. Suppose class `JList` is a sub class of class `Container` while class `List` is not a sub class of class `Composite`. Considering the following transformation rules, we have two type mappings from `Container` to `Composite` and from `JList` to `List`:

```
() [ new Container() ->  new Composite(new Shell(), 0) ]
() [ new JList() -> new List() ]
```

then the following client code,

```
Container x = new JList();
```

will be transformed into the code as follows:

```
Composite x = new List();
```

which contains a type error as class `List` is not a sub class of class `Composite`.

Fourth, the transformation rules must cover all type changes between the source API and target API. If a type `C` is not covered, we cannot guarantee type safety if a client program uses `C`. For example, given type `C` being replaced by type `D` in the new API, the client code snippet with type `C` will not be changed if the transformation rules do not cover this type change. Then there will be type errors between the transformed client code and new API.

Our study [22] has shown that these four conditions are sufficient for type safety. Furthermore, as three non-trivial case studies show, the four conditions never exclude useful transformation for client migration.

This approach requires developers to write the transformation program manually, and the approach can automate this process by using the transformation rules identified in Subsection 2.1. The transformation rules already describe the transformed classes and methods. Based on these rules, the approach, combined with existing type inference techniques, can generate a transformation program. However, in this case, there could be false positives, because the identified transformation rules may not be precise enough. Therefore, developers can choose to automate the whole process while tolerating possible false positives, or write a transformation program manually while ensuring type safety. To further reduce false positives, we can use regression testing techniques described in Sections 3 and 4.

2.2.2 *Related work*

Transformation frameworks. In addition to Twinning [23], there are also many other transformation languages. TXL [24] and Stratego [25] are general-purpose and grammar-oriented transformation languages. These two languages accept grammar description, an under-transforming program, and a transformation program as input, and output the transformed program. Like our work, these languages perform the transformations on the abstract syntax tree of a program, but they do not consider specific type systems for its general purpose. These two approaches cannot assure the well-typedness of the transformed program. Like Twinning, Refaster [26] focuses on transformation over Java code. Refaster uses before-and-after examples of Java code to illustrate a refactoring. This work mainly deals with method replacement similar to our work. But Refaster cannot assure the well-typedness of transformed code, as it only requires that each replaced code snippet is well typed. As we have discussed earlier, it cannot assure the well-typedness of the whole program.

https://engine.scichina.com/doi/10.1007/s11432-015-5572-2

Type-safe transformations. Hula [27] is a rule-based update language for Haskell, and it can assure the well-typedness while updating Haskell programs. The type-safe update depends on a core update calculus [28], which acts type-safe transformation on lambda calculus. Like our work, update calculus also sets some constraints on transformation rules, and assures these rules can preserve well-typedness of transformed programs of lambda calculus. This work allows one type to be transformed to only a more generic type. Thus, some useful type changes, such as `Vector` to `ArrayList`, cannot be described using these approaches.

Semantics-preserving transformations. Reba [29] treats the transformation as a set of refactorings. When API upgrades, the update operations are recorded by tools as refactorings. These operations can be replayed on client code and preserve its semantics. This approach limits transformation as refactoring, which is not sufficient in real API update. Moreover, this approach cannot support API switching. Leather et al. [30] provide an approach to preserve the semantics of a program. The transformation rules of this work allow a type to be replaced by only its isomorphic type, being limited in program transformation. This work mainly focuses on transformation over lambda calculus, avoiding the problems introduced by object orientation, such as subtyping.

3 Regression test generation

Besides framework evolution that causes client programs to evolve, client programs by themselves continue to evolve throughout their lifetime undergoing various kinds of changes. These changes can be for the purpose of introducing new features, refactoring, or fault fixing. The changes made to a program can interact with other parts of the program in an unexpected way, resulting in regression faults. Thus, to assure high quality in software evolution, it is highly desirable to detect regression faults as soon as possible to reduce the cost in fixing them.

One effective way to detect regression faults is to adopt continuous testing. Continuous testing [31] tests a program continuously as soon as a developer makes changes to the program. In particular, continuous testing executes an existing regression test suite as soon as a developer makes changes to the program (and the changes are compilable). The failing tests represent the behavioral differences [1)] between the new and the old program versions. The developer can then inspect the behavioral differences to quickly detect and fix the regression faults that are introduced by the changes. Hence, continuous testing reduces the cost of fixing regression faults by detecting them as soon as the changes are saved.

However, the effectiveness of continuous testing is limited by the effectiveness of the existing regression test suite. According to the PIE model [32] of error propagation, a fault can be detected by a test suite only if all the following three conditions are met:

- The test suite executes (E) the faulty statement.
- The program state is infected (I) by the execution of the faulty statement.
- The state infection is propagated (P) to an observable output.

Even if the existing test suite achieves 100% structural code coverage such as branch coverage, it may not be able to detect regression faults as it might not satisfy conditions I and P of the PIE model.

State-of-the-art test generation techniques [33–42] can be used to generate tests on the new program version to automatically augment the existing regression test suite to increase the effectiveness of the test suite in terms of detecting behavioral differences. However, existing test generation techniques (such as random test generation [34,36], search-based test generation [40,43], and path-exploration-based test generation [33,35,37–39,41,42] techniques) do not specifically aim at finding behavioral differences between the new and the old program versions. In particular, the tools aim at covering all branches in the program (i.e., satisfying condition E of the PIE model) and not at specifically finding behavioral differences between the old and the new program versions. Hence, these approaches are ineffective and

1) A behavior difference between the old and the new program versions can be reflected by the difference in observable outputs of the two versions. A behavior difference does not necessarily signify a regression fault as the behavioral difference might be intended by the developer.

https://engine.scichina.com/doi/10.1007/s11432-015-5572-4

inefficient in finding behavioral differences, even with increasing computing power thanks to multi-core architectures and cloud computing.

3.1 Our approach

To address the preceding issue, we propose a path-exploration-based test generation (PBTG) approach, called eXpress [44, 45]. PBTG approaches are effective in generating a test suite that achieves high structural coverage. To achieve high structure coverage, PBTG approaches try to explore all feasible paths in the program. For real-world programs that have an exponential number of paths, such path exploration is quite expensive. However, if our aim is to find behavioral differences between the new and the old program versions, we do not need to explore all the paths. In particular, we do not need to explore the paths whose execution cannot detect any behavioral difference between the two program versions (referred to as irrelevant paths). eXpress prunes out all the irrelevant paths from the search space of a PBTG approach. Hence, eXpress makes the path exploration directed towards finding behavioral differences.

eXpress includes a novel practical application of the PIE model of error propagation. The key insight of this work is that the execution of many paths in the program guarantees not to satisfy any of the conditions E, I, or P of the PIE model. These paths can be pruned from the search strategy of a PBST approach so that its efforts are directed towards finding behavioral differences. eXpress first determines a set of paths ($P_{\not E}$) that cannot lead to any changed region in the program and a set of paths ($P_{\not P}$) that cannot propagate a state infection to an observable output. eXpress prunes the set of paths $P_{\not E} \bigcup P_{\not P}$ from the search space of the PBTG approach. In addition, eXpress prunes a set of paths ($P_{\not I}$) that it determines during path exploration by a PBTG approach. The paths in the set ($P_{\not I}$) are guaranteed not to infect the program state after the execution of a changed region.

We have implemented eXpress as a search strategy for dynamic symbolic execution (DSE) [35, 37], a state-of-the-art PBTG approach. In particular, the implementation guides DSE to avoid flipping branching nodes whose unexplored side is guaranteed to lead to an irrelevant path (i.e., not satisfying any of the conditions E, I, or P of the PIE model). To make the path exploration even faster, eXpress includes a technique that can exploit the existing test suite. In particular, eXpress seeds the tests in the test suite to the path exploration to efficiently cover various changed code regions that are not covered by the existing test suite. As a result, behavioral differences are shown to be found earlier in the path exploration.

3.2 Related work and challenges

Regression test generation based on symbolic execution. Although symbolic execution is effective in systematically exploring program paths, it suffers from scalability issues such as path explosion. To address the scalability issues, recent work explores the directions of exploiting existing test suites or input partitions to reduce the search space. Marinescu et al. [46] propose an approach that leverages existing test suites as seed inputs to reach certain program parts that are difficult to reach by applying symbolic execution from scratch, and explores additional paths using lightweight symbolic execution to detect regression errors. Böhme et al. [47] propose an approach that divides the common input space between two program versions into different input partitions that either guarantee behavior-equivalence, or expose differences for a subset of inputs. Rather than generating tests for the whole input space, their approach explores differential partitions in a gradual manner and generates representative tests gradually for each partition. To further improve the scalability, selective symbolic execution [48, 49] can be used to limit symbolic execution on critical parts of the changed programs.

Besides the scalability issues, symbolic execution also suffers from other issues [50–53], such as dealing with complex objects, external environments, and loops. Although there exist specialized tools for generating complex objects [51, 54, 55], automatically mocking environments [56–58], and dealing with loops [59, 60], these tools cannot address all the issues encountered in complex software and are not

https://engine.scichina.com/doi/10.1007/s11432-016-5572-2

optimized for regression test generation. Thus, there is also a strong need in addressing these issues for regression test generation.

Change-interaction errors. As a type of regression errors, change-interaction errors happen when multiple changes introduced in a program interact in unexpected ways. Santelices et al. [9] propose an approach that formally models change interactions and uses dynamic slicing to detect change interactions based on the models. Böhme et al. [4, 61] identify and formalize change-interaction errors in evolving software, and propose a regression-test generation approach that uses symbolic execution to generate tests for exposing change-interaction errors via the summarized control-flow and dependencies across changes. Essentially, given a sequence of code changes C, there are $2^{|C|}$ program configurations to be analyzed. These approaches explore only the approximation of these configurations, and there is a need for a more precise and more scalable approach that can expose the change-interaction errors for the whole space of the program configurations.

4 Test optimization

In software evolution, programs are evolved from one version to another in the continuous development. To ensure the quality of the program during software evolution, developers usually reuse the tests for an early version while testing its latter version. As more and more tests for the previous version are added to test the current version, the number of tests grows rapidly. Thus, it is not practical to reuse all the tests collected in software evolution as it is costly to run such a large number of tests. It is necessary to reduce the number of tests so as to reduce the cost on running tests in software evolution. To address this problem, various approaches have been proposed to reduce the number of tests in software evolution, especially regression testing, presented in Subsection 4.1. On the other hand, due to the cost concern (including time and effort), sometimes it is impossible to reuse all these tests. The process of testing on the current version may be stopped at any time. To maximize the fault-detection effectiveness of existing selected tests considering such concerns, various approaches have been proposed to prioritize tests, presented in Subsection 4.2.

4.1 Test-suite reduction

In the literature, most test-suite reduction techniques aim to select a minimal representative subset of tests from a given test suite guaranteeing the same testing requirements, which are mostly code coverage in software evolution. Typically, these techniques use the code coverage criterion as test requirements. In particular, Jones and Harrold [62] use the modified condition/decision coverage as a criterion, while some researchers [63, 64] use multiple coverage (e.g., branch coverage and definition-use pair coverage) as a criterion. Using these code coverage criteria, researchers propose various techniques to select tests satisfying the same code coverage criterion as the original test suite. In particular, Harrold et al. [6] propose a heuristic to select tests by removing redundant and obsolete tests, Chen and Lau [65] propose another heuristic combined with a greedy strategy, and Yoo and Harman [12] transform the problem of test-suite minimization into a multi-objective optimization problem.

Existing test-suite reduction techniques produce a representative subset of tests with the same code coverage, but test-suite reduction may cause loss in fault-detection ability. Sometimes, the loss is dramatically huge. For example, according to our previous work [66], although the average loss in fault-detection ability of the "replace" program resulting from test-suite reduction is 57.0%, the actual loss in fault-detection ability ranges from 0% to 91.3%. The loss in fault-detection ability for the existing test-suite reduction techniques may vary dramatically when they are applied in practice.

To control the loss in fault-detection ability resulting from test-suite reduction, we propose an on-demand test-suite reduction approach [66], which is designed to control the loss in fault-detection resulting from test-suite reduction for any given confidence level $c\%$. That is, this approach first identifies the relation between the loss in fault-detection ability and the code coverage of a test suite for any given $c\%$, and then reduces a test suite based on the preceding relation. In particular, this approach first

https://engine.scichina.com/doi/10.1007/s11432-016-5572-2

identifies the quantitative relation between the loss in fault-detection ability and the code coverage by a controlled experimental study. Formally, each cell in the fault-detection-loss table can be represented by a variable $V_c(i_1, i_2)$, which shows the loss in fault-detection capability in at least $c\%$ circumstances when the coverage of a statement changes from i_1 to i_2.

Based on the fault-detection-loss table, we formulate the problem of on-demand test-suite reduction as an integer linear programming problem (i.e., ILP problem), which targets minimizing the number of tests by guaranteeing either global constraints or local constraints [66]. In particular, the global constraints ensure that for each statement in the program at the confidence level $c\%$ there exists at most $l\%$ loss in the fault-detection capability, whereas the local constraints ensure that for the program under test at the confidence level $c\%$ the average loss in fault-detection capability is at most $l\%$. By solving these constraints, we produce a subset of tests whose total fault-detection loss is at most $l\%$ in at least $c\%$ circumstances.

Compared with existing test-suite reduction techniques, the on-demand test-suite reduction approach can guarantee at most a certain level of loss in fault-detection capability in test-suite reduction by selecting a reasonable number of tests. In particular, compared with the typical test-suite reduction approach proposed by Harrold et al. [6], the on-demand approach selects at most 60% more tests but guarantees at most a certain level of loss in fault-detection capability whereas the former approach does not.

To further improve the efficiency of our approach, we can combine test reduction techniques with regression test generation techniques described in Section 3. In software evolution, existing tests that do not cover new or modified statements tend to have lower fault-detection capability than the newly generated tests that aim to cover new branches. Therefore, we can always keep the tests generated via regression test generation techniques, and reduce the tests that do not cover new or modified statements. For the rest of the tests, which exist in the original test suite and also may cover new statements, we can then apply the on-demand test suite reduction. In this way we can improve the efficiency of our approach.

4.2 Test prioritization

Test prioritization is firstly proposed in regression testing, aiming to schedule the execution order of tests so that faults may be revealed early. In the literature, many test prioritization techniques [67–70] have been proposed to schedule the execution order of tests based on some structural coverage (e.g., statement coverage, branch coverage, modified condition/decision coverage [62]). Based on these coverage criteria, various prioritization algorithms have been proposed. For example, Jiang et al. [71] propose an adaptive random approach, which selects tests based on the distance between selected tests and remaining unselected tests. Li et al. [72] view test prioritization as a searching problem and thus propose a genetic programming based approach to prioritize tests. Zhang et al. [73] propose to use integer linear programming to represent and solve time-aware test prioritization. Besides these approaches, two typical greedy strategies (i.e., the total and additional strategies) are widely studied and are evaluated to be very effective even compared with some newly proposed test prioritization approaches.

Considering the benefit of the two greedy strategies, we propose a unified approach that combines these two strategies by defining a basic model and an extended model [74, 75]. In particular, this approach defines the probability that each unit u_j (e.g., statement or method) contains undetected faults by $Prob[j]$ and calculates the priority of each test by the sum of $Prob[j]$ if the test covers the unit u_j. Furthermore, during test prioritization, as selected tests may have revealed some faults, it is necessary to modify $Prob[j]$ if the faults in the unit u_j may have been detected by existing selected tests. In particular, this unified approach presents two models (i.e., the basic model and the extended model) to define how to modify $Prob[j]$ after selecting a new test. In the basic model, this approach uses p to represent the probability that any test t can detect faults in any unit u and modifies $Prob[j]$ based on p. In the extended model, this approach not only modifies $Prob[j]$ based on p, but also considers how many times the unit is covered by existing tests. Based on the priority of each test, this approach selects the test with the largest priority

https://engine.scichina.com/doi/10.1007/s11432-016-5572-2

and modifies the priority of each test at the same time, until all the tests are scheduled. Based on the parameter p and the two models, this approach may yield a series of generic strategies ranging from the total to the additional strategies. In other words, the total and additional strategies can be viewed as two instances of this approach. Furthermore, according to our empirical study, this approach using either the basic or the extended model with uniform p values can significantly outperform the total strategy, and sometimes outperform the additional strategy.

On the other hand, we present an adaptive test prioritization approach [76], which selects tests and runs the selected tests simultaneously so that it is possible to apply the precise execution information of the selected tests on the current program to improve further test prioritization. In particular, this approach first calculates the fault-detection effectiveness of all tests based on their execution information on the previous program and then selects the test with the largest fault-detection capability. In the subsequent test prioritization process, this approach modifies the fault-detection capability of unselected tests based on the output of the latest selected test, and selects the remaining tests with the largest fault-detection capability until all the tests are selected. According to our empirical study, the adaptive test prioritization approach performs better than the additional strategy in some subjects.

Furthermore, as it is costly to collect the execution information of programs, we propose to use the static call graph of a test to estimate its dynamic coverage [77]. In particular, this static approach targets tests written in the JUnit testing framework and views the sequence of methods called by a test as the method coverage of the test. In other words, based on the static call graph of a test, this approach learns which methods are called by this test and constructs the method coverage (as well as the statement coverage). Based on static method coverage (or static statement coverage), this approach schedules the execution order of these tests. Compared with existing dynamic approaches, which schedule the execution order of tests based on their dynamic coverage, this static approach is less costly because it does not require the execution of instrumented programs. Furthermore, according to the empirical results, this static approach is competitive in prioritizing tests even when compared with the dynamic approaches.

Similar to test reduction, tests that cover new or modified statements may have better fault-detection capability in software evolution, especially for the generated tests. Therefore, we give higher priority to these tests in regression testing. For the rest of the tests, we further apply the previously described test prioritization techniques.

5 Debugging

Test generation and test optimization techniques aim to find whether there are failures. Program debugging, on the other hand, aims to localize faults and repair them. When software evolves, we seek to use evolution information for automated program debugging. To localize faults as software evolves, we utilize mutants and define a new strategy to achieve more accurate fault localization, presented in Subsection 5.1. In software repair, we propose a static approach for automatically repairing difficult-to-fix memory leaks by using regression tests during evolution, presented in Subsection 5.2.

5.1 Fault localization

It is typical that before a fault is fixed, the faulty statement(s) should be identified in the first place. In fact, the results of fault localization can either provide developers with some clues for fault fixing or serve as an important input for an automatic algorithm of fault fixing. As a result, there have been various research efforts in the past decade. In general, there are two main lines of research on this topic.

The first line of research considers the situation that there exists only a faulty program. Approaches in this line (e.g., TARANTULA [78] and SAFL [79]) typically compare the coverage information of passing tests with the coverage information of failing tests. In particular, based on the coverage information, such an approach calculates the suspiciousness score of each covered statement in order to rank the statements. Typical formulae for calculating the suspiciousness score include TARANTULA [78], SBI [80], Ochiai [81], and Jaccard [82]. Besides using the basic coverage information, some other approaches further use some

https://engine.scichina.com/doi/10.1007/s11432-016-5572-2

additional information. One way to acquire additional information is to mutate the program to obtain the execution information of the mutants [83]; another way is to manipulate program executions to obtain the contrived coverage information (e.g., predicate switching [84], value replacement [85], and object replacement [86]). Xuan and Monperrus [87] propose spectrum-driven test purification. This approach separates existing tests into small fractions, and enhances test oracles for fault localization. Although most of these approaches are applicable for localizing faults introduced in software evolution, this line of research typically does not explicitly consider software evolution.

Unlike the first line of research, the second line of research is specific to software under evolution. In this line of research, there are two main categories of approaches. Approaches in the first category rely on executing a set of intermediate versions, each of which is composed of the program before evolution and a subset of change edits performed during the evolution. A typical approach in this category is delta debugging (DD) proposed by Zeller [88]. The DD approach uses a binary-search-like process to search in the space of the intermediate versions. As some intermediate versions cannot be compiled, exact binary search is infeasible for DD. Naturally, DD can also be used to identify which parts in the input trigger the failure [89]. Approaches in the second category rely on change impact analysis. Stoerzer et al. [10] use classification to identify a subset of changes that are more likely to be related to the failure. Ren and Ryder [8] use some heuristics to further rank the changes in the identified subset. Zhang et al. [90] and Alves et al. [91] further adapt heuristics in the first line of research (e.g., TARANTULA [78] and SBI [80]) to rank fault-inducing changes. In our work, we utilize mutants to define a new strategy for ranking fault-inducing changes [92]. The basic idea is to map change edits to mutants, which can serve as a means to predict whether a heuristic such as TARANTULA or SBI is likely to achieve good results. In other words, mutation changes made in mutants from the older version of the program can simulate the real edits made by developers. Therefore, if we use mutation changes that can simulate the real edits, and calculate suspiciousness scores by combining both mutation changes and real edits, we can find the fault location more accurately.

Suppose that developers edit a method `f()` and introduce a new fault. The fault is exposed by a method call to `f()`. Because it may happen that some correct statements are executed more in the failing tests, the suspiciousness score of the method call to `f()` may not be ranked the highest. Using our approach, in the original program, we can mutate statements in the method `f()` to indicate there is a change in the implementation of method `f()`, and calculate the suspiciousness score using executions from both the real edits and the obtained mutants. Therefore, we can reduce the impact from other statements, and it is more likely to rank the failure-inducing change highest.

In the previous step of test optimization, we provide techniques to reduce and schedule tests. If a test is executed earlier, we consider it to be more likely to discover a regression fault. In fault localization, we also provide different weights to the same statement covered by different tests. If a test is executed earlier, then its covered statement has a higher weight.

5.2 Repair

There are two main lines of research on automatic program repair. The first line of research considers general faults. Existing approaches mainly generate patches without evolution information [93–96]. Nguyen et al. [97] propose an approach to fix recurring faults, some of which have similar patterns between different versions during software evolution. The second line is specific to certain types of faults, and approaches are specifically designed to fix these faults [98–100].

In software evolution, although many faults can be found through regression testing, some of them, e.g., memory leaks, are hard to find via testing approaches [101, 102]. There has been lots of research on memory-leak detection [103–109]. However, all of them may introduce false positives.

Our work [110] aims to fix memory leaks automatically, and guarantee the correctness of the fix. We require correctness because, if a fix may be wrong, programmers still need to examine all of the generated patches by understanding the program and finding the correct fix position. Dynamic approaches need to run the program, and it is difficult and time-consuming to explore all program paths to ensure fix

https://engine.scichina.com/doi/10.1007/s11432-015-5572-2

correctness. Therefore, we explore static approaches to automatically detect and fix memory leaks.

The definition for a correct fix is as follows. A correct fix for a memory leak is an insertion of a deallocation statement s into a program, and satisfies the following three conditions:

(1) A memory chunk c is allocated before s, and s releases c.

(2) There is no other deallocation statement after the inserted statement.

(3) There is no use of c after the execution of s.

To satisfy the three conditions, we design an algorithm based on the control flow graph (CFG). A control flow graph is a graph on which nodes represent statements and edges represent jumps between statements.

First, in order to ensure that the inserted deallocation statement frees the corresponding memory chunk, we perform pointer analysis [111] to identify the references of each memory chunk. There have been various research efforts on pointer analysis [111–114], and therefore we utilize existing pointer analysis algorithms, rather than designing one by ourselves. Our approach requires pointer analysis to provide inter-procedural and complete results, i.e., each pointer points to all possible memory locations. Flow-/context-/field-sensitivities can provide more accurate results. The pointer analysis algorithm used in our implementation is DSA [111], which is inter-procedural, flow-insensitive, context-sensitive with heap cloning, field-sensitive, unification-based and SSA-based.

Second, to perform inter-procedural analysis, we need to either construct a super CFG of all procedures, or build procedure summaries. The overhead of the former is too high and not feasible in practice. Therefore we build summaries for each procedure. The summaries indicate heap memory usage in the points-to graph of each procedure, i.e., whether a memory location represented as a node in the points-to graph is allocated, used, freed, or escaped. Then we use each procedure summary to identify CFG nodes whose corresponding statements allocate, use, or free a certain memory chunk.

Third, based on the identified CFG nodes, we perform four passes of intra-procedural data flow analysis, using the monotone framework [115]. The data at each program point is a set of CFG nodes, whose corresponding statements are allocations. The four passes of data flow analysis are as follows:

(1) A forward data flow analysis to ensure that there are no deallocation statements before the inserted deallocation statement.

In this step, we denote data at each CFG node n as Df_n, which is a set of allocations that may have been deallocated before or at the current CFG node. The meet operator is set union, and the transfer function at node n is $f1_n(Df)$:

$$f1_n(Df) = \begin{cases} Df \cup \{m_1, \ldots, m_k\}, & n \text{ labelled as } \texttt{Dealloc}(m_1, \ldots, m_k), \\ Df, & \text{otherwise.} \end{cases}$$

(2) A backward data flow analysis to ensure that there is no use or deallocation statements after the inserted deallocation statement.

We denote data at each node n as (Db_n, U_d). Db_n is a set of allocations that may be deallocated after or at the current CFG node, and U_d is a set of allocations that may be referenced after or at the current CFG node. The meet operator is the set union of each component, and the transfer function at node n is $f2_n((Db, U))$:

$$f2_n((Db, U)) = (f1_n(Db), f2'_n(U)),$$

where $f2'_n(U) = \begin{cases} U \cup \{m_1, \ldots, m_k\}, & n \text{ labelled as } \texttt{Use}(m_1, \ldots, m_k), \\ U, & \text{otherwise.} \end{cases}$

(3) A forward data flow analysis to identify variables that point to leaked allocation chunk.

We first calculate that, at each CFG edge, the set of CFG nodes whose corresponding allocation chunks are safe to free. We use the following formula:

$$A_e = M_p - Df_{e.from} - Db_{e.to} - U_{e.to}.$$

https://engine.scichina.com/doi/10.1007/s11432-016-5572-2

We use $e.from$ to denote the head of e and use $e.to$ to denote the tail of e.

```
1   public void handleArgs(String[] args){
2       if (args[0].equals("-t")){
3           Hashtable<String,Integer>t
4               =new Hashtable<String,Integer>0;
5           for (int i=1; i<args.length; ++i)
6               t.put(args[i], i);
7           processTable(t);
8       }
9       else if (args[0].equals("-h"))
10          showHelp();
11      else if (args[0].equals("-v"))
12          showVersion();
13  }
```

```
            Framework evolution
rev.1   Changed Hashtable to HashTable
rev.2   Changed HashTableto HashMap
            Client evolution
13      else if (args[0].equals("-s")) {
14          f();
...     ...
            Test inputs
1:      args = { "-t" }
2:      args = { "-t", "a" }
3:      args = { "-t", "a", "b" }
4:      args = { "-h" }
5:      args = { "-v" }
```

Figure 2 A code snippet

Then we perform depth-first search in the points-to graph, and find the variable v, such that v points to a set of nodes s in the points-to graph, and $s \subseteq A_e$. Therefore, v is the variable that we use for the inserted deallocation statement.

(4) A forward data flow analysis that selects possibly early points to insert the deallocation statement.

To achieve this goal, we use a greedy algorithm. In particular, we record CFG nodes in the data set of each CFG edge during forward analysis. Once the data set of an edge contains the same CFG node, we filter the edge to avoid possible double free.

Last, after we have identified edges to fix, we insert the deallocation statement $free(v)$ at the corresponding program point, where v represents the variable pointing to the leaked allocation chunk.

We have implemented our approach as a tool, *LeakFix*. The evaluation shows that *LeakFix* is able to fix a substantial number of leaks in the SPEC2000 benchmark, and is scalable for large applications.

6 Demonstrative example

In this section, we study an example of a code snippet (shown in Figure 2), and illustrate the effectiveness of our approaches in the evolution of the code snippet. This example contains a method for processing program arguments. Five existing tests are provided to test the code snippet. The method first checks the first argument. If the argument equals "-t", the method puts the rest of the arguments into a hash table, and then calls another method to process the hash table. If the argument equals "-h", the method calls another method to print help information. If the argument equals "-v", the method calls another method to print the version of the program. Initially this code snippet has no faults. However, in the evolution process, there are two scenarios that may lead to faults for this code snippet:

(1) Framework evolution. The Java SDK may update `Hashtable` to `HashMap` via two revisions, and the existing code that uses `Hashtable` (Lines 3–4) will lead to a compilation error.

(2) Client evolution. There may be a new requirement to add a new argument option "-s", and a new feature is to be implemented for this option. Therefore, developers may add a new `if` block and a method call to `f()`. It is possible that the new block of the program contains new faults.

In the first scenario, the transformation rule changes `Hashtable` to `HashMap` in two revisions. We use natural language processing techniques to identify the changes in each revision, i.e., `Hashtable` is changed to `HashTable`, and `HashTable` is changed to `HashMap`. Then we aggregate the changes in both revisions to generate the transformation rule, which is changing `Hashtable` to `HashMap`. With the presented transformation rule, developers can write a transformation program using our transformation language. The transformation program is shown as follows:

https://engine.scichina.com/doi/10.1007/s11432-016-5572-2

```
(t : Hashtable ->> HashMap)
```

Given the code snippet in Figure 2 and the preceding transformation program, our approach can then automatically transform the code snippet to adapt to the new API.

In the second scenario, we aim to find potential faults via testing, and fix these faults via debugging. There are three steps for this scenario.

The first step is to generate tests. We prune all paths whose executions are guaranteed not to detect any regression fault, and guide symbolic execution to cover the newly added branch. In this way, we make symbolic execution scalable for generating tests that are effective in detecting regression faults, and generate the new test {"-s"}.

The second step is to optimize tests. During evolution, more tests in Figure 2 are added and the cost in testing increases. We select a portion of the tests by constructing a fault-detection-loss table by a preliminary study on the relation between the loss in fault detection and statement coverage, and then reschedule the execution order of tests. For example, we determine that test input 3 may have better fault-detection ability than test input 4 by leveraging previous execution information.

If we find a failure through testing, the third step is to locate the corresponding fault and fix it. In this code snippet, we aim to locate the potential fault in the method call to `f()` automatically. As described in Subsection 5.1, we can mutate the statements in `f()` to simulate real edits, which can help find the fault location more accurately.

7 Conclusion

Software continues to evolve, and programmers may introduce faults during software evolution. Therefore it is important to maintain high quality during software evolution. In our work, we explore approaches to assure high confidence in the case of framework evolution and client evolution. To correctly migrate client code, we utilize history information and design a programming language that describes differences between different versions of API framework code. In addition, we generate regression tests based on change differences between two versions using symbolic execution, and optimize tests mainly based on evolution information. Furthermore, we use evolution information for fault localization, and propose a static approach for automatically fixing memory leaks. The results show that such direction is promising and deserves further investigation.

Acknowledgements This work was supported by National Basic Research Program of China (Grant No. 2015AA01A202), National Natural Science Foundation of China (Grant Nos. 61421091, 61529201, 61272157, 61225007). Tao Xie's work was supported in part by National Science Foundation (Grant Nos. CCF-1409423, CNS-1434582, CCF-1434596, CNS-1513939, CNS-1564274), and a Google Faculty Research Award.

Conflict of interest The authors declare that they have no conflict of interest.

References

1 Godfrey M W, German D M. The past, present, and future of software evolution. In: Proceedings of Frontiers of Software Maintenance, Beijing, 2008. 129–138
2 Kemerer C, Slaughter S. An empirical approach to studying software evolution. IEEE Trans Softw Eng, 1999, 25: 493–509
3 Lehman M M, Ramil J F, Wernick P D, et al. Metrics and laws of software evolution - the nineties view. In: Proceedings of the 4th International Software Metrics Symposium, Albuquerque, 1997. 20–32
4 Böhme M, Oliveira B C d S, Roychoudhury A. Regression tests to expose change interaction errors. In: Proceedings of the 9th Joint Meeting on Foundations of Software Engineering. New York: ACM, 2013. 334–344
5 Dig D, Manzoor K, Johnson R, et al. Refactoring-aware configuration management for object-oriented programs. In: Proceedings of the 29th International Conference on Software Engineering, Minneapolis, 2007. 427–436
6 Harrold M J, Gupta R, Soffa M L. A methodology for controlling the size of a test suite. ACM Trans Softw Eng Methodol, 1993, 2: 270–285
7 Henkel J, Diwan A. Catchup! capturing and replaying refactorings to support API evolution. In: Proceedings of the 27th International Conference on Software Engineering. New York: ACM, 2005. 274–283
8 Ren X, Ryder B G. Heuristic ranking of Java program edits for fault localization. In: Proceedings of the International Symposium on Software Testing and Analysis. New York: ACM, 2007. 239–249

https://engine.scichina.com/doi/10.1007/s11432-015-5472-2

9 Santelices R, Harrold M J, Orso A. Precisely detecting runtime change interactions for evolving software. In: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation, Paris, 2010. 429–438

10 Stoerzer M, Ryder B G, Ren X, et al. Finding failure-inducing changes in Java programs using change classification. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York: ACM, 2006. 57–68

11 Wu W, Guéhéneuc Y-G, Antoniol G, et al. Aura: a hybrid approach to identify framework evolution. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering. New York: ACM, 2010. 325–334

12 Yoo S, Harman M. Using hybrid algorithm for pareto efficient multi-objective test suite minimisation. J Syst Softw, 2010, 83: 689–701

13 Zhang W, Yi L, Zhao H Y, et al. Feature-oriented stigmergy-based collaborative requirements modeling: an exploratory approach for requirements elicitation and evolution based on web-enabled collective intelligence. Sci China Inf Sci, 2013, 56: 082108

14 Kim M, Notkin D, Grossman D. Automatic inference of structural changes for matching across program versions. In: Proceedings of the International Conference on Software Engineering, Minneapolis, 2007. 333–343

15 Xing Z, Stroulia E. API-evolution support with Diff-CatchUp. IEEE Trans Softw Eng, 2007, 33: 818–836

16 Malpohl G, Hunt J, Tichy W. Renaming detection. In: Proceedings of the 15th IEEE International Conference on Automated Software Engineering, Grenoble, 2000. 73–80

17 Weissgerber P, Diehl S. Identifying refactorings from source-code changes. In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06), Tokyo, 2006. 231–240

18 Godfrey M, Zou L. Using origin analysis to detect merging and splitting of source code entities. IEEE Trans Softw Eng, 2005, 31: 166–181

19 Nguyen H A, Nguyen T T, Wilson G, et al. A graph-based approach to API usage adaptation. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. New York: ACM, 2010. 302–321

20 Meng S, Wang X, Zhang L, et al. A history-based matching approach to identification of framework evolution. In: Proceedings of the 34th International Conference on Software Engineering (ICSE), Zurich, 2012. 353–363

21 Li J, Wang C, Xiong Y, et al. SWIN: towards type-safe Java program adaptation between APIs. In: Proceedings of the Workshop on Partial Evaluation and Program Manipulation. New York: ACM, 2015. 91–102

22 Wang C L, Li J, Xiong Y F, et al. Formal Definition of SWIN Language. https://github.com/Mestway/SWIN-Project/blob/master/docs/pepm-15/TR/TR.pdf. 2014

23 Nita M, Notkin D. Using twinning to adapt programs to alternative APIs. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, New York: ACM, 2010. 205–214

24 Cordy J R. The TXL source transformation language. Sci Comput Program, 2006, 61: 190–210

25 Bravenboer M, Kalleberg K T, ermaas R V, et al. Stratego/XT 0.17. a language and toolset for program transformation. Sci Comput Program, 2008, 72: 52–70

26 Wasserman L. Scalable, example-based refactorings with refaster. In: Proceedings of the ACM Workshop on Workshop on Refactoring Tools. New York: ACM, 2013. 25–28

27 Erwig M, Ren D. A rule-based language for programming software updates. In: Proceedings of the ACM SIGPLAN Workshop on Rule-Based Programming. New York: ACM, 2002. 67–78

28 Erwig M, Ren D. An update calculus for expressing type-safe program updates. Sci Comput Program, 2007, 67: 199–222

29 Dig D, Negara S, Mohindra V, et al. Reba: refactoring-aware binary adaptation of evolving libraries. In: Proceedings of the 30th International Conference on Software Engineering. New York: ACM, 2008. 441–450

30 Leather S, Jeuring J, Löh A, et al. Type-changing rewriting and semantics-preserving transformation. In: Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. New York: ACM, 2014. 109–120

31 Saff D, Ernst M D. Reducing wasted development time via continuous testing. In: Proceedings of the 14th International Symposium on Software Reliability Engineering, Denver, 2003. 281–292

32 Voas J. PIE: a dynamic failure-based technique. IEEE Trans Softw Eng, 1992, 18: 717–727

33 Clarke L. A system to generate test data and symbolically execute programs. IEEE Trans Softw Eng, 1976, 2: 215–222

34 Csallner C, Smaragdakis Y. JCrasher: an automatic robustness tester for Java. Softw Pract Exp, 2004, 34: 1025–1050

35 Godefroid P, Klarlund N, Sen K. DART: directed automated random testing. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2005. 213–223

36 Pacheco C, Lahiri S K, Ernst M D, et al. Feedback-directed random test generation. In: Proceedings of the 29th International Conference on Software Engineering (ICSE'07), Minneapolis, 2007. 75–84

37 Sen K, Marinov D, Agha G. CUTE: a concolic unit testing engine for C. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York: ACM, 2005. 263–272

38 Tian T, Gong D-W. Test data generation for path coverage of message-passing parallel programs based on co-evolutionary genetic algorithms. Aut Softw Eng, 2014, 22: 1–32

39 Tillmann N, Halleux J de. Pex-white box test generation for .NET. In: Tests and Proofs. Berlin: Springer, 2008. 134–153

https://engine.scichina.com/doi/10.1007/s11432-016-5572-2

40 Tonella P. Evolutionary testing of classes. In: Proceedings of the ACM SIGSOFT International Symposium on

Software Testing and Analysis. New York: ACM, 2004. 119–128
41 Visser W, Păsăreanu C S, Khurshid S. Test input generation with Java pathfinder. In: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis. New York: ACM, 2004. 97–107
42 Zhang W-Q, Gong D-W, Yao X-J, et al. Evolutionary generation of test data for many paths coverage. In: Proceedings of Chinese Control and Decision Conference, Xuzhou, 2010. 230–235
43 Inkumsah K, Xie T. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, L'Aquila, 2008. 297–306
44 Taneja K, Xie T, Tillmann N, et al. eXpress: guided path exploration for efficient regression test generation. In: Proceedings of the International Symposium on Software Testing and Analysis. New York: ACM, 2011. 1–11
45 Taneja K, Xie T, Tillmann N, et al. Guided path exploration for regression test generation. In: Proceedings of the 31st International Conference on Software Engineering, Vancouver, 2009. 311–314
46 Marinescu P D, Cadar C. Make test-zesti: a symbolic execution solution for improving regression testing. In: Proceedings of the 34th International Conference on Software Engineering. Piscataway: IEEE Press, 2012. 716–726
47 Böhme M, Oliveira B C d S, Roychoudhury A. Partition-based regression verification. In: Proceedings of the International Conference on Software Engineering, Piscataway, 2013. 302–311
48 Chipounov V, Georgescu V, Zamfir C, et al. Selective symbolic execution. In: Proceedings of the 5th Workshop on Hot Topics in System Dependability, Lisbon, 2009. 1–6
49 Chipounov V, Kuznetsov V, Candea G. The S2E platform: design, implementation, and applications. ACM Trans Comput Syst, 2012, 30: 1–49
50 Fraser G, Arcuri A. Sound empirical evidence in software testing. In: Proceedings of the 34th International Conference on Software Engineering (ICSE), Zurich, 2012. 178–188
51 Jaygarl H, Kim S, Xie T, et al. OCAT: object capture-based automated testing. In: Proceedings of the 19th International Symposium on Software Testing and Analysis. New York: ACM, 2010. 159–170
52 Xiao X, Li S, Xie T, et al. Characteristic studies of loop problems for structural test generation via symbolic execution. In: Proceedings of IEEE/ACM 28th International Conference on Automated Software Engineering (ASE), Silicon Valley, 2013. 246–256
53 Xiao X, Xie T, Tillmann N, et al. Precise identification of problems for structural test generation. In: Proceedings of the 33rd International Conference on Software Engineering. New York: ACM, 2011. 611–620
54 Thummalapenta S, Xie T, Tillmann N, et al. Synthesizing method sequences for high-coverage testing. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. New York: ACM, 2011. 189–206
55 Thummalapenta S, Xie T, Tillmann N, et al. MSeqGen: object-oriented unit-test generation via mining source code. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. New York: ACM, 2009. 193–202
56 Qi D, Sumner W N, Qin F, et al. Modeling software execution environment. In: Proceedings of the 19th Working Conference on Reverse Engineering, Kingston, 2012. 415–424
57 Samimi H, Hicks R, Fogel A, et al. Declarative mocking. In: Proceedings of the International Symposium on Software Testing and Analysis. New York: ACM, 2013. 246–256
58 Zhang L, Ma X, Lu J, et al. Environmental modeling for automated cloud application testing. IEEE Softw, 2012, 29: 30–35
59 Godefroid P, Luchaup D. Automatic partial loop summarization in dynamic test generation. In: Proceedings of the International Symposium on Software Testing and Analysis. New York: ACM, 2011. 23–33
60 Xie T, Tillmann N, de Halleux P, et al. Fitness-guided path exploration in dynamic symbolic execution. In: Proceedings of IEEE/IFIP International Conference on Dependable Systems & Networks, Lisbon, 2009. 359–368
61 Böhme M. Automated regression testing and verification of complex code changes. Dissertation for Ph.D. Degree. Singapore: National University of Singapore, 2014
62 Jones J A, Harrold M J. Test-suite reduction and prioritization for modified condition/decision coverage. IEEE Trans Softw Eng, 2003, 29: 195–209
63 Jeffrey D, Gupta N. Improving fault detection capability by selectively retaining test cases during test suite reduction. IEEE Trans Softw Eng, 2007, 33: 108–123
64 Lin J-W, Huang C-Y. Analysis of test suite reduction with enhanced tie-breaking techniques. Inf Softw Tech, 2009, 51: 679–690
65 Chen T Y, Lau M F. A new heuristic for test suite reduction. Inf Softw Tech, 1998, 40: 347–354
66 Hao D, Zhang L, Wu X, et al. On-demand test suite reduction. In: Proceedings of the 34th International Conference on Software Engineering, Piscataway, 2012. 738–748
67 Do H, Rothermel G. A controlled experiment assessing test case prioritization techniques via mutation faults. In: Proceedings of the 21st IEEE International Conference on Software Maintenance, Budapest, 2005. 411–420
68 Elbaum S, Malishevsky A, Rothermel G. Prioritizing test cases for regression testing. In: Proceedings of International Symposium of Software Testing and Analysis, Portland, 2000. 102–112
69 Elbaum S, Malishevsky A, Rothermel G. Incorporating varying test costs and fault severities into test case prioritization. In: Proceedings of the 23rd International Conference on Software Engineering, Washington, 2001. 329–338
70 Hou S S, Zhang L, Xie T, et al. Quota-constrained test-case prioritization for regression testing of service-centric systems. In: Proceedings of IEEE International Conference on Software Maintenance, Beijing, 2008. 257–266

https://engine.scichina.com/doi/10.1007/s11432-015-5572-2

71 Jiang B, Zhang Z, Chan W K, et al. Adaptive random test case prioritization. In: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, 2009. 257–266
72 Li Z, Harman M, Hierons R. Search algorithms for regression test case prioritisation. IEEE Trans Softw Eng, 2007, 33: 225–237
73 Zhang L, Hou S, Guo C, et al. Time-aware test-case prioritization using integer linear programming. In: Proceedings of the 18th International Symposium on Software Testing and Analysis. New York: ACM, 2009. 213–224
74 Hao D, Zhang L, Zhang L, et al. A unified test case prioritization approach. ACM Trans Softw Eng Methodol, 2014, 24: 1–31
75 Zhang L, Hao D, Zhang L, et al. Bridging the gap between the total and additional test-case prioritization strategies. In: Proceedings of the 35th International Conference on Software Engineering (ICSE), San Francisco, 2013. 192–201
76 Hao D, Zhao X, Zhang L. Adaptive test-case prioritization guided by output inspection. In: Proceedings of the 37th Annual Computer Software and Applications Conference (COMPSAC), Kyoto, 2013. 169–179
77 Mei H, Hao D, Zhang L, et al. A static approach to prioritizing JUnit test cases. IEEE Trans Softw Eng, 2012, 38: 1258–1275
78 Jones J A, Harrold M J, Stasko J. Visualization of test information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering. New York: ACM, 2002. 467–477
79 Hao D, Zhang L, Pan Y, et al. On similarity-awareness in testing-based fault localization. Aut Softw Eng, 2008, 15: 207–249
80 Liblit B, Naik M, Zheng A X, et al. Scalable statistical bug isolation. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2005. 15–26
81 Yu Y, Jones J A, Harrold M J. An empirical study of the effects of test-suite reduction on fault localization. In: Proceedings of the 30th International Conference on Software Engineering. New York: ACM, 2008. 201–210
82 Abreu R, Zoeteweij P, van Gemund A J C. On the accuracy of spectrum-based fault localization. In: Proceedings of Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, Windsor, 2007. 89–98
83 Papadakis M, Traon Y L. Using mutants to locate "unknown" faults. In: Proceedings of IEEE 5th International Conference on Software Testing, Verification and Validation, Montreal, 2012. 691–700
84 Zhang X, Gupta N, Gupta R. Locating faults through automated predicate switching. In: Proceedings of the 28th International Conference on Software Engineering. New York: ACM, 2006. 272–281
85 Jeffrey D, Gupta N, Gupta R. Fault localization using value replacement. In: Proceedings of the International Symposium on Software Testing and Analysis. New York: ACM, 2008. 167–178
86 Zhang S, Zhang C, Ernst M D. Automated documentation inference to explain failed tests. In: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lawrence, 2011. 63–72
87 Xuan J, Monperrus M. Test case purification for improving fault localization. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York: ACM, 2014. 52–63
88 Zeller A. Yesterday, my program worked. today, it does not. why? In: Software Engineering — ESEC/FSE'99. Berlin: Springer, 1999. 253–267
89 Zeller A, Hildebrandt R. Simplifying and isolating failure-inducing input. IEEE Trans Softw Eng, 2002, 28: 183–200
90 Zhang L, Kim M, Khurshid S. Localizing failure-inducing program edits based on spectrum information. In: Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM), Williamsburg, 2011. 23–32
91 Alves E, Gligoric M, Jagannath V, et al. Fault-localization using dynamic slicing and change impact analysis. In: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering, Washington, 2011. 520–523
92 Zhang L, Zhang L, Khurshid S. Injecting mechanical faults to localize developer faults for evolving software. In: Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications. New York: ACM, 2013. 765–784
93 Kim D, Nam J, Song J, et al. Automatic patch generation learned from human-written patches. In: Proceedings of the International Conference on Software Engineering. Piscataway: IEEE Press, 2013. 802–811
94 Goues C L, Nguyen T, Forrest S, et al. Genprog: a generic method for automatic software repair. IEEE Trans Softw Eng, 2012, 38: 54–72
95 Nguyen H D T, Qi D, Roychoudhury A, et al. Semfix: program repair via semantic analysis. In: Proceedings of the International Conference on Software Engineering. Piscataway: IEEE Press, 2013. 772–781
96 Qi Y, Mao X, Lei Y, et al. The strength of random search on automated program repair. In: Proceedings of the 36th International Conference on Software Engineering. New York: ACM, 2014. 254–265
97 Nguyen T T, Nguyen H A, Pham N H, et al. Recurring bug fixes in object-oriented programs. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering. New York: ACM, 2010. 315–324
98 Coker Z, Hafiz M. Program transformations to fix C integers. In: Proceedings of the International Conference on Software Engineering. Piscataway: IEEE Press, 2013. 792–801
99 Jin G, Song L, Zhang W, et al. Automated atomicity-violation fixing. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2011. 389–400
100 Jin G, Zhang W, Deng D, et al. Automated concurrency-bug fixing. In: Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, Hollywood, 2012. 221–236
101 Li H H, Qi J, Liu F, et al. The research progress of fuzz testing technology (in Chinese). Sci China Inform, 2014,

https://engine.scichina.com/doi/10.1007/s11432-015-5572-2

44: 1305–1322
102 Xie T, Zhang L, Xiao X, et al. Cooperative software testing and analysis: advances and challenges. J Comput Sci Tech, 2014, 29: 713–723
103 Cherem S, Princehouse L, Rugina R. Practical memory leak detection using guarded value-flow analysis. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2007. 480–491
104 Hackett B, Rugina R. Region-based shape analysis with tracked locations. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York: ACM, 2005. 310–323
105 Heine D L, Lam M S. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2003. 168–181
106 Jung Y, Yi K. Practical memory leak detector based on parameterized procedural summaries. In: Proceedings of the 7th International Symposium on Memory Management. New York: ACM, 2008. 131–140
107 Sui Y, Ye D, Xue J. Static memory leak detection using full-sparse value-flow analysis. In: Proceedings of the International Symposium on Software Testing and Analysis. New York: ACM, 2012. 254–264
108 Torlak E, Chandra S. Effective interprocedural resource leak detection. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering. New York: ACM, 2010. 535–544
109 Xie Y, Aiken A. Context- and path-sensitive memory leak detection. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York: ACM, 2005. 115–125
110 Gao Q, Xiong Y, Mi Y, et al. Safe memory-leak fixing for C programs. In: Proceedings of the 37th IEEE International Conference on Software Engineering (ICSE), Florence, 2015. 459–470
111 Lattner C, Lenharth A, Adve V. Making context-sensitive points-to analysis with heap cloning practical for the real world. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2007. 278–289
112 Hardekopf B, Lin C. Semi-sparse flow-sensitive pointer analysis. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York: ACM, 2009. 226–238
113 Hardekopf B, Lin C. Flow-sensitive pointer analysis for millions of lines of code. In: Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Chamonix, 2011. 289–298
114 Wang J, Ma X-D, Dong W, et al. Demand-driven memory leak detection based on flow- and context-sensitive pointer analysis. J Comput Sci Tech, 2009, 24: 347–356
115 Kam J, Ullman J. Monotone data flow analysis frameworks. Act Inf, 1977, 7: 305–317

https://engine.scichina.com/doi/10.1007/s11432-016-5572-2